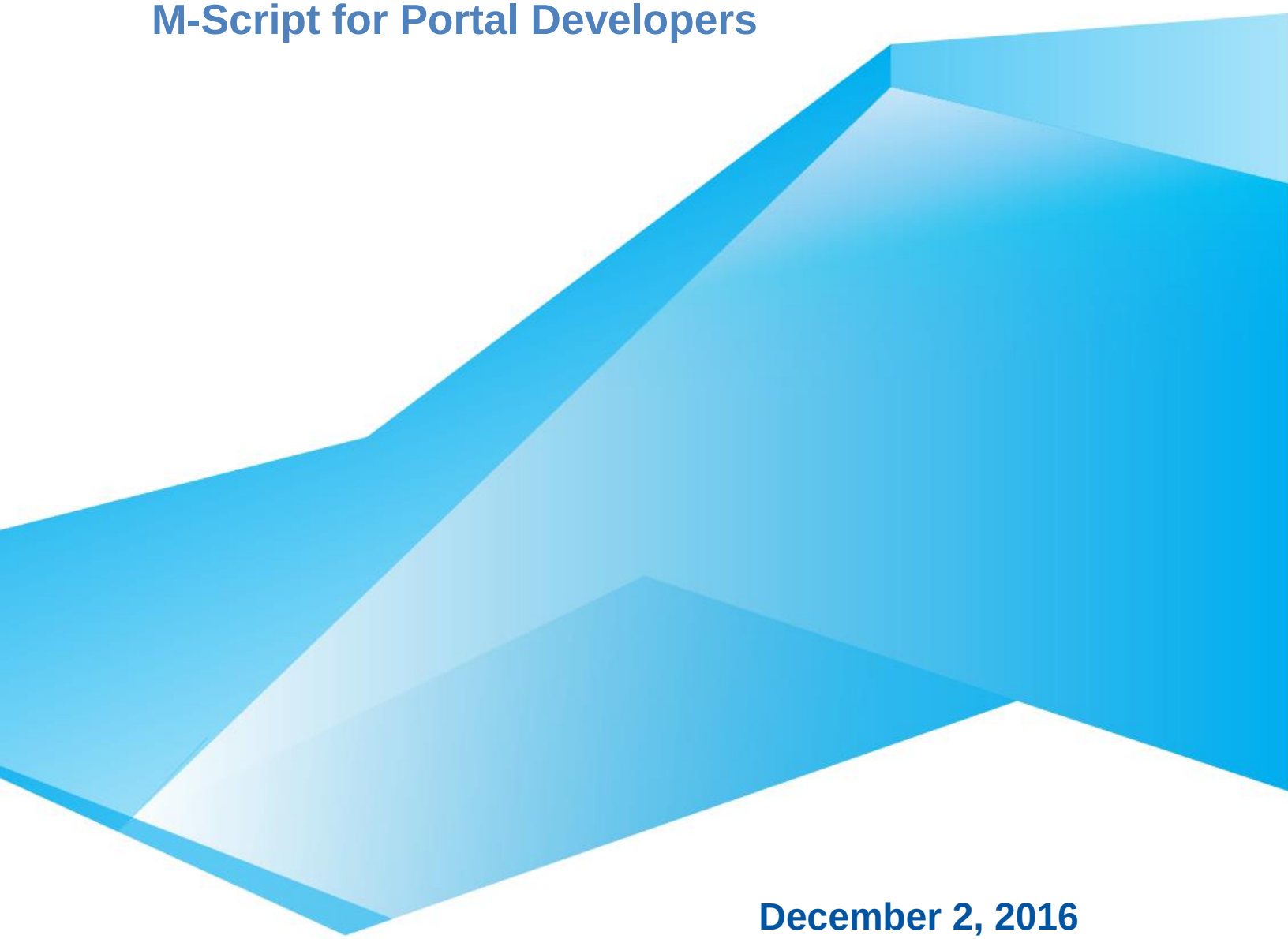
Deltek Know more. Do more.™

Deltek Maconomy 2.3 GA

M-Script for Portal Developers



December 2, 2016

While Deltek has attempted to verify that the information in this document is accurate and complete, some typographical or technical errors may exist. The recipient of this document is solely responsible for all decisions relating to or use of the information provided herein.

The information contained in this publication is effective as of the publication date below and is subject to change without notice.

This edition published December 2016.

Contents

About This Document

This document contains five sections about the M-Script language.

- The **M-Script Language Reference** describes the basic M-Script language constructs.
- The **M-Script Maconomy API Reference** describes how MScript interfaces with the Maconomy application.
- The **M-Script GUI II API Reference** describes how graphical user interface (GUI) components as seen in the Portal are created using M-Script. This section describes the functionality of the MScript Maconomy API that is supported by MScript 18.10 and MaconomyServer 46.00.
- The **M-Script Favorite Values** section describes how server-side M-Script can be used for creating a list of Favorite values in Maconomy dialogs to simplify data entry.
- The **M-Script Standard Packages** section describes how you can extend the Maconomy M-Script language by using packages.

Part 1 – M-Script Language Reference

The Maconomy M-Script language is designed for generating dynamic Web pages based on data from a Maconomy system. This section describes the basic M-Script language constructs.

Structure of an M-Script

All M-Scripts must begin with a line that contains a hash mark and the string `version` followed by the version number of the expected M-Script interpreter:

```
#version 18.10 // Latest MScript version is '18.10'
```

The version number is read by the M-Script interpreter and compared to its own version number. If these do not match—that is, the stated version number is greater than the interpreter's—the interpreter will not execute the M-Script. The version number may also contain a minor version number, for instance 15.0.

After the version line it is possible to have a line that specifies the encoding that should be used when the script is parsed:

```
#encoding UTF8
```

Because variables and identifiers are restricted to ASCII letters and digits in Maconomy 2.1 and onwards, this encoding is only relevant if the script contains string literals in an encoding that is not compatible with the default encoding (ISO88591).

Next follows the actual M-Script code. This code is executed from the first line and forward, and so a `main` function as in the C/C++ language is not required.

The most common M-Script command is probably the `print()` function. This function takes one or more arguments and prints them to M-Script's standard output:

```
print("Hello world!<br>");
```

If more than one argument is passed to the `print()` function, these can be referred to with the `^n` argument reference—the value of `n` refers to the position of the argument, starting from one with the first argument after the formatting string:

```
print("^1 ^2!<br>", "Hello", "World");
```

The same output could also be generated in this way:

```
print("^2 ^1!<br>", "World", "Hello");
```

Another useful function is `dumpvalue()`. This function takes one argument of any type and prints a readable representation of it.

M-Script automatically adds a `content-type` line to all of the output that is generated from M-Script to release the programmer from that job. In addition to this it also adds an `expires` date, `last-modified` date, and a `content-length` field to the HTTP header.

Starting Stand-Alone M-Script

An M-Script can be executed directly from the command line in this way:

```
MaconomyMScript.exe scriptname.ms
```

Make sure that you have one of the following environment variables set; either `TMP`, `TEMP`, or `MaconomyTmpDir` must point to a place where temporary files can be stored. If none of these can be set, a `TmpDir` entry must be specified in M-Script's initialization file.

If needed, a query string can be supplied with the `-q` option:

```
maconomy.exe -q querystring scriptname.ms
```

The version of the M-Script interpreter can be obtained using the `-v` option.

To execute and MScript via the web server, complete the following steps:

1. Create or find a CGI directory for the web server. This directory must be flagged as executable for the web server.
2. Move the MScript interpreter program into the CGI directory.
3. Create or find a script directory in which the MScripts should be stored.

   This should, for security reasons, not be the same as the CGI directory. In fact, it should not even be part of the web server's root, in which case web browsers would be able to read and view the MScript code.
4. Create an initialization file for the MScript interpreter in the CGI directory. Make sure that it contains at least a search path, a server address, a daemon port number, and a listen port number A simple file could look like this:

   ```
   searchpath = /home/www/mscript
   serverIP   = www.home.com
   DaemonPort = 4101
   ListenFrom = 4200
   ```
5. Make sure that the environment variables `TMP`, `TEMP`, or `MaconomyTmpDir` are available to the web server, and thereby to the MScript interpreter.
6. Point the browser to the interpreter program and add the name of a script in the URL. If, for example, the CGI directory is named `cgi-bin` and the script is named `intro.ms`, the URL should be something like the following:

   ```
   http://myhost/cgi-bin/MaconomyMScript.exe/intro.ms
   ```

Sessions are stored locally on the disk where M-Script resides. This may sometimes need to be cleaned out—typically when a web user has closed his or her web connection instead of logging out correctly. Normally this is done automatically by M-Script. (See the description of `SessionCleanUpProbability`). However, if the M-Script interpreter is started with a `c` argument, it cleans out all session files that are older than a certain number of days. The default value is two days, but other values can be specified in the initialization file.

Note that stand-alone execution of M-Script is not the only way to execute an M-Script.

Command-Line Options

Option	Description
`-v` `--version`	Print version info and exit.
`-vv` `--vversion`	Show extended version information, such as component version numbers and build time.
`-q str`	Start interpreter with query string "str." The query string should only contain `key=val` elements separated with `&`. The actual script name should not be part of the string.
`-c`	Clean up old session files. When invoked on Windows

Option	Description
`--cleanup`	like this, the M-Script interpreter removes all session files that are older than the time that is specified in the initialization file using the `SessionLifeTime` tag. On Unix this option also makes M-Script do a cleanup of old session files. But in addition to this it also spawns an M-Script cleanup daemon that runs until the system is rebooted.

`-p [label | ?]`
`--ping [label | ?]`
`--pingall`
	This option makes M-Script try to contact a Maconomy server while at the same time it prints diagnostic information. This can be quite useful for debugging the Server > webdaemon > M-Script connection. The `-p/--ping` command contacts the server with the specified server label. If no server label is given, the default server is assumed. If you specify a question mark instead of a server label, all available servers are displayed. `--pingall` contacts all defined servers one by one.

`-e` `--noheaders`	Omit html-headers in output. This option is useful when the generated output is not intended for a browser. Also it disables output buffering, which makes debugging easier. This in turn disables the use of the built-in procedure `discardoutput()`.
`-I file` `--inifile file`	Use the file `<file>.I` as initialization file.
`-x` `--noexecution`	Prevents M-Script from actually executing the script. In this mode M-Script only parses the scripts, verifies package dependencies, and resolves identifier references.
`-d v=s` `--define v=s`	Set environment variable “v” to the string “s.” Passing the empty string for “s” will delete “v.”
`--crypt code`	Set encryption for server communication. Legal values for `code` are: ▪ `default`: Use default encryption as decided by the server. ▪ `none`: Do not use encryption (for example, while debugging). ▪ `ssl`: Use SSL encryption. ▪ `simple`: Use simple in-place encryption.

Exit Codes

When the M-Script interpreter exits, it returns either zero to the calling program (if everything went well) or 255 if any run-time or parse error or similar occurred. Other return codes can be specified inside an M-Script program by the use of either a `return` statement or a call to the `exit` function.

Reading this Section

In a few places where the formal syntax of the M-Script language is given by its BNF, a special list template is used to avoid complex BNF productions every time that a list of something is needed. The notation is `x-list` or `x-list[,]` when a list of the type `x` is needed. The token in the brackets is used as a delimiter for the list. The template definition is:

```
x-list ::=
  x
  | x-list x


x-list[,] ::=
  x
  | x-list , x
```

The two templates are used to define space-separated lists of elements, for instance the sequence “a b c d e”, and comma-separated lists, for instance “a,b,c,d,e.”

For every function or procedure in this section and the M-Script Maconomy API Reference, a note of the permitted contexts is indicated for each function or procedure. The allowed contexts are specified in the “Context” section.

For some packages, the context specification applies to all functions and procedures in the package. In those cases, the allowed context is specified at the top of the package description.

An example of context specification for the function `currentsession()`:

Context Stand-alone, Server command-line.

This line means that the function `currentsession()` is available in the following contexts:

- Standalone execution scripts
- Script issued on the server command line
- And nowhere else

The contexts are described further in the previous section.

Note that for all `gui::` functions (in the M-Script GUI-II API Reference), only one context is permitted: stand-alone execution. For this reason, the `gui::` functions do not include a Context section.

M-Script Execution Contexts

M-Script is used in many different contexts in Maconomy; the stand-alone execution as described previously is only one of these contexts. Actually, M-Script is often run by the server (as opposed to the web server).

The range of functions that are available in M-Script depends on the current context. For instance, when using M-Script for stand-alone execution, the full range of M-Script functions is available, but when using M-Script for calculating favorites, only a limited number of M-Script functions and features are available. Therefore, it is important to be aware of the context in which your M-Scripts are to be run.

The following is a general overview of the different execution contexts of M-Script. The permitted context is indicated in the document.

- **Stand-alone execution** — M-scripts executed by `MaconomyMScript.exe` (Windows) or `MaconomyMScript` (Unix).
- **Server command-line execution** — MScripts executed by the `MaconomyServer` executable. See “Server Options” in the System Administrator Guide. The ability to run MScript from the Maconomy server was added in TPU 9.0p5.
- **Server custom actions** — MScript package procedures can be run by custom actions using the Maconomy extension languages.
  - Guardscripts and visiblescripts
  - The custom actions as such

  For more information about custom actions, see the Maconomy Extension Languages Reference.
- **Server workflows** — Guardscripts, pre-scripts, post-scripts, and notification receiver scripts in connection with the execution of workflow transitions. For more information about workflows, see the Maconomy Workflow Language Reference.
- **Server print layouts (MPL)** — In MPL layouts, you can execute M-Scripts for calculating various attributes, such as colors and typeface, for instance to be used in connection with traffic lighting. For more information, see “MPL Reference” in the System Administrator Guide.
- **Server report execution (MRL)** — When running MRL reports, you can execute M-Script pre-scripts and post-scripts for validating and updating parameter values.
- **Server favorites** — Executing packages for the calculation of favorites—that is, entries such as frequently used job numbers in combo boxes in the Maconomy client for Java™ and for the Portal. For more information, see the M-Script Favorite Values section.

Overview

The Maconomy M-Script language is designed for generating dynamic web pages based on data from a Maconomy system.

An M-Script program, or simply an M-script, is executed, or rather interpreted, by a web server component when a user sends a request for a given web page. The M-Script interpreter executes the M-script and returns the output to the user's web browser.

The M-Script language is based on the syntax and semantics of the JavaScript language; it might, therefore, be useful to read an introductory book on JavaScript before reading this M-Script reference document.

M-Script is in many ways similar to other web programming languages such as PHP, ASP, PERL, and serverside Java, so why use M-Script at all? Why was M-Script invented? There are a number of good reasons for this:

- M-Script is platform independent and runs on Windows NT, IBM AIX, HP UNIX, Linux, and Solaris. This fact rules out the use of all Microsoft products.
- M-Script integrates seamlessly with Maconomy data. This means that you get Maconomy's access control, access to all Maconomy's dialogs, and access to Analyzer reports, and so forth, all as an integral part of the language. This could be difficult to obtain in the other public domain languages.
- M-Script is, as opposed to Java, designed to be easy to learn, and does not require highly specialized programmers who have a solid background in object oriented programming.
- M-Script is efficient because there is no need to waste server time by going through special COM/DCOM objects or similar to access Maconomy data.

M-Script and HTML

One of the main features of M-Script is its ability to seamlessly interact with standard HTML pages. This can be done in two ways—you can write stand-alone M-Scripts that simply use the `print` function to generate HTML output from scratch. This may be preferable in cases where a lot of M-Script logic is needed to generate very little HTML output. On the other hand, you can also embed M-Script code inside HTML comments or the special <% and %> markers. When used inside HTML comments there must be an M-Script marker as the first thing that is found inside the comments. This marker is not case sensitive, so for instance, both `mscript` and `MSCriPt` work. A small example is shown in the following:

```
<html>
  <body>
  <!--MScript #version 15
     print("Hello World"); -->
  <% print("<br>Hello again"); %>
  </body>
</html>
```

The next step is to tell the M-Script interpreter whether it should look for HTML code or M-Script code from the beginning. The exact mode is defined by the filename extension, but only if the `.I` file has been set up with sections for `.hms` and `.ms` files.

Extension	Mode
`.ms`	M-Script file
`.hms`	M-Script embedded in HTML
`any other`	M-Script file

M-Script Embedded in HTML

When M-Script code is embedded in HTML with the `<!--MScript ... ->` tags, the surrounding HTML code is considered a `print` statement. With this convention it is possible to allow branch and loop constructions in M-Script to span HTML code. It is, for instance, possible to repeat a line of HTML code ten times as the following example shows:

```
<!--MScript for (var n=0 ; n<10 ; ++n) { -->
    Hello<br>
<!--MScript } -->
</body>
</html>
```

The enclosing braces `{` and `}` are not always mandatory—all of the HTML code between the first and second block of M-Script is considered one single `print` statement, no matter how many lines it spans. If the second M-Script block in this example had been left out, the `</body>` and `</html>` tags would have been repeated with the “Hello” line.

Lexical Structure

Identifiers

Identifiers consist of the letters a-z, A-Z, underscores, and digits. Identifiers must start with a letter or an underscore.

Comments

Use `//` for one-line comments and `/* ... */` for multi-line comments. Examples:

```
// This is a one-line comment
// This is another one-line comment
/* This is a
   multi-line comment */
```

“Here” Texts

“Here” texts are convenient for building large strings in M-Script. The name may seem a bit odd, but it means something like “the text is here right after the line,” and the term comes originally from older Unix languages like Shell Script and Perl. A “here” text consists of a named placeholder that is defined in single quotes. The first line following any line with a “here” placeholder is considered to be the first line in the “here” text. The “here” text ends with the first line thereafter that contains exactly the name of the placeholder (with optional leading white space). Example:

```
print('QED');
Here is the first line of 'here' text
This is the second line
Now comes the end marker
  QED
```

“Here” texts can be used anywhere in an M-Script expression where a string is expected. It is also possible to use more than one “here” text in the same expression. To do this, the content of the “here” texts must follow immediately after each other. Example (note that strings can be concatenated with the + operator):

```
print('QED1' + 'QED2');
Here is 'here' text one.
  QED1
And here's 'here' text two.
QED2
```

“Here” texts are not allowed to overlap with M-Script comments, `<--MScript` or `-->` tags.

Escaped Characters

The following character escape sequences can be used in all strings (except “here” texts).

Sequence	Meaning
`\n`	newline

Sequence	Meaning
\t	horizontal tab
\\	backslash
\"	double quote
\xHH	two-digit hexadecimal number representation of a character

Including Other M-Scripts

You can include other M-scripts with the `#include` command. This command should be followed by a filename enclosed in either angles `<...>` or quotes `"..."` If the filename is enclosed in angles, the M-Script interpreter only searches through the search path to find the file. If, on the other hand, the filename is enclosed in quotes, the interpreter first looks in the same directory in which the including file is placed, and thereafter in the search path.

Note that M-Script has been extended with a package feature that is better suited for building complex M-Script programs.

The following example illustrates the difference between angles and quotes in the include command. Here it is possible to include the file `draw.ms` from the file `graph.ms` even though it is not in the search path.

```
sources
|
+--main.ms (#include <html/graph.ms>
|
+-- html
    |
    +-- graph.ms (#include "draw.ms")
    |
    +-- draw.ms
```

Filename and Line Number

You can access the name of the current M-Script via the predefined variable `__FILE__`. In the same way You can also get the line number of the current statement via the variable `__LINE__`. `__FILE__` returns the file name of the script or package, possibly prepended by the name of the root from which the package is loaded, or by a "`./`" if the package is loaded from one of the default search paths. For the main script and packages loaded from the default package root, nothing is prepended. For example, consider these scripts:

- `main.ms` is loaded from a search path.
- `main.1.ms` is a package loaded from the search path.
- `main.2.ms` is a package loaded from the default package root.
- `main.3.ms` is a package loaded from the named package root "`p`."

The corresponding values of __FILE__ in each of these scripts are:

```
main.ms
./main.1.ms
main.2.ms
p/main.3.ms
```

If a package is accessed through multiple package roots in the same execution (possible if one package root overlaps another), the value of __FILE__ is decided by the package root through which the package is first loaded. In other words, be careful with overlapping package routs if the value of __FILE__ is critical. Also, if the name and path of the script are used to open a file in the same location, the structure of the `FileSystemRoot` variable should match the structure of the `PackageRoot` variable.

Data Types

The basic M-Script types are the following.

Type	Syntax	Description
Null	`null`	A distinguished type—and value. The `null` value is interpreted in such a way that any arithmetic operation that involves a `null` value, always evaluates to a `null` value.
Boolean	`bool`	Can be either `true` or `false`.
Integer	`int`	The basic integer number.
Real	`real`	The basic floating point number.
Amount	`amount`	Fixed-point number with two decimals for representing currency.
Date	`date`	Date.
Time	`time`	Time of day.
String	`string`	Text string.
Bytes	`bytes`	Bytes, could be binary data.
Object	`object`	Dynamic records.
Array	`array`	Dynamic arrays.
Function	`function`	Function (as well as procedure) pointer.
EnumRef	`user defined`	Reference to the type of an enumerate set.
Enum	`user defined`	The literal members defined in `enum` statements.
Generic pointer	`none`	Generic pointers are used to represent various internal values such as streams, network sockets, regular expressions, and Dialog records in the API. Most generic pointers cannot be operated upon with any of the standard operators.

Short Description of the Types

Null

The `null` type (and value) can be used to represent non-existing information or error values such as an empty data field or an illegal computation. If `null` is used in an expression, the output of that expression is also `null`. This means for instance that `5 + null == null`. Some expressions

do not obey these rules, however. These are the Boolean `&&` and `||` operators and the conditional `? :` expression, which all are conditionally evaluated.

A `null` value cannot be used in a conditional expression such as found in for instance an `if`-statement.

Booleans

The Boolean data type only has two values, namely `true` and `false`, representing the “truth” of an expression. Booleans are used in conditional branches and loop constructions to control the program execution. Booleans are generally the result of a comparison between two elements, for instance `x == y`.

Booleans are automatically converted to integers and can be used as such. So it is legal to write an expression such as `x + 5*y`. In this case the false value is converted to zero and the true value is converted to one.

Integers

Integers should require quite little explanation. They can be used together with most operators and should be preferred over reals whenever it is possible—mostly for reasons of precision. Integers are automatically converted to reals when necessary.

Integer literals consists of the digits 0..9 only. Example: `42`.

Integers can also be written in hex notation as `\xNNN`. Examples: `\xFF` or `\x7A10`.

Reals

Reals are used to represent floating point numbers and can be used to represent values much larger than integers. The range is ±1.798¥10308.

Real literals are in the format '`xxx.yyy`' or 'xxx.yyyEzz' where 'zz' is the base-10 exponent. Examples: `3.1415 and 3.14e3` (which is the same as 3140.0).

Amounts

The amount type is used to represent a currency in M-Script. Amounts are implemented as 64-bit integers, where the last two digits always represent the smallest unit of currency. Amounts can be added and subtracted, as well as multiplied with either integers or reals.

Amount literals are in the format '`xxx.yyA`'. In order to distinguish amounts from reals they must be appended with an 'A'. Example: `9.95A`.

Dates

The date type is used to represent dates in M-Script. Dates can be added with integers or reals, in which case the added value is considered a number of days. Dates can also be subtracted to find the number of days between them.

Date literals are in the format '`DD.MM.YYYY`' where DD = days, MM = months, and

YYYY=years (all four digits). Example: `25.2.2000`

Date literals are only used for the internal representation of dates—the external representation (as presented on the Web) can be defined in the initialization file.

A special null-date exists. This date represents empty date fields in the Maconomy application and lies earlier in time than any other date. The null-date can be archived by a type conversion from the null value to a date type. Null-dates are always represented externally as an empty space.

Times

The time type is used to represent a time of the day in M-Script. Times can be added with integers or reals, in which case the added value is considered a number of seconds. Times can also be subtracted to find the number of seconds between them.

Time literals are in the format '`HH:MM:SS`' where HH = hours, MM = minutes, and SS = seconds. Example: `22:05:59`.

Time literals are only used for the internal representation of time—the external representation (as presented on the web) can be defined in the initialization file.

A special null-time exists. This time represents empty time fields in the Maconomy application and lies earlier in time than any other time. The null-time can be archived by a type conversion from the null value to a time type. Null-times are always represented externally as an empty space.

Strings

Strings are used to store texts in M-Script. Strings may be concatenated with the + operator, which means for instance that "`aaa`" + "`bbb`" becomes "`aaabbb`". Strings are immutable which means it is impossible to modify a string once it is created. This means you always have to create new strings instead of modifying already existing ones.

Indexing

Strings can be indexed for read-only with the index operator [..]. Indices start from zero:

String	Value	Description
`a`	`"abcdefgh"`	Whole string
`a[0]`	`"a"`	Indexed substring of length 1
`a[1..3]`	`"bcd"`	Indexed substring
`a[2..]`	`"cdefgh"`	Trailing substring
`a[..2]`	`"abc"`	Leading substring
`a[3..2]`	`""`	Empty substring

Indexing out of bounds results in a run-time error.

Bytes

Bytes are used to store binary data in M-Script, for instance PDF data. Bytes may be concatenated with the + operator, which means for instance that "`aaa`" + "`bbb`" becomes "`aaabbb`". Bytes are immutable, which means it is impossible to modify a byte variable after it is created.

Indexing

Bytes can be indexed for read-only with the index operator [..]. Indices start from zero:

String	Value	Description
`b`	`"abcdefgh"`	All the bytes
`b[0]`	`"a"`	Indexed subrange of length 1

String	Value	Description
`b[1..3]`	`"bcd"`	Indexed subrange
`b[2..]`	`"cdefgh"`	Trailing subrange
`b[..2]`	`"abc"`	Leading subrange
`b[3..2]`	`""`	Empty subrange

Indexing out of bounds results in a run-time error.

Type Casts

Values of different types may, for most combinations of types, be converted to and from each other. This is what is called a type cast. The syntax for a type cast is:

```
typeCast ::= typeExpr ( expr )
  typeExpr ::=
    null
  | bool
  | int
  | real
  | amount
  | time
  | date
  | string
  | bytes
```

Objects, arrays, and functions cannot be cast to anything.

The following table lists all possible type casts.

	Destination								
Source	null	bool	int	real	amount	time	date	string	bytes
null		false	0	0.0	0.00	null-time	null-date	""	“”
bool			0/1	0.0/1.0				i	
int		a		b	c			i	
real		d	e		f			i	
amount			g	h				i	
time								i	
date								i	
string		j	j	j	j	j	j		k
bytes								l	

Key

a	0 Æ false, otherwise Æ true.
b	A real number where the integer part is identical to the integer source and the fraction part is zero.
c	An amount where the basic monetary unit is identical to the integer source (5 Æ 5.00A)
d	0.0 Æ false, otherwise Æ true.
e	Truncation, fraction part removed—i.e., 4.5 Æ 4 and –5.3 Æ -5.
f	An amount where the fractional of the real is rounded to two decimals (5.958 Æ 5.96A). g. The integer part of the amount (5.95A Æ 5).
h	As expected (5.95A Æ 5.95).
i	Casting to a string returns the string image of the value—using the current global format settings if necessary.
j	Casting from a string to another type is done using the global format settings. If the conversion fails then a null value is returned. The conversion matches the first necessary characters and silently ignores the rest of the string. For instance, the expression `int("100 boxes")` converts correctly to 100.
k	Casting from a string to bytes is always allowed. The bytes are the (multibyte) characters that the UTF8 string consists of. The length of the bytes is possibly larger than the length of the string.
l	Casting from bytes to a string is only allowed if the bytes comprise a valid UTF8 string. The length of the string is possibly smaller than the length of the bytes.

Examples:

```
var r = real(false);         // r = 0.0
var t = time(null);          // t = nullTime
var a = amount(3.1415);      // a = 3.14A
var d = date("06.05.2005"); // d = May 6th 2005 (user defined format)
```

Type Conversions of Null Values

The conversion to and from null values, null-dates, null-times, and strings can be a bit confusing, so the following goes into more details about these problems.

`null` always converts to a null-date or null-time:

```
date(null) // yields a null-date
time(null) // yields a null-time
```

An empty string or a string that consists of whitespace only always converts to a null-date or null-time:

```
date("")  // yields a null-date
date(" ") // yields a null-date
```

```
time("")  // yields a null-time
time(" ") // yields a null-time
```

A null-date or null-time always converts to an empty string:

```
string(date(null)) // yields ""
string(time(null)) // yields ""
```

Bad input always converts to `null`:

```
date("xxx") // yields null
time("xxx") // yields null
```

`null` always converts to `"null"`:

```
string(null) // yields "null"
```

Type Operators

Name	Description
`typeof(expr)`	Returns a string that contains the type name of the expression.
`def(id)`	Returns true if the variable or property is defined; otherwise, it returns false.
`copyof(a)`	Returns a copy of an object or array. If this operator is used on something that is not an object or array, a run-time error occurs.
`sizeof(v)`	Returns the integer size of a value. For strings the number of UTF8 characters is returned. For bytes the number of bytes is returned.
`ordinal(e)`	Returns ordinal (index) value of the enumerated element `e`. This also works for Maconomy pop-up values where it returns the index of the pop-up.

typeof(expr)

Returns a string that contains the type name of the expression.

Type	Value
`null`	"null"
`bool`	"bool"
`int`	"int"
`real`	"real"
`amount`	"amount"
`date`	"date"
`time`	"time"
`string`	"string"
`bytes`	“bytes”
`object`	"object"
`array`	"array"
`function`	"function"
`enumRef`	"enum"
`enumMember`	user defined

argsof(expr)

Returns an array that describes the arguments to the specified routine. A run-time error occurs if `expr` does not evaluate to a procedure or a function.

Each argument is represented as an object with the following property:

`name`	string	The name of the argument.

Example

```
function f(a, b, c) { ... }
procedure g(x, y) { ... }

dumpvalueln(argsof(f));
dumpvalueln(argsof(g));
  -->
[
  { name: a },
  { name: b },
  { name: c }
]
[
  { name: x },
  { name: y }
]
```

def(var)

Returns true if `var` is defined and false otherwise. The `var` argument must be a reference to a property or an array element, and may not be a general expression. Example:

```
var a = { };
var c = def(a.b);  // c = false
```

sizeof(expr)

Returns the integer size of a value:

Type	Result
`null`	null
`bool`	null
`int`	null
`real`	null
`amount`	null

Type	Result
`date`	null
`time`	null
`string`	Length of the string
`object`	Number of defined properties in the object
`array`	One plus the index of the highest assigned element or zero if empty
`function`	null
`enumRef`	Number of elements in the enumerated set
`enum`	null

Automatic Conversions (Type Promotion)

Type	Promotes automatically to:
`bool`	int,real
`int`	real

Enumerated Values

User-defined enumerated values can be used to give symbolic names to a set of elements. The syntax is:

```
enum id0 { id1[ = expr ], id2[ = expr ], ... };
```

Here the first identifier `id0` is the name of the enumerated type and the remaining identifiers `idi` are the names of the elements in the set. An enumerated type that can be used to identify opened and closed fields in a data row could, for instance, be defined as:

```
enum Field { Opened, Closed };
```

It is also possible to assign explicit values to the elements:

```
enum Field { Opened=10, Closed=99 };
```

The enumerated members can be compared to each other with the usual relational operators. Note that it is the order of appearance in the enum statement that defines the ordering, not the assigned value of the members:

```
enum Field { Opened=99, Closed=10 };


var a = (Opened < Closed);  // True
var b = (Opened > Closed);  // False
```

The enumerated members can be converted to integers. This yields the assigned value. The position in the sequence (the ordinal) can be obtained with the `ordinal` operator:

```
enum Field { Opened=99, Closed=10 };
```

```
var a = int(Opened);      // a = 99
var b = int(Closed);      // b = 10
var c = ordinal(Opened);  // c = 0
var d = ordinal(Closed);  // d = 1
```

It is also possible to convert integers to enumerated values, as well as getting the number of elements in an enumerated set:

```
enum Field { Opened=99, Closed=10 };


var a = Field(0);      // a = 'Opened'
var b = Field(1);      // b = 'Closed'
var c = sizeof(Field); // c = 2
```

Enumerated can also be converted to strings, but you cannot convert a string to an enumerated value:

```
enum Field { Opened=99, Closed=10 };


var a = string(Opened);  // a = "Opened"
var b = string(Closed);  // b = "Closed"
var c = string(Field));  // c = "Field";
```

The type of an enumerated member or definition can be found using the `typeof` operator:

```
enum Field { Opened=99, Closed=10 };


var a = typeof(Opened);  // a = "Field"
var b = typeof(Closed);  // b = "Field"
var c = typeof(Field));  // c = "enum";
```

Variables

All variables are declared with the keyword `var` followed by an identifier and an optional initializer:

```
vardeclStmt ::=
 [ session ] var vardecl-list[,];
vardecl ::=
 varRef [ = expr ]
```

Variables without an initializer have the type and value `null`, unless they are session variables with a previously defined value.

Variables must be declared before they are used, and are not created silently when referred to or assigned to. Variables may not be declared twice.

Referring or assigning to an undefined variable results in a run-time error.

A variable declaration is a valid statement and may occur anywhere in the script.

Local Variables

Variables declared inside functions are local variables. Such variables exist only while the function is active.

Example:

```
function f()
{
 var x = 10, y = 20;
 var z;
}
```

Global Variables

Variables defined outside of a function are global variables. Such variables exist until the program finishes, that is, during one execution of the M-Script interpreter.

Session Variables

Global variables defined with the `session` modifier exist across multiple calls to the M-Script interpreter during the same session. Multiple instances of a session variable can exist, one per session. The initializer is used only when the variable is created; that is, at the first encountered declaration of the session variable in each session. This implies that a session must be created before any session variable declarations are encountered.

Session variables are specific to the packages in which they are declared. Thus a package can declare its own variables without having to fear name clashes with other package's variables. If access is needed to a shared session variable, it must be declared public in a package and accessed through that package.

Session variables are handled via cookies or session-IDs supplied to the URL, and can be used for cross-page information.

Example:

```
// In file A
```

```
session var x = 20;
 // In file B
session var x = "Hello";
```

If file A is executed before B, X has the value 20 in file B. If, on the other hand, file B is executed before A, X has the value “Hello” in file A. Local session variables are not allowed.

Functions and enumerated values cannot be stored in session variables.

Objects

An object is a collection of named pieces of data. These named values are usually referred to as properties or fields of the object. The properties of an object can be accessed by the dot "." operator. For example, if an object named `user` has two properties named `password` and `accesslevel`, you can refer to these properties as follows:

```
user.password
user.accesslevel
```

Properties of objects are, in many ways, just like M-Script variables and can contain any type of data, including other objects, arrays, and functions.

If a function is stored in an object, that function is typically called a method. To invoke the method of an object, use the dot operator to access the function and the "`()`" syntax to call the function. For example, to invoke the function `buildHTML` in an object named `table`, you should use:

```
table.buildHTML();
```

Objects in M-Script have the ability to serve as associative arrays—that is, they can associate arbitrary data values with arbitrary strings (property names). When objects are used in this way, a different syntax for accessing the properties should be used; a string that contains the name of the property is enclosed within square brackets. This allows M-Script programs to build strings at run time and access properties with these, instead of only being able to access hard-coded properties. To access the `password` and `accesslevel` properties in this way you could use the following:

```
user["password"];
user["accesslevel"];
```

Accessing properties which do not exist is illegal and results in a run-time error. To avoid this, you may use the `def` operator to test for existence of the property before referring to it. This operator yields `true` if the property exists and `false` otherwise. The argument to this operator may only be a reference to a property, not any general expression. If you try this operator on the `user` object you get:

```
if (def(user.password)) ...    // Evaluates to true
if (def(user.notdefined)) ...  // Evaluates to false
```

Creating Objects

Objects are created with object literals, where properties are named with an identifier followed by a colon and then the value of the property. To create the user object mentioned previously, you could use the following:

```
var user = { password:"abc", accesslevel:10 };
```

New properties may be added later on with the `new` operator (if you are used to JavaScript, this is one of the places where M-Script and JavaScript differ):

```
new user.name = "John Smith"; // Must be new property
```

You can also create properties or overwrite existing ones with the `set` operator:

```
set user.name = "Alex Madsen"; // Property may exist already
```

New properties may also be added with the `[]` syntax:

```
new user["age"] = 25;
```

Note that the use of `[]` allows you to create and access properties that cannot be accessed with the dot operator:

```
user["100"] = "text"
user.100 = "sorry - number!";  // Parse error! "100" is not
                               // a valid identifier.
```

Iterating through Properties

It is possible to iterate through all defined properties of an object with the `for (x in...)` construction:

```
var user = { password:"abc", accesslevel:10 };
for (var p in user)
{
  print("^1 = ^2\n", p, user[p]);
}
```

When this construction is used, the iterator variable (in this example `p`) is assigned the name of the properties in the object, one by one, and it can then be used to look up the property using the `[]` syntax.

Circular References

Circular object references are not allowed and there are no checks for it in the M-Script interpreter. The behavior of programs with circular references is undefined, but the M-Script interpreter will probably crash. Example:

```
var o = { a:10 };
new o.b = o;       // Error! circular reference
```

Merging Objects

It is possible to overwrite the properties in one object with the properties of another object:

```
var person = { name:"John", age:25 };
var location = { address:"Kings Road 12B", city:"London" };
var user = person + location;
```

If the preceding M-script is executed, the `user` object is:

```
user: { name:"John", age:25, address:"Kings Road 12B",
        city:"London" };
```

The merge operator (`+`) is neither recursive nor type-sensitive. A property in the right operand always overwrites a property in the left operand. Note that the merge operator is not commutative. That is, the order of the operands does matter.

Merging objects can also be done with the `+=` operator

Deleting Properties

Object properties may be removed with the `delete` operator:

```
user = { password:"abc", accesslevel:10 };
delete user.password;
```

Case-Insensitive Objects

From M-Script version 6.0 you can case-insensitive objects; that is, objects that disregard casing when comparing property names. Thus, the properties “abc,” “ABC,” and “aBc” are all considered equal in such an object. Case-insensitive objects are created with the object literal @{...}. Example:

```
#version 15


var o = @{ a:10, b:20 };
print("a = ^1\n", o.a);
print("A = ^1\n", o.A);
print("b = ^1\n", o.b);
print("B = ^1\n", o.B);
```

A case-sensitive and a case-insensitive object can be compared using the equivalence operator `<=>` , but the result will always be `false`.

Merging Objects of Different Casing

The merge operators `+` and `+=` can be applied to objects of different casing, in which case the resulting casing rule is defined by the left operand. Thus, in the expression “a = b + c” the casing rule of a is set to that of b, and in the expression “a += b” the casing rule of a is unaltered.

As a result of this it is possible for two different properties in a case-sensitive object to be collapsed into a single property in a case-insensitive object. Example:

```
#version 15


var o1 = @{ a:10, b:20 };
var o2 =  { a:5, A:15 };
var o3 = o1 + o2;


dumpvalue(o3);
The output is:
@{
  a:15,   (First overwritten by a:5 and then by A:15)
  b:20
}
```

Performance Tips

When working with objects that have a high degree of nesting, you should use references to the innermost objects instead of accessing them with several dots. Example:

Instead of doing this:

```
var o = ... // very nested object
for (var i in o.a.b.c.d)
{
  ... use o.a.b.c.d[i] ...
}
```

You should do this:

```
var o = ...         // very nested object

var d = o.a.b.c.d; // Create reference

for (var i in d)

{

  ... use d[i] ...

}
```

This solution should be faster than the first, because it saves a lot of property lookups. The “copy” of `o.a.b.c.d` does not take much time, because it only copies a reference to the object.

Arrays

An array is, just like an object, a collection of data values. While each data value in an object has a name, each data value in an array has a number, starting from zero.

Arrays are in many ways similar to objects—new elements must be created with the `new` operator, it is possible to detect the existence of an element with the `def` operator, and unused elements may be deleted with the `delete` statement.

Array elements are accessed with the `[]` syntax, just like properties, but with an integer index instead of a string. For example, if `a` is an array with at least three elements in it (0,1,2) these can be accessed as:

```
a[0]
a[1]
a[2]
```

A copy of a subrange of an array, also called an array slice, can be selected by using both a starting and an ending index:

```
a[0..1]  // Subtable starting from 0 and ending on 1
a[2..]   // Subtable from 2 (included) and up to last element
a[..2]   // Subtable from first element and up to 2 (included)
```

An array slice is a “shallow” copy of the subrange. This means—as usual—that all arrays and objects in the slice are copied by reference, and thereby that subsequent changes to these elements in the slice are reflected in the original array.

Creating Arrays

New arrays are created with array literals. An array literal is simply a (possibly empty) comma separated data list enclosed in brackets:

```
[ ]                 // Empty array
[ "John", "Lisa" ]  // Array with two strings
```

Arrays may contain elements of different types and be initialized with expressions that depend on other data elements in the program:

```
[ "John", 06.05.1970, 9.95A ]  // Array with different types
[ x, y, x+2*y ];               // Array with computed elements
```

New elements may be added with the `new` operator:

```
var a = [ "John", "Lisa" ];  // Array with two elements
new a[2] = "Peter";          // Add "Peter" at position '2'
```

Any elements between the last used element in an array and the newly added element are filled with `null` values:

```
var a = [ "John", "Lisa" ];  // Array with two elements
new a[5] = "Mary";           // Add "Mary" at position '5'
// Now a = [ "John", "Lisa", null, null, null, "Mary" ]
```

It is even possible to insert more than one element before existing elements:

```
var a = [ "John", "Mary" ];
new a[1] = "Lisa";
```

```
// Now a = [ "John", "Lisa", "Mary" ]
new a[2..3];  // Insert two elements before element 2
// Now a = [ "John", "Lisa", null, null, "Mary" ]
```

The return value of the `new` operator depends on the use of it; if one array element is created (using `[i]`), the `new` operator returns a reference to that element—that is why it is possible to write `new a[1] = "Lisa"`. If operator `new` is used to insert one or more elements with `[i..j]`, `new` returns a reference to the array itself. In this way it is possible to create expressions like the following:

```
var a = [ "John", "Mary" ];
(new a[1..2])[0] = "Peter";
// now a = [ "Peter", null, null, "Mary" ]
```

Concatenating Arrays

One array may be concatenated with another with the `+` operator:

```
var a = [ "John", "Lisa" ];
var b = [ "Mike", "Mary" ];
var c = a + b;
// now a = [ "John", "Lisa", "Mike", "Mary" ];
```

The `+=` operator may be used as well (with the same side effects as for objects).

Iterating through Arrays

It is possible to iterate through all elements of an array with the `for (x in ...)` construct:

```
var a = [ "John", "Lisa", "Mike", "Mary" ];
for (var p in a)
  print("a[^1] = ^2\n", p, a[p]);
```

In this construct, the iterator variable (in this example `p`) is assigned the values `0`...`sizeof(a)-1` — one for each iteration. These values can then be used to index the elements in the array.

Circular References

Circular array references are not allowed, and there are no checks for it in the M-Script interpreter. The behavior of programs with circular references is undefined, but the M-Script interpreter will probably crash. Example:

```
var a = [];
new a[0] = a;  // Error! circular reference
```

Deleting Elements

Single array elements may be deleted with the `delete` operator:

```
var a = [ "John", "Mary", "Jennie" ];
delete a[1];
// Now a = [ "John", "Jennie" ];
```

The `delete` operator removes only the element pointed to by the index and then shifts all of the subsequent elements one to the left.

A whole subrange of an array may also be deleted:

```
var a = [ "John", "Mary", "Peter", "Jennie" ];
delete a[1..2];
// Now a = [ "John", "Jennie" ]
```

Expressions and Operators

An expression in M-Script is a combination of values and operators, and it can be evaluated to produce a value. The simplest expressions, or atomic expressions, are hard-coded constant values such as Booleans, integers, reals, and strings:

```
true
42
3.1415
"M-Script is fantastic!"
```

The value of a constant is simply the constant itself. One or more constant values may be combined to define object and array literals:

```
{ a:10, b:20, c:30 }
[ "a", "b", "c" ]
```

At last you can construct complex expressions from the literals and operators:

```
10 + 20
a + b*2;
```

The value of an expression is determined by first evaluating its subexpressions and then joining these with the operators.

Literals

The basic operands of an expression are the literals:

```
literal ::=
  intLiteral          // 1234
  | realLiteral       // 12.345
  | stringLiteral     // "abcd" or 'ID' ... ID ('here' text)
  | dateLiteral       // 06.05.2000 or 25.2.2000
  | timeLiteral       // 12:05:42
  | amountLiteral     // 135.76A
  | boolLiteral       // true / false
  | nullLiteral       // null
  | objectLiteral
  | arrayLiteral

objectLiteral ::=     // { x:5, y:7 }
  { objLit-list[,] }

objLit ::=
  id : expr

arrayLiteral ::=     // [ 5, "aaa", {x:3, y:"ab"} ]
  [ exprlist[,] ]
```

String Localization

String localization is the process of translating text strings to a specified locale, that is, language. In M-Script it is possible to mark up strings for static localization or to perform dynamic run-time localization of the marked-up strings.

A string is marked for localization by prefixing it with # and a locale specifier. The specified locale must be a single uppercase letter in the range “A”-”M” and “O”-”Z” (“N” is reserved; see the following):

```
#T "Hello world";  // Localize "Hello world" to locale T
#T message;        // Localize the value of message
#T (expr);         // Localize the result of expr
```

If M-Script has a dictionary associated with the “T” locale (by convention this locale stands for template localization) it performs a lookup in that dictionary, and if a translation is specified, the string is translated. If no translation is specified, or if the type of the variable is not a string, the value remains unchanged.

There is a special form of comment which may occur in strings to be localized:

```
#T "Text{comment}more text{another comment}";
```

These comments can be used to resolve non-objective translations by identifying different interpretations of the same key. The comments are stripped from the string being localized if the locale is known to M-Script, regardless of whether it was translated or not. If the locale is unknown to M-Script, the string always remains unchanged.

As an example, consider the Danish term “gem,” which translated to English can be either “save” or “hide,” depending on the context. The translation engine has no way of knowing which one is correct, so you must tell it. In the dictionary you therefore specify two translations of “gem” distinguished by a comment:

```
gem{save}    save
gem{hide}    hide
```

You can now uniquely specify which translation should be used:

```
#T "gem{save}";// translates to "save"
#T "gem{hide}";// translates to "hide"
```

If no translation of `gem{...}` has been specified in the dictionary, the result would simply be `gem`, that is, the original string with the comment removed.

If you want to specify that a string should *not* be localized, you can simply prefix the string with # or, as mentioned previously, with #N (for “No translation”). The #N convention was introduced in M-Script version 17 to match the Portal development conventions.

Operator Overview

The M-Script operators all have a precedence associated with them. This precedence is used by the M-Script interpreter to figure out how to group the expression when ambiguities occur. For example, multiplication has higher precedence than addition, and so the expression `x+y*z` is evaluated as `x+(y*z)`.

The following is a list of all operators available in M-Script. The list is written with operators of the same precedence inside the same box, and operators with the highest precedence at the top. The “numbers” that the operand types refer to may be either integers or reals.

All assignment operators are right associative whereas all other operators are left associative. The operator associativity defines whether it is grouped from left to right or from right to left:

```
a + b + c + d   // left associative:  (((a + b) + c) + d)
a = b = c = d   // right associative: (a = (b = (c = d)))
```

The full set of operators is:

Operator	Left operand type	Right operand type	Result type	Description
.	object	property	any	property access
[ ]	array	int	any	array index
( )	function	arguments	any	function call (including the `typeof`, `sizeof`, `copyof` and `def` operators)
new		property	property ref.	creates a new property and returns a reference to it
set		property	property ref.	creates a new property or overwrites an existing one and returns a reference to it
++ --		mixed	mixed	increment, decrement
++ --	mixed		mixed	increment, decrement (right associative)
+ -		number	number	unary plus, unary minus
~		int	int	binary negation
!		bool	bool	Boolean negation
@		string	string	databasename conversion
* / %	number	number	number	multiplication, division, remainder
+ -	mixed	mixed	mixed	addition, subtraction
<<	int	int	int	shift-left with zero-extension
>>	int	int	int	shift-right with sign-extension
< <=	mixed	mixed	bool	less-than, less-than or equal
> >=	mixed	mixed	bool	greater-than, greater-than or equal
== !=	any	any	bool	equal, not-equal (by value

Operator	Left operand type	Right operand type	Result type	Description
				for atomic types and strings, and by reference for the rest)
<== >==	any	any	bool	partial equality for compound values, equality for simple values
<=>	any	any	bool	equivalence for compound values, equality for simple values
<= >=	any	any	bool	partial equivalence for compound values
&	int	int	int	bitwise AND
^	int	int	int	bitwise XOR
|	int	int	int	bitwise OR
&&	bool	bool	bool	conjunction (conditional evaluation)
||	bool	bool	bool	disjunction (conditional evaluation)
? :	bool	any, any	any	conditional operator (right associative)
=	variable reference	any	any	assignment (right associative)
*= /= %= += -= <<= >>= &= ^= |=	variable reference	any	any	assignment with operation
,	Any	any	any	multiple evaluation

The next table lists all legal combinations of types and operators. Any combination which cannot be automatically converted to any of the legal combinations results in a run-time error.

Left operand type	Operator	Right operand type	Result type
object	.	identifier	type of identifier

Left operand type	Operator	Right operand type	Result type
object	[ ]	string	type of identifier
array	[ ]	integer	type of element
string	[ ]	interval	string
identifier	( )	arguments	any
int,date,time reference	++ --		type of operand
	++ --	int, date, time reference	type of operand
	+ -	int, real	type of operand
	+ -	null	null
	~	int	int
	~	null	null
	!	bool	bool
	!	null	null
	@	string	string
null	+ - * / %	any	null
any	+ - * / %	null	null
int	* /	int	int
real	* /	real	real
amount	* /	real	amount
real	*	amount	amount
int	%	int	int
real	%	real	real
int	+ -	int	int
real	+ -	real	real
date	+ -	int (days)	date
date	-	date	int (days)

Left operand type	Operator	Right operand type	Result type
time	+ -	int (seconds)	time (modulo 24 hours)
time	-	time	int (seconds)
amount	+ -	amount	amount
string	+	string	string
object	+	object	object
array	+	array	array
any	<< >>	null	null
null	<< >>	any	null
int	<< >>	int	int
null	< <= > >=	any	null
any	< <= > >=	null	null
int, real, amount, date, time, string	< <= > >=	same as left operand	bool
any	== !=	same as left operand	bool
any	== !=	null	bool
null	!= ==	any	bool
any	<=> <== >==	same as left operand	bool
any	& ^ |	null	null
null	& ^ |	any	null
int	& ^ |	int	int
bool	&& ||	bool	bool[a]
bool	? :	any, any	type of right operand
var reference	=	any	type of right operand

Left operand type	Operator	Right operand type	Result type
int reference	`*= /= %= += -= <<= >>= &= ^= |=`	int	int
real reference	`*= /= %= += -=`	real	real
amount reference	`*= /=`	real	amount
amount reference	`+= -=`	amount	amount
date reference	`+= -=`	int	date
time reference	`+= -=`	int	time
string reference	`+=`	string	string
object reference	`+=`	object	object
array reference	`+=`	array	array
array reference	`<<= >>=`	int	array
any	`,`	any	type of right operand

[a] Note that the conditional evaluation is affected by the null value; if any subexpression yields `null`, `null` is returned without evaluation of the rest of the expression.

Arithmetic Operators

- **Addition (+)** — The `+` operator adds numbers, concatenates strings, merges objects, and concatenates arrays.
- **Subtraction ()** — The - operator subtracts numbers, days and times.
- **Multiplication (*)** — The `*` operator multiplies numbers.
- **Division (/)** — The `/` operator divides numbers. Division by zero is not allowed and results in a run-time error.
- **Modulo (%)** — The `%` operator computes the remainder of the integer division of the left and right operand. The modulo operator also applies to reals, in which case the remainder `r` of `x % y` satisfies `x = k*y + r`, where `k` is the largest integer below `x/y`.
- **Unary negation ()** — The unary - operator used as `x` corresponds to `0-x`.
- **Increment (++)** — The increment operator adds one to its operand. It comes in a prefix and postfix version, which means one can write `++x` as well as `x++`. When used as `++x`, it first increments `x` and then returns the value of `x`. If used as `x++`, it stores the value of `x`, increments `x`, and then returns the stored value. Dates are incremented in intervals of one day, and times in seconds.

- **Decrement ()** — The decrement operator works in the same way as the increment operator, except, of course, that it subtracts one from its operand.

Bitwise Operators

- **Bitwise And (&)** — The `&` operator performs a Boolean AND operation on each bit of its integer operands.
- **Bitwise Xor (^)** — The `^` operator performs a Boolean XOR operation on each bit of its integer operands.
- **Bitwise Or (|)** — The `|` operator performs a Boolean OR operation on each bit of its integer operands.
- **Bitwise Not (~)** — The unary `~` operator performs a Boolean NOT operation on each bit of its integer operand.
- **Bitwise Shift Left (<<)** — The `<<` operator performs a bitwise shift left operation on its left operand. The bits are shifted as many times as the right operand evaluates to. Zeros are inserted from the right.
- **Bitwise Shift Right with Sign (>>)** — The `>>` operator performs a bitwise shift right operation on its left operand. The bits are shifted as many times as the right operand evaluates to. The result is sign extended.

Relational Operators

A range of operators is defined for comparing both simple and compound M-Script types. In addition to the usual equality and less/greater than operators, M-Script also offers shallow and recursive comparison of compound types such as objects and arrays.

Simple Comparison Operators

The comparison operators apply to Booleans, integers, reals, strings, amounts, dates, times, and enumerated values. Strings are compared lexically—the other values all have a natural ordering (where null-dates and null-times are the first elements in time, so a null-date is always less than any other date).

- **Equality (==)** — The `==` operator evaluates to `true` if its two operands are equal, `false` otherwise. The definition of what it means to be “equal” depends on the type of the operands.
- **Inequality (!=)** — The `!=` operator returns the opposite of the equality operator, such that `a!=b` is equivalent to `!(a==b)`.
- **Equivalence (<=>)** — The `<=>` operator tests for equivalence of two compound values (as well as simple values). The definition of what it means to be “equivalent” depends on the type of the operands.
- **Less Than (<)** — The `<` operator evaluates to `true` if its first operand is less than its second operand; otherwise it evaluates to `false`.
- **Less Than or Equal (<=)** — The `<=` operator evaluates to `true` if its first operand is less than or equal to its second operand; otherwise it evaluates to `false`.
- **Greater Than (>)** — The `>` operator evaluates to `true` if its first operand is greater than its second operand; otherwise it evaluates to `false`.
- **Greater Than or Equal (>=)** — The `>=` operator evaluates to `true` if its first operand is greater than or equal to its second operand; otherwise it evaluates to `false`.

The Equivalence, Less/Greater Than or Equal operators are also defined for compound values (objects and arrays). See the following for details.

Equality and Equivalence

Booleans, integers, reals, strings, amounts, dates, times, and enumerated values are all compared by value. This means that two expressions are equal if, and only if, they evaluate to the same value.

Objects, arrays, and functions on the other hand, are compared by reference. This means that two expressions are equal if, and only if, they refer to the same object, array or function. Two separate arrays are in this way never equal, even if they contain exactly the same values. The same goes for two variables that contain a reference to an object, array, or a function. Such two variables compare as equal only if they refer to exactly the same object.

The `null` value only compares as equal to other `null` values.

The inequality operator compares in the same way as the equality operator, except that it evaluates to `false` whenever the equality operator evaluates to `true`—and vice versa.

Intuitively “equivalent” means “all elements in the compound values are identical”. The precise formulation of equivalence between two values `a` and `b` is:

- If `a` and `b` are of different types then they are not equivalent.
- If `a` and `b` are simple values then they are equivalent according to the same rules as the equality operator (`==`).
- If `a` and `b` are arrays then they are equivalent if:
  - They both have the same number of elements.
  - Each element `a[i]` is equivalent to `b[i]` for all positions `i` in the arrays.
- If `a` and `b` are objects then they are equivalent if:
  - They both have the same number of properties.
  - All properties in a have an equivalent property with the same name in `b`.

  This makes the equivalence operator quite different from the equality operator in that it compares by value, whereas the equality operator compares by reference. This also means that the equality operator is far more efficient than the equivalence operator and should be preferred whenever possible.

Partial Equivalence

The partial equality operators test for partial equality of two compound values (as well as simple values). Partial equivalence can be tested using either a shallow or a recursive strategy.

- **Shallow Subset (<==)** — The `<==` operator is satisfied if there exists a subset of the right-hand value equivalent with the left-hand value: `a <== b iff $ b′Õ b` such that `a <=> b′` (where `<=>` is the equivalence operator).
- **Shallow Superset (>==)** — The comparison `a>==b` is equivalent to `b<==a`.

  A textual definition of the shallow subset comparison `a<==b` is:
  - If `a` and `b` are of different types the comparison will return `false`.
  - If `a` and `b` are simple values the comparison tests whether `a==b`.
  - If `a` and `b` are arrays the comparison will return `true` if, and only if:

- Array `a` has the same number of elements as `b` or less.
- Each element `a[i]` is equivalent to `b[i]` for all positions `i` in the arrays.

- If `a` and `b` are objects the comparison will return `true` if, and only if:
  - Object `a` has the same number of properties as object `b` or less.
  - All properties in `a` have an equivalent property with the same name in `b`.

Example

```
{ a:5 } <== { a:5, b:10 }  // true
{ a:0 } <== { a:5, b:10 }  // false
{ a:{b:5} } <== { a:{b:5}, b:10 }  // true
{ a:{b:5} } <== { a:{b:5,c:6}, b:10 }// false
```

- **Recursive Subset (<=)** — The `<=` operator is satisfied if all elements in `a` are subsets of the same elements in `b`:

  `a <= b iff " a'Œ a $ b'Œ b` such that `a'<= b'` (where `<=` is the recursive subset operator).

- **Recursive Superset (>=)** — The comparison `a>=b` is equivalent to `b<=a`.

  Partial equivalence is the same as testing recursively for subsets—that is, `a <= b` if all elements in `a` are subsets of the same elements in `b`. A textual definition of the recursive subset comparison `a<=b` is:

  - If `a` and `b` are of different types then they are not partially equivalent.
  - If `a` and `b` are simple values then they are partially equivalent according to the same rules as the equality operator (==). They are not compared using the standard <= operator.
  - If `a` and `b` are arrays then they are partially equivalent if:
    - Array `a` has the same number of elements as `b` or fewer.
    - Each element `a[i]` is a recursive subset of `b[i]` for all positions `i` in the `a` array.
  - If `a` and `b` are objects then they are partially equivalent if:
    - If `a` has the same number of properties as `b` or fewer.
    - All properties in `a` have a partially equivalent property with the same name in `b`.

  Example

```
{ a:5 } <= { a:5, b:10 }  // true
{ a:0 } <= { a:5, b:10 }  // false
{ a:{b:5} } <= { a:{b:5}, b:10 }  // true
{ a:{b:5} } <= { a:{b:5,c:6}, b:10 }  // true
```

Logical Operators

The logical and operator and the logical or operator are both evaluated conditionally. This means they first evaluate their left operand and then decide whether or not they should evaluate the right operand. This means that expressions with side effects, such as `++` and function calls, may or may not be evaluated, depending on the value of the left operand. This can be useful in some cases,

as well as quite confusing in other cases. It is for instance useful first to check for existence of a property and then evaluating that property with the `&&` operator:

```
var o = some object ...
if (def(o.size)  &&  o.size == 5)
  do something ...;
```

A more confusing example is:

```
if ((a == b)  &&  (++c < 10))
  do something ...;
```

In this case, the variable `c` is not always incremented as one might expect. The (evaluated) operands must always evaluate to a Boolean value or `null`. A `null` value is also conditionally evaluated, which means that if the left operand evaluates to `null`, then the operators skips the right operand and returns `null`.

- **Logical And (`&&`)** — The `&&` operator first evaluates its left operand; if this evaluates to `false`, then the `&&` operator returns `false`, if not then it evaluates its right operand and returns this value.
- **Logical Or (`||`)** — The `||` operator first evaluates its left operand; if this evaluates to `true`, then the `||` operator returns `true`, if not then it evaluates its right operand and returns this value
- **Logical Not (`!`)** — The unary `!` operator evaluates its operand and returns the Boolean negation of it.

Assignment Operators

Assignment in M-Script is done by reference for objects, arrays and functions, and by value for the remaining types. This means that if one variable contains a reference to an object and it is assigned to another variable, then that variable refers to exactly the same object, and subsequent modifications through either of the variables can be seen in the other variable:

```
var o1 = { a:10, b:20 };  // o1 = { a:10, b:20 }
var o2 = o1;              // o2 = { a:10, b:20 }
o1.a = 5;                 // both o1.a=5 and o2.a=5
```

If a copy of an object or array (not a function) is wanted, then you should use the `copyof` operator.

All of the arithmetic and bitwise operators may be combined with the assignment operator:

```
var a = 5;
a = a + 5;  // This and the next line are identical
a += 5;
```

The difference between doing `a=a+5` and `a+=5` is that the first version creates a temporary value corresponding to `a+5` and then assigns this value to `a`. The second version takes the variable `a` and adds 5 to it, without creating any temporary value. This may not seem like such a big difference, but it does have a huge performance impact, especially with strings and objects where it can be rather expensive to create the temporary copy. The difference can be illustrated with this example:

```
var x = { a:2, b:3 };  // x refers to { a:2, b:3 }
var y = x;             // y refers to the same object as x
x = x + { c:4 };       // a new object { a:2, b:3, c:4 } is created
                       // and x refers to this
```

```
                        // -- so y is still referring to { a:2, b:3 }
x = y;                  // x now points to the same object as y again
x += { d:5 };           // the object referred to by x is modified --
                        // now both x and y refers to { a:2, b:3, d:5 }
```

Other Operators

Comma Operator (,)

The comma operator first evaluates its left operand, then it evaluates its right operand, whereafter it returns the value of the right operand evaluation. The typical use of this operator is in `for` loops:

```
for (var x=0, y=9 ; x<10 ; ++x, --y)
  print("x,y = ^1,^2\n", x, y);
// Outputs  x,y = 0,9
//          x,y = 1,8
//          ...
//          x,y = 9,0
```

The following example illustrates how the evaluated value of the comma operator can be used:

```
print("(5,10) = ^1\n", (5, 10) );  // Outputs "(5,10) = 10"
```

The parentheses around (5,10) is needed to distinguish 5,10 from being two separate arguments to the `print` function.

Conditional Operator (? :)

The conditional operator corresponds roughly to an `if` statement with a return value. The operator is used as follows:

```
y==0 ? null : x/y
```

The operator first evaluates the expression to the left of the question mark; if it evaluates to `true` then the expression to the left of the colon is evaluated and returned as the result of the conditional operator. If the conditional expression evaluates to `false`, the right expression is evaluated and returned as the result.

The conditional operator is right associative, so the following expression:

```
a == b ? c : d == e ? f : g
```

is identical to

```
a == b ? c : (d == e ? f : g)
```

The conditional evaluation also applies to `null` values, so the expression `true ? x : null` evaluates to `x`, whereas `false ? x : null` evaluates to `null`. The conditional operator is often used as shown below to get an optional property value:

```
var x = (def(o.p) ? o.p : null);
```

This has led to the following shorthand for the same:

```
var x = o.p ?: null;
```

This only works with object properties to the left of the question mark. In this form the construction checks for the existence of the ‘p’ property—if it exists it is used; otherwise, the value to the right of the colon is used (in this case `null`).

New Operator (new)

The `new` operator works on both objects and arrays and can be used to add new properties or elements to these data structures.

Set Operator (set)

The `set` operator works on objects and can be used to add new properties or overwrite existing ones. The `set` operator has some syntactical restrictions in order to allow the use of “set” as a function name in the built-in packages (for instance `format::set`). To use `set` as an operator it must either be the first word on a line or have white space in front of it. It must also always be followed by whitespace.

Delete Operator (delete)

The `delete` operator works on both objects and arrays and can be used to remove properties or elements from these data structures.

Database Name Conversion (@)

The `@` operator converts its argument string to a Maconomy database name. This means lowercasing the string, and truncating it to 20 characters. When `@` is used as part of an identifier, it converts the name of the identifier, not the value referred to by the identifier:

```
   // Identical to "var maa = 5;"
var @maa = 5;
   // Identical to "obj.maa = 2;"
obj.@maa = 2;
   // Identical to "obj["må"] = 3;"
obj[@"må"] = 3;
   // Identical to "name = @(text);" (note the space)
name = @ text;
   // Identical to "name = text";
name = @text;
```

Statements

The actual program structure and control path in M-Script is made up of statements. These are evaluated one by one in the sequence they occur in the input, but contrary to expressions, they do not evaluate to a value. Instead they decide what statements to execute next, based on the evaluation of the embedded expression.

Statements can either be single statements or compound statements, which are again made up of other statements. A compound statement is enclosed in braces:

```
stmt1;    // A single statement stmt2;    // A single statement
{         // The start of a compound statement
       // consisting of two single ones.
   stmt3;
   stmt4;
}
```

A compound statement also defines a variable scope—that is, all variables defined inside a compound statement can only be referred to inside the enclosing braces.

Expression Statement

The simplest kind of statements is expressions:

```
x = y*2;
y++;
```

An expression statement must be terminated with a semicolon.

Variable Declarations

Syntax

```
vardeclStmt ::=
   [ session ] var vardecl-list[,]
vardecl ::=
   id [ = expr ]
```

A variable declaration is a statement that creates one or more variables. These variables can be referred to right after the declaration and until the end of the enclosing scope.

See also Variables for more information.

If – Else

Syntax

```
ifStmt ::=
   if ( expr ) statement1 [ else statement2 ]
```

The if keyword evaluates the expression expr and executes statement1 if it evaluates to true. If the expression evaluates to false and the second expression is present, statement1 is executed. Any other value of the conditional expression, including null, is considered a run-time error.

When if statements are nested, each else branch is associated with the closest if that lacks an else branch, that is, the two following statements are semantically identical:

```
if (e1)
   if (e2)
      x++;
   else
      y++;
if (e1)
{
   if (e2)
      x++;
   else
      y++;
}
```

Switch

Syntax

```
switchStmt ::=
   switch ( expr ) { case-list }
   case ::=
      case expr : statement-list
      | default : statement-list
```

The `switch` statement behaves like a multiway if statement. The first expression is evaluated once and then compared to each of the expressions in the case list (from top to bottom). If it compares equal to any of the expressions, control is transferred to the associated statement sequence which is then executed. Note that if no break statement is found in this statement sequence, control passes on to the next case statement sequence.

If a break statement is found in the statement sequence, control is passed to the end of the switch statement.

If none of the case expressions compares equal to the first expression, control is passed to the default statement, if such exists.

Example

```
enum Field { Open, Closed, Unused, Hidden, Encrypted };
   var x = Closed;
   switch (x)
   {
   case Open:            // If x equals Open
      print("Opened\n"); // Print "Opened" and break out of switch break;
   case Closed:          // If x equals Closed
      print("Closed\n"); // Print "Closed" and continue in next case
   case Unused:          // If x equals Unused (or Closed)
   print("Unused\n"); // Print "Unused" and break out of switch break;
   default:                        // If none of the above fits then
   print("Unused or Hidden\n");  // Print and break out of switch break;
}
```

While – Do

Syntax

```
whileStmt ::=
while ( expr ) statement
```

The while statement first evaluates the conditional expression; if it evaluates to true, the statement body is evaluated, whereafter the process starts over again with evaluation of the expression. This continues until the first time that the expression evaluates to false.

Example

This example uses the variable i to count from 0 to 9:

```
var i = 0;

while (i < 10)
{
   print("i = ^1\n", i);
   ++i;
}
```

Do – While

Syntax

```
doStmt ::=
   do statement while ( expr ) ;
```

The do statement differs only from the while statement in that it first executes its body and then evaluates its conditional expression.

For

The for loop comes in two versions; the first version is based on the loop construct found in the C programming language, whereas the second is more JavaScript style and can be used to iterate through the elements of an object or array.

C-Style

Syntax

```
forStmt ::=
   for ( expr1 ; expr2 ; expr3 ) statement
```

The for loop consists of a statement body and three different expressions:

1. Initializer (expr1) — This expression is evaluated initially and only once. It should be an expression that initializes variables in the body. This initializer may include variable declarations—variables declared here are only visible inside the scope of the for loop.
2. Condition (expr2) — This expression is evaluated every time that a new iteration is started. If it evaluates to true, the body is executed; if it evaluates to false, the for loop is skipped and control is passed to the first statement after the for loop's body. Any other value of the conditional expression, including null, results in a run-time error.
3. Increment (expr3) — This expression is evaluated after each pass through the body and is used to update the loop variables. It could, for instance, increment a loop variable: ++i.

Example

This example uses the variable i to count from 0 to 9:

```
for (var i=0 ; i<10 ; ++i)
   print("i = ^1\n", i);
```

JavaScript-Style

Syntax

```
forStmt ::=
   for ( id in expr ) statement
```

This for loop expects the expression expr to evaluate to an array, an object, or a dynamic package reference—any other value results in a run-time error. If it evaluates to an object or package reference, the for loop iterates over each of the properties in the object or public elements in the package, assigning the property or element name to the variable id for every execution of the body. If the expression, on the other hand, evaluates to an array, the for loop iterates over each of the elements in the array, assigning the index value to the variable id for every element in the array.

Examples

```
var p = loadpackage(...);
for (var i in p)
   print(p[^1] = ^2\n, i, typeof(p[i]));

var o = { a:5, b:10, c:15 };
for (var i in o)
   print("o[^1] = ^2\n", i, o[i]);

var a = [ "One", "Two", "Three" ];
for (var i in a)
   print("a[^1] = ^2\n", i, a[i]);
```

Break — Continue

Syntax

```
breakStmt ::=
   break ;

contStmt ::=
   continue ;
```

The break statement is used to “break out of” a program loop, and may therefore only be used inside switch, while, do, and for statements. When a break statement is executed, the program jumps to the first statement following the loop or switch.

The continue statement does more or less the same—it stops execution of the loop, skips the rest of the loop statements, but then it repeats the loop as if all of the loop statements had been executed as usual. The continue statement may be used in while, do, and for statements.

Examples

This example prints the values 0,2,3 (but not 1):

```
for (var i=0 ; i<4 ; ++i)
{
   if (i == 1)
      continue;
   print("i = ^1\n", i);
}
```

This example prints the values 0,1 (but not 2,3):

```
for (var i=0 ; i<4 ; ++i)
{
   if (i == 2)
      break;
   print("i = ^1\n", i);
}
```

Return

Syntax

```
returnStmt ::=
   return [ expr ]
```

A return statement is used stop execution of some subroutine (or the main program) and optionally return a value to the calling program.

A return statement outside a subroutine terminates the M-Script and passes the returned integer value to the operating system.

The Empty Statement

The empty statement is simply a single semicolon. Executing the empty statement obviously has no effect and performs no action. This can be useful in loop constructs such as the following:

```
// Initialize an array a.
for (var i=0 ; i<10 ; a[i++] = 0)  ;
```

The Throw Statement

Syntax

```
stmtThrow ::=
   throw [ type-qualifier ( [ expr ] ) ] ;

type-qualifier ::=
   id [ :: id ]*
```

The throw statement is used for throwing an exception. The optional expression is evaluated into the value being thrown. If no expression is given, an already caught exception is re-thrown.

The Try – Catch – Finally Statement

Syntax

```
stmtTryCatchFinally ::=
    try statement
    [ catch [ type-qualifier [ , type-qualifier ]* ]
        [ id ] ) statementi=1...n
    ]+
    [ finally statementn+1 ]
| try statement1 finally statement2

type-qualifier ::=
```

See The Throw Statement.

The try-catch-finally statement is used for catching and handling exceptions. An exception thrown from within the try-handler try statement will cause execution to jump to the first catch-handler catch type ([id]) statement where the type matches the type of the exception. Here the exception is assigned to the optional identifier id. A finally-handler can either be trailing the catch-handlers or replace them. The statement within the finally-handler finally statement is evaluated when the try-catch handler exits.

Subroutines

Subroutines in M-Script can be functions, which must return a value, and procedures, which may not return a value. Differentiating between these two types of subroutines allows the M-Script programmer to explicitly state whether or not a subroutine should return a value.

Defining and Invoking Subroutines

A subroutine is defined with the function or procedure statement followed by the name of the subroutine, a list of comma-separated parameter names, and the subroutine body.

Syntax

```
subroutine ::=
    ( function | procedure ) id ( parameters )
       {
          statementSequence
       }

    parameters ::=
       id-list[,] [ ... ]
```

Examples

```
// Print "Hello" followed by the name of a friend. procedure hello(friend)
{
    print("Hello ^1\n", friend);
}

// Return the minimum value of a and b. function min(a, b)
{
    if (a < b)
       return a;
    else
       return b;
}
```

Subroutines may be invoked with the `()` operator. The parentheses should appear right after the name of the subroutine and may include a comma-separated list of parameter values. The preceding subroutines may, for instance, be invoked like the following:

```
hello("John");
hello("Jennifer");

print("Min 5,3 = ^1\n", min(5,3) );
print("Min 1,3 = ^1\n", min(1,3) );
```

When a subroutine is invoked, each of the expressions in the parameter list is evaluated and the resulting values are used as arguments to the functions.

The name of a subroutine is known throughout the whole M-Script and not only after its definition.

Subroutines with a Variable Number of Parameters

Subroutines that accept a variable number of parameters can be defined using ellipses instead of a fixed number of parameters. When such a subroutine is called, the arguments that are passed can be accessed via the arguments array. This array is always defined in a subroutine and can be used as any other array. The following procedure illustrates this by printing out all of its arguments:

```
procedure f(...)
{
   for (var a in arguments)
      print("^1 = ^2\n", a, arguments[a]);
}
```

The caller of a subroutine with a variable number of arguments must supply at least the number of parameters listed before the ellipsis. In the preceding example this means zero arguments are required. In the next example, at least two arguments must be passed:

```
procedure f(a, b, ...)
{
   for (var a in arguments)
      print("^1 = ^2\n", a, arguments[a]);
}
```

Value Passing

The rules for passing values to a subroutine are the same as for comparing values; objects and arrays are passed by reference, whereas the remaining types are passed by value. This means that operations performed on objects and arrays, in subroutines, are reflected in the calling program. The following example illustrates this:

```
procedure modifyObject(obj)
{
   new obj.c = 30;
}

var o = { a:10, b:20 };

dumpvalue(o); print("\n");  // o = { a:10, b:20 }
modifyObject(o);
dumpvalue(o); print("\n");  // o = { a:10, b:20, c:30 }
```

This cannot be done with the other types.

Invoking Subroutines with Arrays as Arguments

You can invoke a subroutine with an array that contains the parameters for the subroutine. The expansion is by position, so the first array element is moved into the first parameter, and so on. To do so, the function must be invoked with square brackets instead of parentheses. The following example illustrates this:

```
procedure f(a,b,c)
{
   print("A: ^1, B: ^2, C: ^3\n", a,b,c);
}
```

```
var args = [10,20,30];

f[args]; // prints "A: 10, B: 20, C: 30"
```

Invoking Subroutines with Objects as Arguments

You can also invoke a subroutine with an object that contains the parameters for the subroutine. The expansion is done by name, so the property value named “a” will be copied into the “a” argument and so on. To do so, the function must be invoked with square brackets instead of parentheses. The following example illustrates this:

```
procedure f(a,b,c)
{
   print("A: ^1, B: ^2, C: ^3\n", a,b,c);
}

var args = {b:10, c:20, a:30};

f[args]; // Prints "A: 30, B: 10, C: 20"
```

Recursive Subroutines

It is perfectly legal to make subroutines that call themselves. The following function calculates factorials (n!) recursively:

```
function fact(i)
{
   if (i <= 0)
      return 1;
   return i * fact(i-1);
}

print("4! = ^1\n", fact(4));
```

Subroutine Literals

A subroutine literal is an unnamed subroutine that can be assigned to a variable, passed to another subroutine or invoked as a normal subroutine. Subroutine literals are created in the same way as subroutines, except that no subroutine name is used. The following two ways of declaring a subroutine are almost identical, except that the function f is known in the whole M-Script, whereas the function g is first known after the var g = ... statement:

```
f(10);   // Legal
g(10);   // Illegal – 'g' is unknown

function f(a)
{
   return a*a;
}

var g = function (a) { return a*a; };

f(10);   // Legal
```

```
g(10);   // Legal – now 'g' is known
```

Subroutines, as well as subroutine literals, can be assigned to other variables:

```
var ff = f;  // 'ff' now refers to 'f'
var gg = g;  // 'gg' now refers to 'g'

ff(10);      // Invoke 'f' through 'ff'
gg(10);      // Invoke 'g' through 'gg'
```

Subroutine literals that have been assigned to a property are also referred to as object methods.

Object Methods

A subroutine that is assigned to an object property and called through that property, may refer to the parent object through a variable called this. This variable is declared by the M-Script interpreter and can be used freely as any other variable:

```
procedure dump()
{
   print("My 'x' = ^1\n", this.x);
}

var o = { x:10, dump:dump };  // Create object with
                 // method 'dump'
 o.dump();                     // Outputs "My 'x' = 10"

o.x = 20;
o.dump();                     // Outputs "My 'x' = 20"
```

Packages

M-Script supports a package system for managing program libraries. This system enables the library programmer to supply a controlled interface to the contents of an M-Script package. A simple but powerful mechanism allows client programs access to the library packages through their specified interfaces.

Specifying Package Locations

M-Script packages may contain functions and variables, as well as other packages. The contents of a package are accessed by specifying the package name and element name separated by double-colons. This is called a package qualifier, and its BNF grammar is:

```
package-qualifier ::=
       package-spec::package-qualifier ;
    | package-spec::id ;
package-spec ::=
    . | .. | id
```

Each package name must be a valid M-Script identifier. Packages with names that contain spaces, dashes, or similar will be rejected by the M-Script interpreter.

With a statement A::B you access the element B in package A. If B is itself a package, its elements can then be accessed with the notation A::B::C and so on.

When the element being accessed is a function, procedure, or variable, it behaves just like a local declaration. Thus a function f in package A::B can be invoked by A::B::f(), and if the return value is to be saved to a variable v, the syntax would be v=A::B::f().

Package specifications can also be made relative to the current package's or program's location. A single dot represents the current location of the package or program, while a double dot moves you one step up in the hierarchy. It is therefore possible for package B in the preceding examples to access a function g in the package A through the notation ..::A::g(), and if A contains another package B2, it can be accessed from B with either ..::A::B2 or .::B2. In the same way, a main program can access the local packages A and B in its own directory using .::A or .::B.

Accessing Packages

Before members of a package can be accessed, the package must be imported into the program. The BNF grammar for an import statement is:

```
uses package-version-qualifier [ as id ] ;
package-version-qualifier ::=
       package-spec::package-version-qualifier ;
    | package-spec::id ( version ) ;
package-spec ::=
    . | .. | id

version ::=
       MajorVersion
    | MajorVersion.MinorVersion
    | MajorVersion.MinorVersion.RevisionCode
```

The optional part of the syntax (as id) denotes an alias that can be used as a shorthand to the specified package. If the package A::B is included with the import statement uses A::B as b it becomes possible to write b::f() instead of A::B::f() when invoking the function f in B.

In the M-Script file, import statements must appear before all expressions and declarations, but after the #version tag.

Example

```
#version 15
uses stdlib::sendmail(1); // use version 1.0.0
uses stdlib::ftp(1.1);    // use version 1.1.0
uses stdlib::http(1.1.5); // use version 1.1.5
```

Loading Packages at Run Time

As of M-Script 6.0 you can dynamically load a package into a running M- Script program. A package loaded this way is treated in much the same way as an object value, and its public members are accessible through the usual dot notation or property lookup mechanisms used with objects.

Packages are loaded with the built-in function loadpackage. This function takes as an argument a string that contains a package qualifier specification as described previously. If the qualifier is syntactically correct, and the corresponding package can be found, it is loaded, and a reference to it is returned. Otherwise, a run-time error is thrown. The possible exception types are listed in the following table.

Exception Type	Description
error::load	The package could not be loaded, for example because the file does not exist, or access is denied.
error::parse	There is a syntax or parse error in the loaded package.
*	Whatever exception might be thrown during package initialization, for example, error::runtime or a user-defined exception type.

Example

```
#version 15
var A = loadpackage(A(1));
var B = loadpackage(A::B(1));

A.g();    // invokes A::g()
B.f();    // invokes A::B::f()
A[g]();   // equivalent to A.g()
```

Operations on members of a package reference follow the rules for packages, not those for objects. Unlike real objects, it is therefore generally not possible to add or delete the members of a dynamically loaded package.

When a package is loaded dynamically, its top-level instructions are executed just as if it had been imported statically. Dynamically loaded packages exist throughout the remainder of the program, and it is only possible to load each package once. Loading a package that has already been loaded, either statically or dynamically, will simply return a reference to the existing instance of the

package. You can mix the use of dynamic and static loading, and dynamically loaded packages can load other packages as they please.

Dynamic packages offer a way for the M-Script programmer to supply more dynamic behavior in a program by making it possible to delay the decision of which packages to load until run time. Dynamic package loading should, however, not be considered a replacement for the normal package import; all type checks are done at run time, and access to dynamically loaded packages infers a run-time overhead comparable to that of object access.

Package Version Management

Standard libraries are often long-lived and may undergo several major revisions as well as lots of minor changes and bug fixes. This often leads to version conflicts when a newer program attempts to use not-yet-implemented features in an older version of a library, or when an old program suddenly stops working when one of the libraries it uses is updated.

As a means to preempt such problems, M-Script offers a system for marking packages with a version code. The version code enables the developer to keep track of all revisions made to the package, both minor changes and those breaking backward compatibility. The versioning scheme deployed for packages consists of three numbers separated by dots:

```
VersionCode ::= MajorVersion.MinorVersion.RevisionCode
MajorVersion : int
MinorVersion : int
RevisionCode : int
```

MajorVersion is the main version number of the entire package. This number should only change when modifications are made that break backward compatibility with previous major versions of the package.

MinorVersion is used to keep track of additions made to the package that extend its functionality but do not break backward compatibility.

RevisionCode indicates minor changes to a package, for example bug fixes.

This versioning scheme is made available by M-Script, but it is the responsibility of the package developer to update the version information in each file to reflect changes made in that package.

The version of a package is specified with a package header statement. M-Script uses this version information to verify that:

- The major version number of a package is equal to the number specified in the uses statement, and
- The minor version number of a package is equal to or greater than the specified number, and
- If the minor numbers are equal, the revision code of a package is equal to or greater than the specified number.

If only a partial version code is specified in the uses statement, all omitted version numbers are assumed to have a value of zero.

Example

```
uses somePackage(2.1)
```

means “I want a version of somePackage where the major version number is two, and the minor version number is at least one. Any revision will do.” According to the previously described rule, versions of somePackage with version codes 2.1.0 and 2.2.1 will both satisfy this condition, while 1.0.0, 2.0.1, and 3.1.0 will all fail.

Scopes and Initialization

A package import is valid only within the file in which it takes place. It is therefore both legal and necessary to import a package from every file that uses it. It is illegal to import the same package twice within a file, or to import a package from within itself.

M-Script supports two ways of loading packages: *static* and *lazy*.

With static package load, all packages are loaded before execution starts. This has the advantage that the packages are syntax-checked and resolved before-hand, providing some degree of insurance of the overall consistency of the application. The downside to this approach is that it incurs a run-time overhead proportional to the size of the application, regardless of how much or little code is actually evaluated in a given execution.

With lazy package load, package imports are delayed until a statement actually tries to access the package during execution. At that point the execution is suspended, the package is loaded and resolved, and any initialization code within it is executed. Then the normal execution is resumed, and in most cases the calling statement never notices that the package it accessed was not loaded already. The advantage to this load strategy is that it can significantly reduce the overall execution time of small code traces in a large application.

By default M-Script uses the static load strategy. The configuration file option LazyPackageLoad can be used to switch to the lazy strategy.

Static Package Load

Every imported package is initialized at some point before the main program executes. During this initialization, any top-level statements that are present in the package are executed. The purpose of such instructions is typically to perform initialization of package variables, and so on. Before a package has been initialized, all attempts to access its local variables fail with a run-time error. It is, however, possible to invoke functions in uninitialized packages, as long as these functions do not use uninitialized package variables.

The sequence in which packages are imported and therefore executed is decided by their dependencies. The M-Script interpreter attempts to organize the packages in such a way that all packages have been initialized before they are used. Unfortunately, such an ordering is impossible in the case where two or more packages are mutually dependent, and in such cases the risk of accessing uninitialized variables is considerable. Dependency on a particular order of evaluation between packages should therefore be avoided.

Lazy Package Load

When packages are loaded lazily, any given package is initialized the first time that a public member in the package is accessed. At this point the package is loaded, and any top-level instructions within it are executed before normal operations continue. This form of delayed execution might violate implicit assumptions about what happens during execution, and could thus break an application that runs correctly with static package load. Potentially dangerous application designs include packages that expect to be executed before the main script, and all instances of packages initialization code with side effects. Properly designed M-Script applications should avoid such situations at all costs.

Package File Structure

Packages are represented on disk as a hierarchy of files and folders. A default root folder and a set of named root folders can be specified with the field packageRoot in the M-Script initialization file, and all package files must be placed below one of these folders. If a package contains sub-packages, there will also be a folder with the same name as the package, and the sub-packages are then placed in that folder.

Packages can exist in several major versions, but there can be only one version of the package for each major version number. To identify the different versions of a package, M-Script requires that the major version number of a package is reflected in the name of the corresponding M-Script file. The syntax rule for package files with version information is:

```
PackageWithVersionNo ::= PackageName.MajorVersion.ms
```

Thus, if package A has a major version of one, it must contain the version statement of the form package A(1,...) and be placed in the file A.1.ms. If the package at a later point is changed to a major version number of two, the filename must change accordingly to A.2.ms.

Example

An M-Script package A.1.ms placed in the default package root folder can be accessed with the package path A(1), while another package B.2.ms placed in the named package root folder <root2> can be accessed with the package path

```
<root2>::B(2)
```

The M-Script file that corresponds to the package C::D, where C has major version 1 and D has major version 2, can be found as <packageRoot>/C/D.2.ms, while the package C itself is located in <packageRoot>/C.1.ms.

Writing Packages

M-Script packages must contain a package header after the version tag:

```
PackageHd ::= package name (VersionCode) ;
```

The name and major version specified must be in agreement with the file name of the package to be valid.

Within packages, all declarations are private by default. Therefore a file that was not originally intended to be a package will appear empty in the package system. To make a declaration visible from the outside of a package, it must be prefixed by the keyword public.

Example

```
var secret;             // this is a private variable
public var not_secret   // this is a public variable
function f() {}         // this is a private function
public function g() {} // this is a public function
```

Standard Packages and Site Packages

In addition to the built-in standard library, M-Script is shipped with a collection of standard packages. These standard (formerly also known as “official”) packages are an attempt to resolve some issues experienced in the development community.

The standard packages reside at a predefined location outside the normal package root, and the top-level qualifier mscript:: has been reserved for this location. Please observe that as a consequence of this it is not possible to access a top-level package or package folder named mscript.

The packages are distributed in clear text, but with a checksum to protect them from tampering. M-Script refuses to execute a package beneath the mscript:: root if it does not have a valid checksum.

A second folder is defined for packages that are not standard, but should also be globally accessible from all M-Script installations on the server. These packages can be placed beneath

the site:: root, which is also reserved. There are no requirements to control the authenticity of packages beneath this root.

For more information, please see M-Script Standard Packages.

Exception Handling

As of version 6.0, M-Script uses exceptions to provide error-handling capabilities for its programs. An exception is an event that occurs during the execution of a program that disrupts the normal flow of execution.

About Exceptions

You will probably encounter exceptions after programming M-Script for only a short time. Your first encounter with M-Script exceptions might have been in the form of an error message from the compiler like the following:

```
Runtime error in InputFile.ms(10): unknown property 'value'
```

In M-Script 6.0, such messages look slightly different, but the meaning is the same:

```
Uncaught exception error::runtime in InputFile.ms(10): unknown property value
```

The preceding message indicates that the contents of an object value did not contain the expected property. As the M-Script interpreter cannot continue normally after having encountered such an error, it throws an exception that contains information about where the error took place and what caused it.

In earlier versions of M-Script, such errors would cause execution to be completely abandoned. With M-Script 6.0 it became possible to catch these exceptions from within the program and recover from them in a structured manner.

The Exception Type

In M-Script all exceptions share a common format. Each exception carries with it information about its type and from where it was thrown. In addition, it can carry a user-specified M-Script value of arbitrary complexity. To illustrate this, consider the internal format of the previous run-time error:

```
{
   type:[
      "error",
      runtime
   ],
   value:{
      message:unknown property 'value'
   },
   filename:InputFile.ms,
   linenumber:10
}
```

Notice how the exception object contains all the information necessary to create the corresponding error message. The type property is an array of type string, where each string represents a component in the exception's type specification. The preceding value corresponds to the exception type error::runtime, which indicates that the error was thrown by the run-time logic in the interpreter. The value property in run-time exceptions like the preceding one is an object that contains a string with a description of the error.

Exceptions can be considered read-only objects. It is not possible to add, delete, or change the properties within an exception. Nor is it possible to create an exception value in any other way than by throwing an exception.

Exceptions are classified according to a hierarchy of type-identifiers. When throwing and catching exceptions, a syntax similar to that used for specifying packages and package elements is used. For example, run-time errors in the M-Script interpreter run-time logic belong to the class error::runtime, while all errors from the Maconomy API are identified by the class error::stdlib::maconomy.

Within the exception object, these classifications are represented as a list of the type components. The preceding exception classes thus become [error,runtime] and [error,stdlib,maconomy] in the exception object.

Exception classifications are ordered in a hierarchy. A given exception type can match all exception types with that exception type as a prefix. For example, the exception type error matches error::maconomy because “error“ is a prefix of “error::maconomy.” This means that any exception handler that catches exceptions of type error will also catch exceptions of type error::maconomy.

Throwing Exceptions

Exceptions are thrown by the interpreter whenever it encounters an error. In addition to this, the script itself can throw exceptions. This can be used to express that something was not as expected. When throwing an exception, the script is actually throwing an M-Script value and an exception type. The value can be an error code or message, or any compound M-Script type such an object or array. The exception type is a sequence of type specifications as described previously. Such a value val is thrown as an exception of type ty with a simple throw statement:

```
throw ty(val);
```

When throwing an exception, the value and type properties will be assigned the value and type being thrown, while the filename and line number is assigned automatically by the interpreter.

Example

If an object { code: -1, message: I’m hurt } is thrown as an exception of type “ouch” at line 22 in the file InputFile.ms, the resulting exception will have the value:

```
{
   type:[
      ouch
   ],
   value:{
      code:-1,
      message:I’m hurt
   },
   filename:InputFile.ms,
   linenumber:22
}
```

In addition to this, it is possible to re-throw a caught exception. This is done simply by using:

```
throw;
```

This will cause the caught exception to be re-thrown without modifications. An exception that is caught and re-thrown will therefore maintain the filename and line number information from the place where it was originally thrown. The following shows the grammar for the throw statement:

```
stmtThrow ::=
   throw [ type-qualifier ( [ expr ] ) ] ;
```

```
type-qualifier ::=
   id [ :: id ]*
```

Catching Exceptions

Statements from which exceptions should be caught are placed in an exception handler:

```
try statement
   [ catch type-qualifier ( [ id ] ) statementi=1...n ]+
```

An exception thrown from within the try-handler try statement will cause execution to jump to the first catch-handler catch type ([id]) statement where the type matches the type of the exception. Here the exception is assigned to the optional identifier id. Both the try- and catch-handlers can be a single statement, or a block with several statements. Additionally, a catch-handler can contain several comma-separated type qualifiers. In this case, the handler will catch the exception if any of the specified types match the type of the exception.

A finally-handler can either be trailing the catch-handlers or replace them. The statement within the finally-handler finally statementn+1 is evaluated when the try-catch-handler exits, regardless of whether it happens because of an uncaught exception or, for instance, because of a return, break, or continue statement within the try or catch statement. The following shows the complete grammar for an exception handler:

```
stmtTryCatchFinally ::=
   try statement
   [ catch [ type-qualifier [ , type-qualifier ]* ]
      ( [ id ] ) statementi=1...n
   ]+
   [ finally statementn+1 ]
   | try statement1 finally statement2

type-qualifier ::=
   see section 12.2
```

Exceptions in the Standard Libraries

All of the M-Script standard libraries have been made exception-aware in the sense that error conditions that previously have resulted in return values will now throw exceptions on errors.

Exceptions from the standard libraries have types of the form error::stdlib::<package-name>. They contain exactly one property named “message” in the “value” object of the exception. The “message” property is a string that contains a human readable error message.

The old behavior where failure is indicated by, for example, a status object in the return value has not been completely abandoned, however. Existing code will be able to run unmodified against the new interface, while new code will be able to take advantage of exception handling.

When the Maconomy API is called from a script marked with #version 6 or higher, it will automatically switch into exception-aware mode. If you want to mark a script with the latest version without enabling exceptions in the standard libraries, the special flag #set stdlibStyle 5 can be specified after the #version tag in the files where the old behavior should be preserved. Exception behavior can only be regulated on a per-script basis.

When using the #include facility, the included script “inherits” its exception mode from the including script. Thus, if a version 5 script is included from a version 6 script, exceptions will be enabled in both scripts.

Exceptions in the Maconomy API

All calls to the Maconomy API that previously returned status objects now detect erroneous return values and throw exceptions instead. The type of all exceptions from the Maconomy API share the exception type root error::stdlib::maconomy. Several sub-types of exceptions can be caught from the API. The following briefly presents these types. A more thorough description can be found in the Maconomy API Reference.

- ...::maconomy is thrown on general errors such as an invalid number or type of parameters. In addition, return values that contain just an error message on failure (besides the ok-field and null-values) will also be thrown as this exception. The thrown value is a message object.
- ...::maconomy::login is thrown when login fails. In addition to an error message it contains the errorCode field from the loginReturn object.
- ...::maconomy::dialog::<kind> is thrown where the dialogReturn object is normally returned (see M-Script Maconomy API Reference for details). In addition to an error message, these exceptions contain an object status with the properties messages, error, files, and prints from the returned dialogStatus object.

Standard Library

Encoding

As of M-Script 18.6 (2.1 SP1) you can change the input/output encoding used by most of the standard library functions. The encoding can be changed for a single script (corresponding to a single M-Script source file) and the default encoding for all scripts can also be changed.

The encoding used in a script is derived from the following prioritized list:

1. If the function `setencoding()` has been called in a script, that value is used.
2. Otherwise, if the environment variable DefaultEncoding is set in the script execution environment this value is used.
3. Otherwise, if the function `setdefaultencoding()` has been called this value is used.
4. Otherwise, if the setting DefaultEncoding is set in the M-Script initialization file this value is used.
5. Otherwise, the global default value is used.

By default the input/output encoding used by all scripts is ISO-8859-1 (commonly referred to as Latin-1), but more than 200 encodings are supported, as well as many more aliases for them.

All input/output streams in M-Script are assigned the encoding that is currently used by the script at the time when the stream is created. After this the stream keeps this encoding even if the script encoding changes. It is also possible to assign a different encoding to an existing input/output stream, which is useful when operating on streams with different encodings within a single script.

The following functions create input/output streams with an assigned encoding:

```
file::open
net::connect
net::openurl
io::stdout
io::stderr
io::ostring
io::istring
io::log
```

Most of the input/output functions available in the standard library automatically convert all characters read from an input stream to UTF-8, and convert from UTF-8 to the stream encoding when writing to an output stream. These functions are:

```
print
println
sprint
dumpvalue
dumpvalueln
readvalue
file::print
file::println
file::getline
file::import
file::export
file::dumpvalue
```

`file::dumpvalueln`

`file::readvalue`

`io::getstring` (since the introduction of io::getbytes in M-Script 18.8)

The following stream-manipulating functions do not perform character conversion and can be used to perform rudimentary manipulation of binary input/output streams:

`io::write`

`io::getbytes` (introduced in M-Script 18.8)

The following sections describe the new functions that were introduced in M-Script 18.6 to get and set encodings.

getencoding()

Return Value

The current encoding in the calling script. If an encoding has not been set specifically for this script then the default encoding is returned.

Context

All.

Examples

```
getencoding();
```

might yield the string:

```
"ISO-8859-1"
```

setencoding(encoding)

Set the encoding used in the current script. Passing an empty string to this function will restore the default encoding.

Return Value

The encoding previously set for this script. If an encoding was not set specifically for this script then the empty string is returned.

Context

All.

Examples

```
var old_enc = setencoding("UTF-8");
println("Old encoding: \"^1\", new encoding: \"^2\"",
        old_enc, getencoding());
```

might print the string:

```
Old encoding: "", new encoding: "UTF-8"
```

getdefaultencoding()

Return Value

The default encoding currently used in all scripts that have not specifically set an encoding.

Context

All.

Examples

```
getdefaultencoding();
```

might yield the string:

```
"ISO-8859-1"
```

setdefaultencoding(encoding)

Set the default encoding used in all scripts that have not specifically set an encoding.

Return Value

The previous default encoding for all scripts

Context

All.

Examples

```
setdefaultencoding("UTF-8");
getencoding();
```

will yield the string:

```
"UTF-8"
```

unless an encoding for the current script has also been specified, as illustrated in the following:

```
setencoding("US-ASCII");
setdefaultencoding("UTF-8");
getencoding();
```

will yield the string:

```
"US-ASCII"
```

See Also

`file::getencoding(stream)`

`file::setencoding(stream,encoding)`

Substitution String

As of M-Script 18.6 (2.1 SP1) you can change the string that is output instead of characters that have no representation in the chosen output encoding. The default is `<SUB>`, that is `CTRL-Z (0x1A)`. In some contexts this character causes truncation of the output. The function

`setsubstitutionstring()` defines the string to be used and returns the previous value. The function `getsubstitutionstring()` reports what the current value of the substitution string is.

The substitution string used in a script is derived from the following prioritized list:

1. If the function `file::setsubstitutionstring(f)` has been called in a script, that value is used for the output stream `f`.
2. Otherwise, if the `setsubstitutionstring()` has been called in a script, this value is used.
3. Otherwise, the global default value is used.

The following sections describe the functions that were introduced in M-Script 18.6 to get and set encodings.

getsubstitutionstring()

Return Value

The current substitution string in the calling script. If a substitution string has not been set specifically for this script then the default value is returned.

Context

All.

Examples

```
getsubstitutionstring();
```

might yield the string:

`"<X>"` (note that the string may not be printable, the default value f.i. is not)

setsubstitutionstring(substitutionstring)

Set the substitution string used in the current script. Passing an empty string to this function restores the default substitution string.

Return Value

The substitution string previously set for this script. If a substitution string has not been set specifically for this script the default value is returned.

Context

All.

Examples

```
var old_sub = setsubstitutionstring("_");
println("Old substitution string: \"^1\", new substitution string: \"^2\"",
         old_sub, getsubstitutionstring ());
```

might print the string:

```
Old substitution string: "", new substitution string: "_"
```

See Also

```
file::getsubstitutionstring (stream)
file::setsubstitutionstring (stream, substitutionstring)
```

Printing

All output from the printing routines is stored in a temporary file called the standard output file. This file cannot be accessed in any way other than being cleared by a call to the function discardoutput. Using a temporary file like this makes it possible to generate HTTP headers where the expected output size is stated. In addition to this it makes it possible for the M-Script programmer to make more intelligent error handling routines that may clear all output generated before the error occurred.

print(format, ...)

The print() procedure prints the string format to the standard output file. If format contains any ^1, ^2, ... character combinations—and one or more arguments have been supplied after the format string—then these ^n references are substituted with the n-th argument following the format string. If more than nine arguments are passed to the print routine, then these can be accessed as ^(nn).

This example prints the name of the M-Script host machine:

```
print("Host: ^1\n", getenv("SERVER_NAME"));
```

You can also access properties of objects within the format string. If the first argument after the format string is an object containing the property a, then this property can be printed with:

```
print("^(1.a)", { a: "foo" } );
```

If a itself is an object, its properties can be accessed in the same way:

```
print("^(1.a.b)", { a: { b: "bar" } } );
```

The print() procedure automatically tries to convert the printed arguments to strings. Types such as objects, functions and pointers cannot be printed this way. For some of these types, the function dumpvalue can be used.

It is also possible to format the output directly by adding some of the format modifiers. This can either be an “f” for “fill character ” or a “w“ for “fixed width”. Format modifiers can only be added in parentheses, must be preceded with a colon and followed by a format value. A negative width makes the text left-aligned. Examples:

```
print("^(1:w6)", 3.14);     // Prints "  3.14"
print("^(1:w-6)", 3.14);    // Prints "3.14  "
print("^(1:f#:w6)", 3.14); // Prints "##3.14"
```

Return Value

None.

Context

Stand-alone, Server command-line.

sprint(format, ...)

The sprint() function behaves like the print() function except that it returns the formatted string instead of printing it.

Return Value

The formatted string.

Context

All

discardoutput()

This procedure discards all output generated so far.

Return Value

None.

Context

All.

dumpvalue(a [, { escape | settings }])

dumpvalueln(a [, { escape | settings }])

These procedures dump the value a to the standard output file in a format compatible with the JavaScript language syntax. This makes it possible to easily export M-Script values to JavaScript programs. The dumpvalueln() function does exactly the same as dumpvalue() but adds a newline.

Some M-Script types cannot be converted to text with dumpvalue(). Among these are procedures and functions, as well as all pointer types. This includes all stream types, even text streams. A value of a pointer type appears in the output simply as the name of the pointer type (for example, ostream).

dumpvalue() may also be called with a second optional parameter. This can either be a Boolean parameter (strict), or an object (settings) which defines the layout of the output. Currently two Boolean properties can be specified in this object:

If escape is set to true, special tokens such as tabs and newline within strings in the output are converted to escape codes within the string. This ensures that dumpvalue prints its output in a way that readvalue can interpret. The default value is false.

If noWhitespace is set to true, indentation in the output is omitted. If sending large compound values over a network connection, setting this option could considerably increase the transfer speed.

The settings object is introduced in M-Script 6.0 as a replacement for the single optional escape parameter, in order to allow more detailed control over the dump format. For backward compatibility, passing just a Boolean flag corresponds to the object { escape: value }.

Return Value

None. See also file::dumpvalue(.

Context

Stand-alone, Server command-line.

readvalue(literal)

This function reads a literal value from a string. The format of the string is exactly the same as the output from dumpvalue() on a value containing no unsupported types. The return value from the function is an object with three or four properties; value, ok, and message, and if ok is false the line number lineno where the read failed. The value property is the actual value of the literal, ok is a Boolean indicating the success or failure of the parse operation, and message is an optional text string that describes why the parse failed.

Example

```
var x = readvalue("{a:10, b:[20,30], c:\"John\"}");
```

Yields the following object:

```
{
   value:{
      a:10,
      b:[
         20,
         30
      ],
      c:"John"
   },
   ok:true
}
```

See also file::readvalue().

Context

All.

getelapsedmilliseconds()

This function returns a timer, which is a kind of a timestamp. It is not possible to retrieve the value of a timer, but it is possible to subtract to timers and get the time difference in milliseconds.

Example

```
var t1 = getelapsedmilliseconds();
// do something
var t2 = getelapsedmilliseconds();
var diff = t2 - t1; // diff is now the difference between
                    // the two instantiations of the timers
```

Context

All.

Date

mkdate(day, month, year)

This function returns a date constructed from its input arguments. All arguments must be integer values. The year must be given with all four digits. Both day and month count from one.

Return Value

The constructed date.

Example

```
mkdate(21,8,2000) = August 21st, 2000.
```

Context

All.

getdate()

Return Value

Today's date.

Context

All.

getyear(date)

Return Value

The year of the passed date.

Context

All.

getmonth(date)

Return Value

The month of the passed date (1 ... 12).

Context

All

getmday(date)

Return Value

The day of the month of the passed date (1 ... 31).

Context

All.

getwday(date)

Return Value

The day of the week of the passed date (1 = Monday ... 7 = Sunday).

Context

All.

getweek(date)

Return Value

The week number of the passed date according to the ISO 8601:1988 norm. This norm states that a week starts on a Monday and the first week of a year must contain a Thursday.

Context

All.

Time

mktime(hours, minutes, seconds)

Return Value

The time constructed from the arguments.

Context

All.

gettime()

Return Value

The current time.

Context

All.

gethours(time)

Return Value

The hour of the passed time.

Context

All.

getminutes(time)

Return Value

The minutes of the passed time.

Context

All.

getseconds(time)

Return Value

The seconds of the passed time.

Context

All.

String

strpos(text, pos, match)

Return Value

The first position after (or including) pos in the string text where all of the string match is found (or null if match is not found).

Context

All.

Examples

```
strpos("abcdeabcde", 0, "cde")  // = 2
strpos("abcdeabcde", 0, "xx")   // = null
strpos("abcdeabcde", 3, "cde")  // = 7
```

If the text string is empty, strpos always returns null.

For finding more complicated patterns in a string, see regex::find.

strrpos(text, pos, match)

Return Value

The last position before (or including) pos in the string text where all of the string match is found (or null if match is not found).

Context

All.

Examples

```
strrpos("abcdeabcde", 9, "cde")  // = 7
strrpos("abcdeabcde", 9, "xx")   // = null
strrpos("abcdeabcde", 6, "cde")  // = 2
```

If the text string is empty then strrpos always returns null.

strsplit(text, delimiter [, skipws])

This function splits the string text into a set of substrings. The substrings are considered to be separated by the first character in the delimiter string. If the third optional Boolean parameter is true, whitespace before and after the delimiters is removed.

Return Value

An array of the substrings.

Context

All.

Examples

```
strsplit("a;b;c", ";");          // [ "a", "b", "c" ]
strsplit(" a; b ;c", ";");       // [ " a", " b ", "c" ]
strsplit(" a; b ;c", ";", true); // [ "a", "b", "c" ]
strsplit("a;;", ";");            // [ "a", "", "" ]
```

For splitting strings by more complicated patterns, see regex::split.

array2string(arr [, delim [, lastdelim]])

arraytostr(arr [, delim [, lastdelim]])

Return Value

A string constructed from all the elements in the array arr. Each element must be a simple type. The elements are converted to strings using M-Script type cast. If the string delim is supplied, it is inserted between each element. If the string lastdelim is also supplied, it is inserted between the last two elements instead of delim.

Context

All.

Examples

```
var arr = ["Bob", "Alice", "Mark"];
arraytostr(arr);                 // "BobAliceMark"
arraytostr(arr, ", ");           // "Bob, Alice, Mark"
arraytostr(arr, ", ", ", and "); // "Bob, Alice, and Mark"
```

replace(string, pattern, replacement)

Return Value

A string where all occurrences of pattern in string are replaced with replacement.

No wildcards or other special characters in pattern are allowed.

Context

All.

tolower(text)

Return Value

The string text converted to lower case.

Context

All.

toupper()

Return Value

The string text converted to upper case.

Context

All.

capitalize(text [, allwords])

Return Value

The string text with the first character converted to upper case. If the bool allwords is set and is true then the first letter in every word in text is converted to upper case.

The first character in text, underscore ('_') and any alpha numeric character ('a'-'Z' and '0'-'9') following a non-alpha numeric character or underscore is taken to be the start of a word.

Context

All.

Examples

```
var text = "hi, my name is Bob";
capitalize(text);       // " Hi, my name is Bob"
capitalize(text, true); // " Hi, My Name Is Bob"

var weird = capitalize("a,a.a a a\ta:a_a;a1a#a", true);
capitalize(weird);       // "A,a.a a aa:a_a;a1a#a"
capitalize(weird, true); // "A,A.A A AA:A_a;A1a#A"
```

strtoint(text)

Return Value

An array of integers, each of which corresponds to the ASCII value of the characters in the text string.

Context

All.

strtoint(text, pos)

Return Value

The integer ASCII value of a character in the string text at position pos.

Context

All.

inttostr(ints)

Return Value

A string constructed from the integer array ints by interpreting each value as an ASCII character.

Context

All.

trim(text [, str])

Return Value

A string where whitespace characters have been removed from the beginning and end of the string text. By default, white space is considered to be any of the following characters: '\n', '\r', '\t', '\0' and '\x0B'. The default whitespace characters can be overwritten with the string str.

Context

All.

Examples

```
var text = " my name is Bob ";
trim(text);       // "my name is Bob";
trim(text, " obm"); // "y name is B";
```

ltrim(text [, str])

Works like trim but trims only from the left of the string text.

rtrim(text [, str])

Works like trim but trims only from the right of the string text.

left(text, n)

Return Value

Returns at most n characters from the beginning of the string text.

Context

All.

right(text, n)

Return Value

Returns at most n characters from the end of the string text.

Context

All.

substr(text, start [, length])

Return Value

A string starting at position start in the string text. If start is negative, the start position is offset from the end of text. If the integer length is supplied, at most length characters is returned.

Context

All.

Examples

```
var text = "sunshine";
substr(text, 2);      // "nshine" - opposite of left(text, 2)
substr(text, -2);     // "ne"     - same as right(text, 2)
substr(text, 5, 2);   // "in"
substr(text, 5, 8);   // "ine"
substr(text, -4, 10); // "hine"
```

Math

abs(x)

Return Value

The absolute (positive) value of the passed integer, real, or amount value.

Context

All.

ceil(x)

Return Value

The smallest non-fractional value larger than or equal to the passed integer, real, or amount value.

Context

All.

exp(x)

Return Value

The value of the number e raised to the power of the passed integer or real value.

Context

All

floor(x)

Return Value

The largest non-fractional value smaller than or equal to the passed integer, real, or amount value.

Context

All.

log(x)

Return Value

The natural logarithm value of the passed integer or real value. A run-time exception is thrown if x is smaller than or equal to zero.

Context

All.

pow(x, y)

Return Value

The value of the passed integer or real value x raised to the power of the passed integer or real value y. x and y need not be of the same type. A run-time exception is thrown if x is zero and y is less than or equal to zero. A run-time exception is also thrown if x is negative and y is not an integer.

Context

All.

round(x)

Return Value

The passed integer, real, or amount value rounded to the nearest non-fractional value of x. If the fractional part is 0.5, the return value is rounded to the smallest non-fractional value larger than x.

Context

All.

sqrt(x)

Return Value

The square root value of the passed integer or real value. A run-time exception is thrown if x is negative.

Context

All.

Web

setcontenttype(type)

All output from M-Script is prepended with a HTTP header that contains a Content-type line. This line is used by the web browser to identify how it should present the content of the web page. The default content type is text/html, but other values can be set with the setcontenttype() function. The type argument to this function is a text string that should be printed instead of “text/html”.

Context

Stand-alone.

httpheaderset(header, value)

This function can be used to overwrite existing HTTP headers that are either generated by default by M-Script or which have been set with earlier calls to either httpheaderset() or httpheaderadd().

The header parameter is the name (string) of the required HTTP header and value is the actual header value (string).

Context

Stand-alone.

Example

```
httpheaderset("Content-Disposition", "inline; filename=report.pdf");
```

httpheaderadd(header, value)

This function can be used to add an extra HTTP header in addition to existing HTTP headers of the same name, thereby allowing multiple instances of the same header.

The header parameter is the name (string) of the required HTTP header and value is the actual header value (string).

Context

Stand-alone.

httpheaderdelete(header)

Removes all instances of the HTTP header named header (string).

Context

Stand-alone.

httpheaderexists(header)

Returns true if the requested header has been set already and false otherwise.

Context

Stand-alone.

escape(text)

The escape() function converts all space, punctuations, accented characters, and any other characters that are not ASCII letters to the form %xx where xx is the two hexadecimal digits that represent the ISO-8859-1 (Latin-1) encoding of the characters.

Return Value

The escaped string.

Context

All.

Example

```
escape("Halløj Bøje!")  // = Hall%F8j%20B%F8je%21
```

unescape(text)

The unescape() function returns the inverse of the escape function.

Return Value

The unescaped string.

Context

All.

urlencode(text)

The urlencode() function converts a string in the same way as the escape function except that it converts spaces to plus ('+') signs.

Return Value

The URL encoded string.

Context

All.

urldecode(text)

The urldecode() function returns the inverse of the urlencode function.

Return Value

The URL decoded string.

Context

All

base64encode(ostream, string | istream)

The base64encode() function encodes the content of a string or an input stream to Base64 representation and writes it on the specified output stream.

Return Value

None

Context

All.

base64decode(ostream, string | istream)

The base64decode() function decodes the Base64 encoded content of a string or an input stream and writes it on the specified output stream.

Return Value

None

Context

All.

quotedprintabledecode(ostream, string | istream, bool strict = false)

The quotedprintabledecode() function decodes the quoted printable encoded content of a string or an input stream and writes it on the specified output stream. The optional argument strict specifies whether the algorithm should try to resolve (small) errors in the input stream. The algorithm tries this by default (strict=false).

Return Value

None

Context

All.

htmlencode(ostream, string | istream), htmlencode(string)

The htmlencode() routines encode data for web pages by escaping non-printables and all 8-bit characters to printable 7-bit HTML escape sequences on the form &...; or &#...;.

The first prototype is a procedure which reads from the specified string or input stream and writes on the specified output stream. There is no return value.

The second prototype encodes the specified string and returns the result.

Return Value

None for the first prototype, string for the second prototype.

Context

All.

htmldecode(ostream, string | istream), htmldecode(string)

The htmldecode() routines decode data encoded for web pages by converting all HTML escape sequences on the form &...; or &#...; back to their corresponding character value.

The first prototype is a procedure which reads from the specified input stream and writes on the specified output stream. There is no return value.

The second prototype decodes the specified string and returns the result.

Return Value

None for the first prototype, string for the second prototype.

Context

All.

POP3

Post Office Protocol Version 3 (POP3) is a protocol for retrieving mails from a server that holds mails for the client. POP3 is described in W3C RFC 1939. The POP3 protocol defines an unencrypted protocol where all communication including authentication information is sent through the network in plain text.

The POP3 package allows the M-Script programmer to retrieve messages from a POP3 server and to delete messages from the server.

The following is a small sample program showing some of the basics of retrieving messages from the server, extracting information from the messages, and deleting the messages from the server. See also the description of the function getrawmessage for an example of how to forward a message as an attachment.

```
#version 13

var conn = pop3::open("bm", "ppu", "linux-tools");
var count = pop3::messagecount(conn);

if( count > 0 )
{
   print("There are ^1 message(s) on the server\n", count);

   var hd = pop3::getheader(conn, 0);
   print("The first mail is from \"^1\" <^2>\n", hd.From.name,
   hd.From.email);

   print("The subject(s) of the message(s) are:\n");
   var msgs = pop3::getheaders(conn);
   for( var i in msgs )
      print("^1: ^2\n", i, msgs[i].Subject);

   pop3::deletemessages(conn);
   print("All message(s) have been deleted\n");
}
else
   print("There are no messages on server!");
```

With the exception of pop3::open() all of the functions in this library take one or both of the following two arguments.

Argument	Description
Connection	This argument is always required as first argument. It is the POP3 connection variable returned by a call to pop3::open().
Id	An integer argument determining the selected message. Messages are numbered starting from 0.

Context

The permitted context for functions and procedures in the pop3:: package is All.

pop3::open(user, pass, host[, port[, timeout]])

This function tries to establish a connection to a POP3 server on host and log on with user and pass. port is optional and the default port number is 110. timeout sets the time in milliseconds after which the network connection should close if unused. The default value is 10000.

Return Value

A POP3 connection handle used by all the other functions in the library.

Throws

If the attempt to connect and log on to the POP3 server fails error::stdlib::pop3 is thrown with an error message stating the problem.

Example

```
// Use default port and timeout
var conn = pop3::open("me", "mypass", "my.pop3server.com");
```

The POP3 protocol only allows one concurrent connection to a particular mailbox. An M-Script POP3 connection cannot be closed explicitly but will automatically close and delete any message marked for deletion when the connection variable goes out of scope.

pop3::messagecount(connection)

Return Value

The integer number of messages on the server. connection is the identifier retrieved by pop3::open().

pop3::getrawheader(connection, id)

This function fetches the raw header of a message indicated by id from the server. The raw header is an object with a number of properties that correspond to the header fields in the message read from the POP3 server. Each field becomes a property in the raw header object and has the following structure:

```
{
   value: string,
   specifiers:@{
      <specifiers>
   }
```

```
}
```

where <specifiers> is a number of string properties with property name according to the specifier. See the example below.

Although RFC 822 and 2045-2049 recommends a particular casing for known header fields, then field names, field values, and specifiers are still case-insensitive. This function preserves the casing from the message even when the casing does not conform to the recommended casing.

Only specifiers in MIME fields (that is, fields whose names start with Content) are parsed. All other fields are parsed as unstructured data that consists of plain text.

Throws

If the user tries to fetch the header of a message out of bounds or of the header of a message marked for deletion error::stdlib::pop3 is thrown.

Example

```
// A stripped down example of a raw header object
@{
    // Date: Tue, 4 Nov 2003 16:15:51 +0100
       Date:[
          {
             value:"Tue, 4 Nov 2003 16:15:51 +0100",
             specifiers:@{
             }
          }
       ],
    // Content-Type: multipart/report; report-type=delivery-status;
    // boundary="QAA06413.1067958951/linux-tools.maconomy.dk"
       Content-Type:[
          {
             value:"multipart/report",
             specifiers:@{
                report-type:"delivery-status",
                boundary:"QAA06413.1067958951/linux-tools.maconomy.dk"
             }
          }
       ]
    }
```

pop3::getheader(connection, id)

Like getrawheader() this function retrieves the header of the selected message. But unlike getrawheader() this function only includes the subset of header fields in the selected message that are known by the package2 and adjusts the casing of the field names according to the recommendations in RFC 822 and 2045-2049. The function guarantees the existence of the fields To, From, cc, bcc, Reply-To, Date, Subject, and Content-Type.

The included fields are formatted according to the type of their contents. The structure of the property of some of the known fields is:

```
Content-Type:{
   media:string,
```

```
    sub:string,
    specifiers:{
        <specifiers>
    }
}

MIME-Version:{
    major:int,
    minor:int
}

Subject:string
```

where <specifiers> is a set of properties as described in pop3::getrawheader(). The fields Date and Resent-Date are objects with the following structure:

```
{
    UTCDate:date,
    UTCTime:time
}
```

Note that if possible, dates are converted to UTC date and time. If conversions fail, null-date and null-time are used.

The fields Reply-To, From, Sender, Resent-From, Resent-Sender are address objects with the following structure:

```
{
    name:string,
    email:string
}
```

Known fields are Date, Resent-Date, Reply-To, From, Sender, From, Resent- Reply-To, Resent-From, Resent-Sender, Resent-From, To, Resent-To, cc, Resent-cc, bcc, Resent-bcc, Content-Type, Subject, MIME-Version. The fields Resent-Reply-To, To, Resent-To, cc, Resent-cc, bcc, Resent-bcc are properties with an array of addresses where the structure of each address is given above.

Throws

If the user tries to fetch a header of a message out of bounds or the header of a message marked for deletion, error::stdlib::pop3 is thrown.

Example

```
{
    Date:{
        UTCDate:04.11.2003,
        UTCTime:15:15:51
    },
    From:{
        name:"Mail Delivery Subsystem",
        email:"MAILER-DAEMON@bar.maconomy.dk"
    },
```

```
    To:[
        {
            name:"",
            email:"foo@bar.maconomy.dk"
        }
    ],
    MIME-Version:{
        major:1,
        minor:0
    },
    Content-Type:{
        media:"multipart",
        sub:"report",
        specifiers:@{
            report-type:"delivery-status",
            boundary:"QAA06413.1067958951/bar.maconomy.dk"
        }
    },
    Subject:"Warning: could not send message for past 4 hours",
    cc:[
    ],
    bcc:[
    ],
    Reply-To:{
        name:"",
        email:""
    }
}
```

Some of the fields in the example might not actually be present in the message, but the fields are nevertheless ensured by the package. This behavior is used to ease the use of the package such that the developer can rely on the existence of a set of commonly used header fields and to avoid placing the burden of parsing data on every developer using the package.

Default values are used if the field is not present in the message. The default values for guaranteed fields that contain addresses are empty strings. For fields that contain dates the default is a null-date and a null-time. For Subject the default is the empty string. The default value for Content-Type is text/plain; charset=us-ascii as stated in RFC2045

pop3::getrawheaders(connection)

Return Value

An array of raw header objects as described in pop3::getrawheader(). Raw headers of messages marked for deletion are null objects.

pop3::getheaders(connection)

Return Value

An array of header objects as described in pop3::getheader(). Headers of messages marked for deletion are null objects.

pop3::getrawmessage(connection, id)

This function is useful when forwarding a message as an attachment. See the following example.

Return Value

A stream with the raw contents of the message.

Example

```
// This example uses the M-Script package mscript::mail version 2.1.
// Some details are left out.
//    conn : a valid POP3 handle
//    id   : mail integer index retrieved somehow uses mscript::mail(2.1);

var id  = ...;
var conn = ...;

var rawMail = pop3::getrawmessage(conn, id);
var mail    = pop3::getmessage(conn, id);

// Last argument is an array of attachment object structures.
//    content_type : use "message/rfc822" to specify that
//                   the attachment is a message
//    raw          : must be true to avoid that the attached
//                   message is encoded as a single entity mscript::mail::send(
          "McOnomy <foo@bar.dk>",          // To "Donald Duck",
          // From "FW: " + mail.header.Subject,   // Subject
          "Foo",                           // Message text
          [ { name: mail.header.Subject,  // Attachment...
              content_type: "message/rfc822",
              data: rawMail,
              raw: true } ] );
```

pop3::getmessage(connection, id)

This function retrieves the selected message from the server. The package will recursively parse nested multipart messages and build corresponding recursive data structures. The recursive data structure is an example of the composite design pattern.

The returned object always has a kind property. If kind is “part” the object has the following layout:

```
{
   kind      : "part",
   header    : <header>,
   rawHeader : <rawHeader>,
```

```
    length     : int,
    data       : istream
}
```

where <header> and <rawHeader> are objects as described in pop3::getheader() and pop3::getrawheader(). length is the length of the istream data, which holds the contents of the body.

If kind is multipart, which is true for all multiparts and parts of type message/rfc822, the layout of the object is as follows:

```
{
    kind       : "multipart",
    header     : <header>,
    rawHeader  : <rawHeader>,
    parts      : [
        <part-list>
    ]
}
```

where <part-list> is an array of objects where each of the objects can be either of the two types described above but need not all be of the same type. This is clearly a recursive definition of the object structure.

An object in the recursive data structure always have a kind property, a header property and a rawHeader property. Besides these properties there will be either a body and its length or a parts property that holds a list of child objects witch each again has either kind part or multipart.

Guaranteed fields as described in pop3::getheader() only apply to the header property of the topmost object and objects contained in the parts property of objects with Content-Type message/rfc822. See the examples in the Appendix for a complete example of this recursive data structure.

Throws

If the user tries to fetch a message out of bounds or a message marked for deletion error::stdlib::pop3 is thrown.

Return Value

The following is a fairly stripped down example of a multipart message that has an attached PDF file. The complete example can be found in the Appendix.

```
{
    kind:"multipart",
    header:{
        Content-Type:{
            media:"multipart",
            sub:"mixed",
            specifiers:@{
                boundary:"----=_NextPart_000_001B_01C3B28D.DBE3D420"
            }
        }
    },
    rawHeader:@{
```

```
        Content-Type:[
            {
                value:"multipart/mixed",
                specifiers:@{
                    boundary:"----=_NextPart_000_001B_01C3B28D.DBE3D420"
                }
            }
        ]
    },
    parts:[
        {
            kind:"part",
            header:{
                Content-Type:{
                    media:"application",
                    sub:"pdf",
                    specifiers:@{
                        name:"DocBox.pdf"
                    }
                }
            },
            rawHeader:@{
                Content-Type:[
                    {
                        value:"application/pdf",
                        specifiers:@{
                            name:"DocBox.pdf"
                        }
                    }
                ],
                Content-Disposition:[
                    {
                        value:"attachment",
                        specifiers:@{
                            filename:"DocBox.pdf"
                        }
                    }
                ]
            },
            length:5464,
            data:istream
        }
    ]
}
```

pop3::getmessages(connection)

Return Value

An array of message objects as described in pop3::getmessage(). Messages marked for deletion are null objects.

pop3::deletemessage(connection, id)

This function marks the selected message for deletion. Note that according to the POP3 protocol the message will not be deleted before connection to the server is closed. If the connection is closed due to an error (for example, a server timeout or a network error), messages marked for deletion are not deleted.

If pop3::getmessages() or pop3::getheaders() is called after pop3::deletemessage() or pop3::deletemessages() messages marked for deletion is null.

Throws

If a program tries to mark again a message that is already marked for deletion, error::stdlib::pop3 is thrown. This is a design decision, which is made because it is assumed that it is better to indicate an error than a feature if a program tries to delete the same message more than once.

If a message out of bounds is selected error::stdlib::pop3 is thrown.

pop3::deletemessages(connection)

This function marks all messages in the mailbox for deletion. See also pop3::deletemessage().

pop3::undeletemessages(connection)

Because the POP3 server does not really delete messages before the connection to the client is closed it is possible to undelete messages selected for deletion during the lifetime of the connection. It is not possible to undelete messages individually.

Session

newsession()

The newsession() procedure creates a new session. If that is not possible, a run-time error is generated.

Return Value

None.

Context

Stand-alone, Server command-line.

loadsession(sessionId)

The loadsession() procedure loads an existing session. If the session does not exist, a run-time error is generated. It is only possible to call loadsession() once in a program and only of no session already exists. loadsession() must be called before any session variables are declared.

Return Value

None.

Context

Stand-alone, Server command line.

hassession()

Return Value

True if a session is open, or false if it is not. From M-Script version 10.0 this function not only tests for the availability of a session—it also locks the session if possible. So if a session exists, M-Script locks the session and returns true. If the locking fails, an exception is thrown. If no session exists, this function returns false.

Context

Stand-alone, Server command-line.

link(url)

The link() function adds a session identifier to the supplied URL (string).

Context

Stand-alone.

Examples

```
link("maconomy.com")      // = maconomy.com?sessionid=qFI6
link("intro?user=john")   // = intro?user=john&sessionid=qF16
```

A session must be open in order to use this function

currentsession()

Return Value

The current session identifier.

Context

Stand-alone, Server command-line.

dumpsession()

The dumpsession() procedure prints all defined session variables to the standard output file.

Return Value

None.

Context

Stand-alone, Server command line.

deletesession()

The deletesession() procedure deletes all storage used for the session variables and closes the session.

Return Value

None.

Context

Stand-alone, Server command line.

Run-Time Logging

M-Script contains some facilities for inspecting the run-time environment during execution. These facilities might be useful for program optimization and source management, as well as for post-mortem debugging sessions.

When run-time logging is activated, it is possible to retrieve information about which packages and values are being loaded, as well as information about the call stack and various other information that has been logged.

The log contains four major data structures:

- A script log — This array contains one entry for each script that has been loaded while logging is activated.
- A value log — This log contains one entry for each value that has been read using file::readvalue().
- A run-time log — This array can be used to store arbitrary information during execution. The information is available throughout the M-Script invocation. Entries are added to the run-time log with the function addlogentry().
- A stack trace — When logging is activated, a number of operations write entries on the run-time stack. These entries are scope-dependent, meaning, for instance, that when a function returns, all entries that have been added since the function was called are truncated from the stack. It is possible to write entries to the stack with the function addstackentry().

The entries in each of these logs are objects with the following properties.

Entry	Type	Description
filename	string	The name of the file this entry originates from. The value of this property is null in all entries in the run-time log and in entries added manually to the stack trace.
lineno	int	The relevant line number. The value of this property is null in all entries in the value and run-time logs.
description	string	A short description of the entry.
value	any type	Additional information. For the script and value logs, this property contains TAC3 license information for the script or value.

The logging state is automatically saved to and restored from the session, if one exists.

The recorded information, however, is discarded when the M-Script interpreter exits, so if a logging cycle spans several interpreter invocations it is necessary to collect the recorded information from each invocation separately.

islogging()

Return Value

true if logging is currently active, false otherwise.

setlogging(bool)

The setlogging() function activates or deactivates logging based on the value of the boolean argument.

Return Value

true if logging was previously active, false otherwise.

Context

All.

getlog()

The getlog() function returns the current log as an object with five properties:.

logging	bool	The current logging state.
scripts	array	The script log as described in section 13.9.
values	array	The value log as described in section 13.9.
log	array	The run-time log as described in section 13.9.
stack	array	The stack trace as described in section 13.9.

Context

All.

dumplog(stream, format [, params]))

The dumplog() function writes a session log to the stream stream. The log is written in the format specified by the string format. Currently the only supported format is xml. See the following example for an example of the layout in XML format.

The optional argument params is an object where any of the following properties are recognized.

portalVersion	string	If supplied it is used as a version attribute on the portal tag in the XML layout.
maxLogfileLines	int	The maximum number of lines to include from the end of the M-Script log file. A negative number selects all lines. Default is all lines.

XSLStylesheet	string	If an XSLT style sheet is supplied, it is inserted into the XML source of the log.
maconomy	object	Any format including nested properties is allowed. The object is converted according to the specified format. The following an example where the object has been used to supply various server and portal information.

TAC: Tool Access Control.

Context

All.

This function requires a session.

Example

The following is a slightly stripped down sample output from a portal session.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="portallog.1.xsl"?>
<sessionlog sessionid="Y6PzeR98o9">
   <maconomy>
      <portal_error_log date="2004.04.23" time="13:39:20" />
      <maconomy_version_information application="DK_8_0 patch 11"
      server="34.00.0.340054 Pak 10" portal="9.0.0b5" />
      <database_information database_type="oracle" short_name="dk80dem2" />
      <web_server_information server_software="Apache/2.0.47 (Win32)"
      server_protocol="HTTP/1.1" server_name="cph-pc353"
      mscript_executable="/cgi-bin/Maconomy/MaconomyPortal.dk80.exe"
      mscript_version="14.0.0.340054 Pak 10" />
      <client_information browser="msie 6.0" operating_system="windows" />
      <user_information portal_user_name="Administrator" />
   </maconomy>
   <mscript major="14" minor="0" raw="14.0.0.340054 Pak 10">
      <inifile filename="C:/Apache2/maconomy/bm/8.0/cgi-
      bin/Maconomy/MaconomyPortal.dk80.I">
<![CDATA[
]]>
      </inifile>
   </mscript>
   <servers>
      <connection label="DEFAULT:SERVER">
         <serverinfo shortname="dk80dem2" language="DK" database="oracle">
```

```
            <server raw="34.00.0.340054 Pak 10" major="34" minor="0" />
            <application raw="DK_8_0" country="DK" major="8" minor="0"
            patch="11" subpatch="" />
            <installation companyName="DK 80 Demo 2" originalCompanyName="DK
            80 Demo 1" registrationNo="723211915" />
         </serverinfo>
      </connection>
   </servers>
   <portal version="">
      <logfile filename="C:/Apache2/maconomy/bm/8.0/cgi-
      bin/Maconomy/MaconomyPortal.dk80.log">
<![CDATA[
]]>
      </logfile>
      <runtimelogs>
         <runtimelog id="1" date="23.04.2004" time="13:35:46">
            <scripts>
               <entry filename="Framework/packages/portal/std/localize.1.ms"
               lineno="" description="Loaded script
               &apos;Framework::packages::portal::std::localize(1)&apos;">
                  <data>
                     <license TAC_Author="Release Manager Technology"
                     TAC_Organization="Maconomy R&amp;D"
                     TAC_CreationDate="31.03.2004"
                     TAC_CreationTime="12:34:34"
                     TAC_UseDialogAccessList="false"
                     TAC_AllowIgnoreDialogAccessControl="false">
                        <TAC_AddOns>64</TAC_AddOns>
                        <TAC_LoginPrivileges />
                     </license>
                  </data>
               </entry>
            </scripts>
            <values>
               <entry filename="" lineno="" description="file::readvalue:
               Parsed expression from stream">
                  <data license=""
                  source="C:/Apache2/maconomy/bm/8.0/MaconomyPortal/Framework/
                  Themes/Default/portal.I">
<![CDATA[
]]>
                  </data>
               </entry>
            </values>
            <stack></stack>
            <log></log>
         </runtimelog>
      </runtimelogs>
   </portal>
</sessionlog>
```

addlogentry(description, value)

The addlogentry() function adds a log entry to the end of the run-time log array. The description and value properties of the entry are set according to the specified arguments. The remaining properties are set to null. If logging is not currently active the log entry is ignored.

Return Value

True if logging is active, false otherwise.

Context

All.

addstackentry(description, value)

The addstackentry() function adds a log entry to the top of the stack trace array. This log entry persists until the stack is truncated to a point below where the entry was added. The description and value properties of the entry are set according to the specified arguments. The remaining properties are set to null. If logging is not currently active the log entry is ignored.

Return Value

True if logging is active, false otherwise.

Context

All

Other

getmscriptinfo()

Return Value

An object with the following attributes.

Major	string	Major version number of the M-Script interpreter.
Minor	string	Minor version number of the M-Script interpreter.
raw	string	Raw image of the version number.

Context

All.

getscriptinfo([handle])

Retrieves various information about the interpreter, the current script, or the package selected by the optional handle argument.

Return Value

An object with the following attributes.

version	object	Language version of the M-Script interpreter.
isMain	bool	True if the current script is the main script. false if the script is a package.
license	object	License information about the current script.
filename	string	Filename of the current script.
package	object	Only present if the current script is a package—that is, isMain is false.

The version object has the following attributes.

major	int	Major version number of the M-Script interpreter.
minor	int	Minor version number of the M-Script interpreter.
revision	int	Raw image of the version number.

The package object has a version property too, which is the same.

The license object has the following properties.

TAC_Author	string	The identity of the licensee.
TAC_Organization	string	The organization of the licensee.
TAC_CreationDate	date	The current date. (D.M.Y)
TAC_CreationTime	time	The current time. (H:M:S)
TAC_History	array	Creation history of the stamped file. More than one entry may exist. These fields come before any other entries, and the top-most entry is the most recent.
TAC_AddOns	array	A comma-separated list of add-on numbers which are required on the server. Several such lists can be specified.
TAC_CustomerNumbers	array	A customer number that the licensee is allowed to stamp scripts for. More than one entry may exist.

TAC_LoginPrivileges	object	Special login privileges on the Maconomy server.
TAC_UseDialogAccessList	bool	A Boolean value that specifies whether the Maconomy server 's dialog access list should be checked when scripts invoke the M-Script Maconomy API. If set to "yes," the dialogs accessed by the script must be in the dialog access list. If omitted on the command line, it defaults to "yes," but if absent in a stamp, it defaults to "no."
TAC_AllowIgnoreDialogAccessControl	bool	A Boolean value that specifies whether the script is allowed to disable dialog access control when opening dialogs. This enables the script to access dialogs that the user is otherwise not allowed to open.

Context

All.

exit(code)

The exit() procedure stops execution of the M-script and returns the passed value to the calling program (typically the web server). The use of exit does not influence the execution of the start and end scripts described in Optional M-Script Parameters.

Return Value

None.

Context

Stand-alone, Server command line.

random(low, high)

Return Value

A random integer number in the range low ... high (both included).

Context

All.

getenv(key)

Return Value

The value of the environment variable named key, or null if the variable does not exist.

Context

All.

getfullenv()

Return Value

An object with a property for each defined environment variable. The value of the properties are strings that correspond to the values of the environment variables.

Context

All.

getinisetting(key)

Return Value

The value of a specific key in the initialization file. The passed key is a string, and the returned value is also a string. If the key does not exist or has not been set, null is returned. This function returns the current value of the setting, that is, after a suitable section in the initialization file has been selected.

For the keys FileSystemRoot and PackageRoot the default root is returned. For the key Connection null is always returned. Use getxinisetting() (see below) to get the settings for these keys.

Context

All.

getxinisetting(key)

This function is an extended version of getinisetting() function (see above). It can be used to retrieve the setting of fields with multiple values, such as FileSystemRoot, PackageRoot and Connection.

Return Value

An object with the values for the specified key. The object contains a property for each value, with the name of the property being the unique key for the sub-setting. If the key does not exist or has not been set, null is returned.

For file and package roots the object properties are the root names, while it is the server labels for connections. If there is a default value for the key, the corresponding property name of this entry is the key name, converted to all lowercase letters. This is also the case for simple keys without multiple values. All user-defined keys appear exactly as written in the initialization file.

The value of the properties within the returned object depends on the type of setting. Simple values are represented as a string, while compound values such as connections are converted to an object in which each field has its own property.

Context

All.

Example

```
dumpvalue(getxinisetting(FileSystemRoot));
   -->
{
   filesystemroot: ...,
   <firstNamedRoot>: ...,
   <secondNamedRoot>: ...,
   ...
}

dumpvalue(getxinisetting(Connection));
   -->
{
   connection: { serverip: ..., daemonport: ... },
   <firstServerHandle>: { serverip: ..., daemonport: ... },
   <secondServerHandle>: { serverip: ..., daemonport: ... },
   ...
}
// Simple value ServerIP:
dumpvalue(getxinisetting(ServerIP));
   -->
{
   serverip: ...
}
```

loadpackage(path)

Dynamically load a package at run time.

Context

All

setlogmask(mask)

Set the mask used for the log entry in the .I file. The use of this function has an instant effect on the data printed in the log file. The mask value is a string with exactly the same data as might be used on the right-hand side of the = in the .I file.

Context

All.

sleep(sec)

Stop execution of the M-Script program for as many seconds as passed to the function.

Context

All.

system(command [, encoding])

Executes a shell command exactly as if had been executed on the command line by a normal user. If an encoding is specified as a second argument, the command is transcoded to this encoding before being executed and the output produced by the command is assumed to also be in this encoding, and is transcoded back to UTF-8.

In Windows the encoding string may also contain a codepage, separated from the encoding with a horizontal bar (|), for example, as `ISO-8859-1|windows-865`. This codepage is applied to the virtual console before the command is executed. Codepages must be written in one of these forms:

```
windows-<NNN>
DOS-<NNN>
cp<NNN>
IBM<NNN>
```

If the specified encoding and codepage are not compatible the command execution likely fails if the command contains any non-ASCII characters.

Return Value

An object with the following properties:

Name	Type	Description
status	int	The status code returned by the shell that was used to invoke the command. It is not the status returned by the command itself.
stdout	string	All output generated on the standard output of the command.

This command can be exploited in numerous ways when user input is used as part of the command. For instance by accepting a filename form or a web form and then using it for some external command. User input should always be escaped by the use of the shellescape function described in the following.

Context

All.

Example

```
var output = system("ls *.txt");
```

spawn(command, argv, detach)

This procedure works somewhat similarly to system except for a few differences: Arguments to the specified command must be passed in a separate array argument, it is possible to run commands detached from the current process, and it is possible to invoke the M-Script interpreter from a CGI-script.

The command must be a string that contains only the path and name of the command to spawn, while argv is an array of strings that contains the arguments. If the third parameter detach is true, the spawned command runs in parallel with the spawning script; otherwise, the spawning script

waits for the command to finish before continuing. In either case it is not possible for the spawning script to retrieve the return code or the output generated by the spawned command.

Return Value

None.

Context

All.

Examples

The following command spawns the command `run.sh arg1 arg2` and runs it in parallel with its own execution:

```
spawn("run.sh", ["arg1", "arg2"], true);
```

This next command invokes an M-Script interpreter on the script foo.ms with the specified configuration file and session ID and waits for it to finish before continuing:

```
spawn("mscript",
   ["-I", "mscript.portal",
    "-q", link("")[1..],   // skip the prefixed ?
    "foo.ms"
   ], false);
```

spawnscript(argv, detach)

This procedure is a version of spawn specialized for invoking the M-Script interpreter that is running the current script. Therefore, the command parameter is omitted, and the argument array must not contain a configuration file specification. See the preceding description of spawn for a description of the parameters argv and detach.

Return Value

None.

Context

All.

Example

The last example from spawn() above can be simplified with spawnscript:

```
spawnscript(["-q", link("")[1..], "foo.ms"], false);
```

shellescape(str)

Return Value

This function returns a copy of the string str where all special characters are escaped with a backslash.

Context

All.

Example

```
var inputFileName = shellescape(query.values.filename);
var result = system("cat " + inputFileName);
```

coerce(value, typeName)

This function tries to convert the value parameter into the type specified by the typeName parameter, which is a string that names the required type.

Return Value

The type-converted value.

Context

All.

Example

```
var i = coerce("25", "int");    // Convert string to int
var a = coerce(9.95, "amount"); // Convert real to amount
```

Format

The format library provides an interface to the value formatting specifications in M-Script. For each of the built-in value types a format string exists that specifies the text representation for values of that type. These format strings can be specified statically in the M-Script initialization file, but with the format library it is also possible to change the formats dynamically during program execution.

Context

The permitted context for functions and procedures in the format:: package is All.

The Format object

Within M-Script the available format strings are presented in a format object. For each of the customizable formats, this object contains a corresponding property of type string. The complete set of properties in a format object is as follows.

Property	Value
Boolean	Format string for type bool
Integer	Format string for type int
Real	Format string for type real
Amount	Format string for type amount
Date	Format string for type date
Time	Format string for type time

format::get()

Return Value

This function returns a format object containing the current format settings for M-Script value types.

Example

A call to format::get() might return the following format object:

```
{
   Boolean:"Ttrue;Ffalse;",
   Integer:"E,",
   Amount:"E,.",
   Real:".2E,.",
   Date:"E/DDMMYYYY",
   Time:"E:HS"
}
```

format::set(object)

This function takes as argument any subset of the format object, and updates the format settings for the specified properties according to the format strings.

Return Value

The return value is a status object that contains a boolean property status that indicates whether all formats were successfully updated, and a string message that contains an error message if some of the updates failed.

Example

The following call to format::set() updates the format for date and time:

```
format::set( { Date:"E/YYYYMMDD", Time:"E.H" } );
```

Following this function call, the date 24.12.2001 is printed as 2001/12/24, while the time 04:10:30 is printed as 4.10.

format::toString(value)

Converts a simple M-Script value to a string representation using the present formatting settings. This corresponds to casting the value to a string:

```
dumpvalueln(format::toString(7.50A)); // These two statements
dumpvalueln(string(7.50A));           // are equivalent.
-->
7.50
7.50
```

Return Value

A string with a formatted image of the passed argument.

format::toSql(value)

Like toString(), this function converts an M-Script value to a string representation, but the resulting string from toSql() is a representation compatible with the Maconomy SQL interface.

Please note that due to an error correction in M-Script 16.0.0, you must specify 16 in the version tag of your scripts when using this function on M-Script versions 16 and later.

Return Value

A string with an SQL-compatible image of the passed argument.

Stream Operations

The standard library io offers some functionality for stream operations. Streams have been introduced in M-Script as a replacement for the earlier concept of filehandles (see the description of the file package).

A stream in M-Script can be an input stream, an output stream, or in rare cases a combined input-output stream. In addition to this, two main types of streams exist: string streams and data streams. String streams are guaranteed to contain only valid text, and these streams can be readily converted to M-Script strings. Data streams cannot be as easily converted to valid strings, since they may contain any kind of data, including binary data read from files.

Streams cannot be stored in session variables.

io::write(out, in)

The io::write() procedure reads all available data from the input source in to the output stream out. The input can be an input stream or an M-Script string. If the input is a stream, it is empty after the call to io::write().

Context

All.

Throws

error::stdlib::io::write (see section 18.3).

io::read(istream, count)

The io::read() function reads at most count bytes from the input stream istream and returns it as a string. If end-of-file is encountered while reading, the data read so far is returned, except if the stream was already at end-of-file before reading, in which case null is returned.

Return Value

Number of bytes read, or null if stream is already at EOF.

Context

All.

Throws

error::stdlib::io::read.

io::istring(value)

The io::istring() function returns an input string stream initialized to the string value.

Return Value

An input string stream initialized with the specified string.

Context

All.

io::ostring()

The io::ostring() function returns a new output string stream. Data written to an output string stream is kept in memory and can be retrieved with io::getstring().

Return Value

A new empty output string stream.

Context

All.

io::getstring(stream)

The io::getstring() function returns an M-Script string representation of the data in stream. The stream remains unaffected by this operation.

Return Value

The contents of the specified string stream.

Context

All.

Throws

error::stdlib::io::getstring.

io::flush(stream)

The io::flush() function causes a stream containing an output buffer to write this buffer to its destination. A flush command can be useful on file or network streams to make sure that all data is sent to its destination. On streams that do not have an external destination to send data to, such as ostring streams, the flush command is ignored.

Return Value

A boolean that indicates the state of the stream. (true = good, false = bad)

Context

All.

Throws

error::stdlib::io::flush (see section 18.3).

io::stdout()

The io::stdout() function returns a reference to the standard output stream. Anything written to this output stream therefore appears in the standard output of the M-Script program.

Return Value

A reference to the standard output stream.

Context

Stand-alone, Server command-line.

io::stdin()

The io::stdin() function returns a reference to the standard input stream. All data passed to the program's standard input can be read here (unless it is captured by M-Script's built-in CGI-parser).

Return Value

A reference to the standard input stream.

Context

Stand-alone, Server command line.

io::log()

The io::log() function returns a reference to a non-seekable output stream to the M-Script log file. Data written to this stream is appended to the log file along with date, time and file information.

Return Value

An output stream to the M-Script log file.

Context

All.

File Access

With M-Script, it is also possible to access files on the web server. However, a number of security issues are apparent. The most important of these are:

- Web users should not be able read confidential information on the server (such as the password file).
- Web users should not be able to modify or delete important files.

One way to reduce these risks is to restrict M-Script's access to files. Deltek has chosen to do this by requiring the M-Script administrator to explicitly state which parts of the directory structure M-Script has access to. It is possible to specify one default file root and zero or more optional, named file roots. This setting restricts M-Script to access only the files that lie below the specified path, and it is also considered the current working directory and root directory for M-Script.

A default file root can be specified in the initialization file with the line:

```
FileSystemRoot = /your/mscript/data/path
```

while additional file roots are added as follows (add semicolons at the end of each line):

```
FileSystemRoot = /your/mscript/data/path;
root2 = another/folder;
root3 = yet/another/folder
```

In addition to the default folder (/your/mscript/data/path) you now also have access to the folders another/folder and yet/another/folder via the file roots root2 and root3. To use these named file roots, prefix the root to the file name or path:

```
root2/someFile.txt
```

A file must be opened with a call to `file::open()` before it can be read from or written to. This function call returns a file stream that can be used later for reading and writing with, for instance, the functions file::getline() or file::print(). If the specified file cannot be opened, the function either returns null (#version <=5) or throws an exception of type `error::stdlib::file`. A sample program that first writes three lines to a file and then reads them again can look like the following:

```
#version 15
// First open the file for writing
var f = file::open("lines.txt", "w");

// Then print three lines to the file file::print(f, "Line 1\n"); file::print(f,
"Line 2\n"); file::print(f, "Line 3\n");
// And close it again file::close(f);
// Now open the same file for reading f = file::open("lines.txt", "r");

// Read all the lines in the file and print them var line;
while ((line=file::getline(f)) != null)
   print("^1\n", line);
// And close it again file::close(f);

// Finally we remove the file
file::remove("lines.txt");
```

Context

The permitted context for functions and procedures in the file:: package is All.

file::open(filename, mode)

The file::open() function tries to open the file named filename for reading and/or writing, depending on the string passed as mode. The possible values for the mode string are as follows.

Value	Description
"w"	Open the file for writing. This creates the file if it does not exist already. If it exists, then the file is truncated before the next write operation.
"r"	Open the file for reading.
"a"	Open the file for appending. This creates the file if it does not exist already. If it

Value	Description
	exists, then all subsequent write operations append to the file instead of overwriting the content.
"r+"	Open file for both reading and writing. (The file must exist.)
"w+"	Open file for both reading and writing. If the file exists, it is truncated, otherwise the file is created.
"b"	Open the file in binary mode. This flag must be used on Windows NT when opening binary files (PDF files for instance). Files used for file::write, file::read, io::read and io::write should also be opened as binary files.
"s"	Shared mode. Please see the following.
"x"	Exclusive mode. Please see the following.

Return Value

A file stream, or null if the file could not be opened and exceptions are disabled.

Throws

error::stdlib::file::open.

error::stdlib::file::invalidCharacters.

Example

```
// open for write:
   var f1 = file::open("lines.txt", "w");

// open for read:
   var f2 = file::open("lines.txt", "r");

// open for append (file is truncated before write):
   var f3 = file::open("lines.txt", "a");

// open for binary read:
   var f4 = file::open("lines.txt", "rb");

// open for read and write (file is truncated before write):
   var f5 = file::open("lines.txt", "rw");

// open for read and append (data is always appended):
   var f6 = file::open("lines.txt", "ra");
```

If file::open() succeeds, a valid stream pointer is returned. If the call fails, null is returned

If a file is opened in read/write mode (“ra”, “r+”, or “w+”), care must be taken when switching between reading and writing, since reading destroys the state of the output stream and vice versa. To avoid stream corruption, calls to file::seek() with a valid file position should always be performed between read and write operations.

Previous versions of M-Script implemented the file package with file handles, but streams offer a much more flexible way to handle data. The interface to the file package remains semantically backward compatible, so M-Script programs written to use file handles will continue to work with the new implementation using streams.

For the technically interested, it is worth noticing that M-Script by default grants mutually exclusive write and append access. This means that two concurrent M-Script programs cannot write to the same file simultaneously. If one program tries to open a file which is in use by another, then it is put on hold until the other one closes this file, or simply exits. This default behavior can be bypassed if an "s" (shared mode) is added to the mode parameter, such as "ws" or "as". Similarly, it is possible to get exclusive read access to a file if an "x" is added to the mode parameter.

file::close(stream)

This procedure closes an already opened file.

Files are also closed when going out of scope (which also happens if you assign a null value to the file variable).

Throws

error::stdlib::file::close.

Example

```
procedure f()
{
   var f = file::open(...);
   ... use f ...
   file::close(f); // Optional, since f goes out of scope.
}
```

file::getencoding(stream)

Return value

The current encoding for the specified stream.

Context

All.

Examples

```
var f = file::open("lines.txt", "r");
file::getencoding(f);
```

yields the same encoding as a call to `getencoding()` would.

file::setencoding(stream, encoding)

Set the encoding used by the specified stream. Passing an empty string to this function restores the current script encoding to the stream.

Return value

The encoding previously set for this stream.

Context

All.

Examples

```
var f = file::open("lines.txt", "r");
var old_enc = file::setencoding(f, "UTF-8");
println("Old encoding: \"^1\", new encoding: \"^2\"",
         old_enc, file::getencoding(f));
```

might print the string:

```
Old encoding: "ISO-8859-1", new encoding: "UTF-8"
```

file::getsubstitutionstring(stream)

Return value

The current substitution string for the specified stream.

Context

All.

Examples

```
var f = file::open("lines.txt", "w");
file::getsubstitution (f);
```

yields the same substitution string as a call to `getsubstitution()` would.

file::setsubstitutionstring (stream, substitutionstring)

Set the substitution string used by the specified stream. Passing an empty string to this function restores the default substitution string to the stream.

Return Value

The substitution string previously set for this stream.

Context

All.

Examples

```
setsubstitutionstring("_");
```

```
var f = file::open("lines.txt", "w");
var old_sub = file::setsubstitutionstring(f, "?");
println("Old substitution string: \"^1\", setsubstitution string: \"^2\"",
         old_sub, file::getsubstitutionstring (f));
```

prints the string:

```
Old substitution string: "_", new substitution string: "?"
```

file::getsystempath(filename)

Return Value

The full OS system path M-Script would use to open the file if file::open(filename, ...) were called.

Throws

error::stdlib::file::invalidCharacters).

file::tmpname()

Return Value

A temporary filename that can be used for opening a temporary file without overwriting any existing files. Each call to this function results in a new temporary filename, so you must store the first name if you need to reopen the file later.

file::print(ostream, format, ...)

This procedure has the same functionality as print(), except that the output is written to the passed file, instead of to the standard output file.

Throws

error::stdlib::file::print.

file::getline(istream [, delimiter])

This function reads the next text line from the input stream and returns it. If no more input is available, the function returns null.

Return Value

A string with the next line in the input stream, null if no lines are available.

Throws

error::stdlib::file::getline.

Example

```
   // Read all text lines in a file and print them
var f = file::open("lines.txt", "r");
var line;
while ((line=file::getline(f)) != null)
   print("^1\n", line);
```

The optional delimiter argument is a string (one character) that can be used to redefine the newline delimiter. It can for instance be used to read semicolon-separated values.

file::write(ostream, val)

This procedure stores almost any M-Script value in a file for later retrieval with file::read(). The allowed M-Script values are null values, Booleans, integers, reals, amounts, dates, times, strings, objects, and arrays. The output file must be opened as a binary file.

Throws

error::stdlib::file::write (see section 18.2).

Example

```
var b = true;
var o = { a:1, b:"Hello" };
var f = file::open("data.dat", "wb");
file::write(f, b);
file::write(f, o);
```

file::read(istream)

This function reads and returns the next M-Script value stored in the file. The M-Script values must have been stored with file::write() and are read in the same sequence as they were written. The input file must be opened as a binary file.

Return Value

The next M-Script value stored on the specified input stream.

Throws

error::stdlib::file::read.

Examples

```
   // Read the values stored in the previous example
   // (see file::write)
var f = file::open("data.dat", "rb");
var b = file::read(f);
var o = file::read(f);

dumpvalue(b); print("\n");
dumpvalue(o); print("\n");
```

file::copy(from, to)

This procedure copies and/or renames a file or directory. All file permissions on the files and directories being copied are preserved if possible, but otherwise fail silently.

Throws

error::stdlib::file::invalidCharacters.

error::stdlib::file::copy

file::move(from, to)

This procedure moves and/or renames a file or directory. If the source is a directory and the destination is on a different drive partition, the move is accomplished by a directory copy operation, followed by a delete operation on the source directory.

Throws

error::stdlib::file::invalidCharacters.

error::stdlib::file::move.

file::remove(filename)

This function removes (deletes) the specified file or directory. If filename refers to a regular file, it is removed if access permissions allow it. If filename refers to a directory, all of its contents are removed if access permissions allow it.

Return Value

The return value depends on the value of the script's #version tag:

For script versions <13 a boolean is returned, indicating whether the operation succeeded.

For script versions >=13 a status object with the following properties is returned.

Name	Type	Description
ok	bool	Indicates whether the operation succeeded.
message	string	If the operation failed, this property contains the error message.

Throws

error::stdlib::file::invalidCharacters.

file::import(istream, separator)

This function reads a tab or comma separated file into an array.

Return Value

A two-dimensional array, with each inner array containing the string values of one line in the input.

Throws

error::stdlib::file::import.

Examples

```
   // Open file for reading
var f = file::open("data.txt", "r");
   // Import comma separated file
var table = file::import(f, ",");
   // Print result print("Imported table:\n"); dumpvalue(table);
   print("\n");
```

Example input file:

```
Adam,Nicholson,York
Eva,Koch,London
Kai,Nielson,Houston
```

Example output:

```
Imported table:
[ [ "Adam", "Nicholson", "York" ],
  [ "Eva", "Koch", "London" ],
  [ "Kai", "Nielson", "Houston" ]
]
```

The chosen separator may be any single character, for instance \t for a tab,; for a semicolon or , for a comma.

file::export(ostream, array, separator)

This procedure exports a two-dimensional array as a tab- or comma-separated file.

Throws

error::stdlib::file::export.

Examples

```
var a =
[ [ "Adam", "Nicholson", "York" ],
  [ "Eva", "Koch", "London" ],
  [ "Kai", "Nielson", "Houston" ]
];
var f = file::open("data.txt", "w");
   // Export semicolon separated file
var table = file::export(f, a, ";");
```

Output file:

```
Adam;Nicholson;York
Eva;Koch;London
Kai;Nielson;Houston
```

The chosen separator may be any single character, for instance \t for a tab,; for a semicolon or , for a comma.

file::info(filename)

This function returns an object that contains various information about the file specified by filename.

Return Value

An object with file information. The properties in this object are as follows.

Name	Type	Description
size	int	The size of the file in bytes.

Name	Type	Description
isDir	bool	Whether the file is a directory.
isFile	bool	Whether the file is a regular file.
filePath	string	The full file system path of the file.
cdate	object	Creation date (cdate.d) and time (cdate.t).

The function returns null if the file does not exist.

Throws

error::stdlib::file::invalidCharacters.

Examples

```
var info = file::info("data.txt");
dumpvalue(info);
```

Output:

```
{
   size:59,
   isDir:false,
   isFile:true,
   cdate:{
   d:21.02.2001,
   t:14:03:02
}
```

file::makedir(dirname [, options])

This function creates a directory named dirname. By default the parent directory of the new directory must exists, but it is possible to pass an option object as a second argument:

Name	Type	Description
recursive	bool	Set this property to true to create all missing directories in the directory path.

Return Value

True if the operation succeeded and false otherwise.

Throws

error::stdlib::file::invalidCharacters.

Examples

```
file::makedir("testdata");
file::makedir("testdata/john");
file::makedir("testdata/lisa");
```

```
file::makedir("testdata2/peter", {recursive: true});
file::makedir("testdata3/susan"); // will fail if 'testdata3' does not exist.
```

file::removedir(dirname [, options])

This function removes a directory named dirname. By default the directory must be empty, but it is possible to pass an option object as a second argument.

Name	Type	Description
recursive	bool	Set this property to true to delete the directory recursively with all of its contents (corresponds to file::remove()).

Return Value

The return value depends on the value of the script's #version tag:

For script versions <13 a boolean is returned, indicating whether the operation succeeded.

For script versions >=13 a status object with the following properties is returned.

Name	Type	Description
ok	bool	Indicates whether the operation succeeded.
message	string	If the operation failed, this property contains the error message.

Throws

error::stdlib::file::invalidCharacters.

Examples

```
file::removedir("testdata/john");
file::removedir("testdata/lisa");
file::removedir("testdata");
```

Only an empty directory may be removed, so file::removedir("testdata") is illegal if the "john" directory exists or if any other files exist in the directory.

file::readdir(dirname)

This function returns an array of all the files and subdirectories in the directory that is specified by dirname. An empty string, ".", or "/" may be used for the current or top directory. A call to file::info() can be used afterwards to differentiate between files and directories.

Return Value

An array of all files and subdirectories in the specified directory.

Throws

error::stdlib::file::invalidCharacters.

Examples

```
var dir = file::readdir("/");
```

```
dumpvalue(dir);
```

Output

```
[
   "file1.txt",
   "file2.txt",
   "file3.txt"
]
```

file::dumpvalue(ostream, value [, { escape | settings }])

file::dumpvalueln(ostream, value [, { escape | settings }])

The file library contains versions of the functions dumpvalue() and readvalue(). Their behavior is identical to the built-in functions, except that they read and write data using the specified stream.

The procedure file::dumpvalue(stream,value) dumps a text representation of value on the specified output stream. If stream is io::stdout(), the behavior is identical to the built-in dumpvalue.

The dumpvalueln function adds an extra newline after the output but is otherwise identical to dumpvalue.

Throws

error::stdlib::file::dumpvalue.

file::readvalue(istream)

This function tries to read an M-Script value from the specified input stream. The return value is identical to that returned by the built-in function readvalue().

Throws

error::stdlib::file::readvalue.

file::tell(stream)

This function returns the current head position within the specified stream. If the stream is an input stream, this position is the position of the read head, whereas for an output stream it is the position of the write head. The head position can later be used to restore the stream to a given state by calling file::seek(). (See below.)

On io-streams the read and write heads are actually the same. Due to implementation issues, read operations invalidate the write position and vice versa. Using an invalid read or write head leads to undefined behavior, so a tell/seek operation must be performed between alternating read and write operations on an io-stream.

Due to the error-prone nature of such read/write operations, the use of io-streams is generally discouraged for purposes other than networking, where seek and tell are not supported anyway.

Return Value

The current head position within the specified stream.

Throws

error::stdlib::file::tell.

Examples

```
   // Create a file in read/write mode:
var iofile = file::open("test.txt", "rw");
   // Write some text to the file:
file::write(iofile, "This is some text\n");
   // After a write operation the read head is invalid: var s =
   file::getline(iofile);  // undefined!
file::seek(iofile, file::beg());
   // Now we have a valid read head position:
var s = file::getline(iofile);  // ok!
   // After a read the write head is invalid: file::write(iofile, "Some
   more text\n");  // undefined!
file::seek(iofile, file::end());
   // Now we have a valid write head position:
file::write(iofile, "Some more text\n");  // ok!
```

file::seek(stream, pos)

Given a head position pos retrieved from a stream, the stream can be returned to that position with a call to file::seek(stream,pos). If the head position was retrieved from a different stream or if the stream is not seekable (for example, the standard input/output or network streams) the behavior of file::seek() is undefined and likely causes a run-time error.

See file::tell() above for a discussion of the use of seek and tell in conjunction with io-streams.

Throws

error::stdlib::file::seek.

file::seek(stream, offs, rpos)

Where file::seek(stream,pos) sets the head position to a specific position, file::seek(stream, offs, rpos) can be used to change the head position relatively to predefined position rpos. The offset offs must be a positive or negative integer that specifies the relative position of the head with respect to this predefined position. The possible values of rpos are the current head position, the beginning of the stream or the end of the stream. Generic tokens that represent each of these three positions can be retrieved by calls to file::cur(), file::beg() and file::end() respectively. These tokens are unique values unlike all other M-Script values. It therefore does not make sense to compare them to other values, including stream positions that are retrieved with file::get(), nor to use them in arithmetic expressions. Attempts to move the head to an invalid position, such as before the beginning of the stream is undefined and most likely causes a run-time error. Likewise the behavior is undefined if the stream is not seekable.

See file::tell() above for a discussion of the use of seek and tell in conjunction with io-streams.

Throws

error::stdlib::file.

Examples

file::seek(str,1,file::cur()) moves the head position of the stream str one character ahead from its current position. If the current head position within the stream is file::end(), the result is undefined.

file::seek(str,-1,file::cur())moves the head position of the stream str one character back from its current position. If the current head position within the stream is file::beg() the result is undefined.

file::seek(str,5,file::beg()) moves the head position of the stream str to five characters after the beginning of the stream. If the stream contains fewer than five characters of data the result is undefined.

file::seek(str,-5,file::end()) moves the head position of the stream str to five characters before the end of the stream. If the stream contains fewer than five characters of data the result is undefined.

file::cur(), file::beg() and file::end()

These functions return generic tokens that represent the three absolute head positions that exist in any seekable stream: the current head position, the beginning of the stream, and the end of the stream. The tokens can be used as values for rpos in calls to file::seek(stream,offs,rpos) to make relative changes to the position of the read or write head of the stream.

file::basename(filename [, suffix])

Return Value

Strips directory specification from string filename. If string suffix is supplied and matches the end of filename, it is stripped away, too. Note that dots (.) are not treated specially. Both forward (/) and backward (\ ) slashes are treated as path delimiters. The directory is taken to be any character up to and including the last slash (/ or \ ).

Examples

```
var f1 = "/a/b\c/d/hulla.hopsa.exe.jpg";
var f2 = "afsd0934#¤\abe";
file::basename(f1);      // "hulla.hopsa.exe.jpg"
file::basename(f1, "g"); // "hulla.hopsa.exe.jp"
file::basename(f2);      // "abe"
```

Regular expressions

Regular expressions are a form of pattern-matching that is often used in text processing; many users are familiar with the Unix utilities grep, sed, and awk, and the programming language perl, each of which makes extensive use of regular expressions.

The package regex is an M-Script interface to the Regex++ library from the Boost library collection4.

Regular expressions are a powerful tool for text manipulation, and an in-depth description of the possibilities lies beyond the scope of this document. The discussion here is therefore limited to the M-Script-specific aspects of the regex package.

Context

The permitted context for functions and procedures in the regex:: package is All.

Defining and Using Regular Expressions

M-Script supports regular expressions directly in its syntax. A regular expression in M-Script is given by the grammar:

```
regexPattern ::= / pattern / [modifier]*
```

For the pattern, you can use the extended syntax used by perl, awk, and egrep. In addition, the following optional modifiers are supported.

i	Pattern matching is performed case-insensitive. The default is case sensitive pattern matching.
g	Pattern matching is done globally. By default pattern matching is only performed until the first match is found.

As a short example of such a pattern, the following adapts an example found in the Regex++ online documentation. The example shows a regular expression pattern for validating credit card numbers (of the form dddd-dddd-dddd-d where “d” is a digit. The four-digit groups may also be separated by spaces):

```
var creditCardNo = /(\d{4}[- ]){3}(\d)/;
```

You can find the home page of Boost at http://www.boost.org

Here \d is the perl shorthand for the locale-independent POSIX standard form [[:digit:]]. you could also simply use the range specification [0-9]. The parentheses act to group (and mark for future reference) subexpressions, and the {4} means “repeat exactly 4 times.”

Backslashes

Be careful when trying to match backslashes in the text. If you write a regular expression such as /pattern/ directly in the source code, you must escape the back-slashes once, for example, to match “a\b” you would write /a\\b/. But if you write expressions in strings you must escape the backslashes twice! This is necessary because both the M-Script parser and the pattern parser use escapes in the same way. So for instance “a\\b” is reduced by the M-Script string parser to “a\b” which is sent to the pattern parser. This in turn reads “a\b,” which is an illegal escape of “b.” To solve this you need four backslashes, as in “a\\\\b” which M-Script reduces to “a\\b” which again is recognized correctly by the pattern parser as “a\b.”

regex::create(pattern [, modifiers])

In addition to the built-in pattern syntax described previously, a regular expression pattern can also be constructed dynamically from a string representation. The regex::create function constructs such an expression from a pattern and optional modifiers.

Return Value

The newly created regular expression pattern.

Example

```
var creditCardNo = regex::create("(\\d{4}[- ]){3}(\\d)","");
```

Notice how the backslashes within the pattern string must be escaped.

regex::match(pattern, text [, matches])

This function attempts to match the entire input text against a regular expression pattern. An optional array (matches) can be passed as a third argument. Those parts of the input that match the subexpressions within the pattern are then written into the elements of this array.

Return Value

True if the pattern matches, false otherwise.

Example

This example tries to match a string against the credit card pattern shown previously, and gets each digit-group in the credit card number written into the matches array. Using the pattern as-is does not do this:

```
var creditCardNo = /(\d{4}[- ]){3}(\d)/;
var matches = [];
regex::match(creditCardNo, "1234-2341-3412-4", matches);
-->
   matches = ["3412-","4"]
```

There are two problems here; first of all you only get the last of the four-digit groups written into the array; second, you get the delimiter written into the array too. The first problem is caused by the fact that each subexpression only results in one entry in the array, no matter how many times the subexpression is matched. The second problem is that the delimiter is a part of the subexpression.

Both problems can be solved by a simple rewrite of the pattern:

```
var creditCardNo = /(\d{4})[- ](\d{4})[- ](\d{4})[- ](\d)/;
var matches = [];
regex::match(creditCardNo, "1234-2341-3412-4", matches);
-->
   matches = ["1234","2341","3412","4"]
```

regex::find(pattern, text [, matches])

This function attempts to find a substring in the input text that matches the specified pattern. An optional array (matches) can be passed as a third argument. Those parts of the matched substring that match the subexpressions within the pattern are then written into the elements of this array.

For simple patterns, strpos is faster than regex::find.

Return Value

The index of the first character in the substring matching the specified pattern, null if no substring matches the pattern.

regex::split(pattern, text)

This function attempts to split up the input text by using the specified pattern as delimiter. If the pattern itself does not contain subexpressions, then the return value is those parts of the input that do not match the pattern. If the pattern does contain subexpressions, the substrings that are matched by the subexpressions are returned.

For simple delimiters, strsplit is faster than regex::split.

Return Value

The return value is an array of substrings of the input.

regex::replace(pattern, text, replace)

This function scans through the input text for substrings that match the specified pattern. These substrings are then replaced with the specified replace string.

Return Value

The string with the matching substrings replaced.

Example

```
regex::replace(/ang|fer/g, "M-Script language reference", "###");
-->
   "M-Script l###uage re###ence"
```

The replace string may include the following expressions.

Expression	Description
$`	Expands to all the text from beginning of the input string to the beginning of the match.
$’	Expands to all the text from the end of the match to the end of the input string.
$0	Expands to all of the current match.
$N	Expands to the text that matched subexpression N.
(...)	Grouping of subexpressions.
?NE1:E2	Conditional replacement.
\$	The literal $.
\(	The literal (.
\)	The literal).
\?	The literal ?.
\:	The literal :.
\\	The literal \

Remember to escape the backslash when using it in the replacement string, for example, write \\$ to insert a literal $ in the text. To insert a backslash you need to use \\\\.

Conditional Replacement

The expression ?NE1:E2 expands to either E1 or E2, depending on whether the n-th group participated in the match or not. If the n-th group did participate, E1 is used; otherwise, E2 is used.

Example

```
regex::replace(/(abc)|(def)/g,
                "1234abcdef5678",
                "#(?1X:Y)#")
```

The resulting string is "1234#X##Y#5678".

Regex Patterns

Literal Characters

The alphabetic characters and digits match themselves literally as expected, but it is also possible to match certain other characters such as newlines and braces, which are otherwise used for special purposes. The following is a list of all such literals.

Character	Matches
\.	A literal .
*	A literal *
\?	A literal ?
\+	A literal +
\{	A literal {
\}	A literal }
\[	A literal [
\]	A literal]
\(	A literal (
\)	A literal)
\^	A literal ^
\$	A literal $
\\	A literal \
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\0dd	An octal character code, where dd is one or more octal digits.

Character	Matches
\xXX	A hexadecimal character code, where XX is one or more hexadecimal digits.

Wildcard

The dot '.' matches any single character.

Character Classes

A character class is a set of characters enclosed in square brackets that match exactly one of the characters in the set. Thus, the expression /[abc]/ matches either a single "a," a single "b," or a single "c." Negated character classes can be defined by placing a caret "^" as the first character inside the class. This means the expression /[^abc]/ matches any single character except "a," "b," and "c."

A range of characters can be specified using a hyphen in the set. Thus, the expression /[a-z0-9]/ matches any lowercase letter or a single digit.

Special predefined character classes can be named using [:classname:] in the character set; for example [[:space:]] is the set of all whitespace characters. The following is a list of all class names.

Class name	Description
alnum	Any alpha numeric character.
alpha	Any alphabetical character a-z and A-Z.
blank	Space or tab.
digit	Any digit 0-9.
lower	Any lowercase character a-z.
punct	Any punctuation character.
space	Any whitespace character.
Upper	Any upper case character A-Z.
xdigit	Any hexadecimal digit character, 0-9, a-f and A-F.
word	Any word character - all alphanumeric characters plus the underscore.

The following shorthands can also be used for naming character classes.

Shorthand	Class name
\w	[:word:]
\s	[:space:]

Shorthand	Class name
\d	[:digit:]
\l	[:lower:]
\u	[:upper:]

To include a literal "-" in a set declaration, make it the first character after the opening "[" or "[^", the endpoint of a range, or precede with an escape character as in "[\-]". To include a literal "[" or "]" or "^" in a set, make them the endpoint of a range, or precede with an escape character.

Repetitions

A repetition is an expression that is repeated an arbitrary number of times. An expression followed by "*" can be repeated any number of times including zero. An expression followed by "+" can be repeated any number of times, but at least once. Thus, /[a-z]*/ matches a sequence of lowercase characters (including zero characters) and /[0-9]+/ matches one or more digits.

When it is necessary to specify the minimum and maximum number of repetitions explicitly, the bounds operator "{}" may be used; thus a{2} is the letter "a" repeated exactly twice, a{2,4} represents the letter "a" repeated between 2 and 4 times, and a{2,} represents the letter "a" repeated at least twice with no upper limit. Note that there must be no white space inside the {}, and there is no upper limit on the values of the lower and upper bounds.

Character	Description
{n,m}	Match at least n times but no more than m times.
{n,}	Match at least n times.
{n}	Match exactly n times.
?	Match zero or one time. Equivalent to {0,1}.
+	Match one or more times. Equivalent to {1,}.
*	Match zero or more times. Equivalent to {0,}.

Alternation

The | character separates alternative expressions. For example, /ab|cd|ef/ matches either "ab", "cd", or "ef."

Grouping

Subexpressions may be grouped using parentheses (). For example, the expression /(a|b)*/ matches zero or more occurrences of "a" or "b," for instance "aabbaba" or "bbbbab"—as opposed to /a|b*/ which matches a single "a" or zero or more "b"s.

Back References

A back reference is a reference to a previous subexpression that has already been matched. The reference is to what the subexpression matched, not to the expression itself. A back reference

consists of the escape character \ followed by a digit “1” to “9”. \1 refers to the first subexpression, \2 to the second, and so forth. For example, the expression (.*)\1 matches any string that is repeated about its mid-point for example “abcabc” or “xyzxyz”. A back reference to a subexpression that did not participate in any match matches the null string.

Match Position

The caret ^ matches the beginning of a line, and $ matches the end of a line. Thus /xyz$/ can be used to find a string that ends with “xyz.” The match position is not relevant for regex::match() which always matches from the beginning of a string to the end.

Miscellaneous Escape Sequences

Character	Description
\w	Equivalent to [[:word:]]
\W	Equivalent to [^[:word:]]
\s	Equivalent to [[:space:]]
\S	Equivalent to [^[:space:]]
\d	Equivalent to [[:digit:]]
\D	Equivalent to [^[:digit:]]
\l	Equivalent to [[:lower:]]
\L	Equivalent to [^[:lower:]]
\u	Equivalent to [[:upper:]]
\U	Equivalent to [^[:upper:]]

Internet Connections

M-Script includes a few functions for accessing various internet services, such as web and mail servers. First of all it is possible to make a connection to any server on the internet using net::connect. This function takes a server name and port number and returns a data stream, which may be used in most of the file and streams functions. For accessing a web server, you can use net::openurl which initiates an HTTP GET or POST operation on a web server.

Both of these functions return a data stream that is closed when the stream handle goes out of scope (which also happens when a null value is assigned to it). For this reason, there is no net::close function.

Finally, there is net::parseurl, which parses an URL and decomposes it into its basic elements (protocol, host, port, and so on).

Context

The permitted context for functions and procedures in the net:: package is All.

net::connect(hostName, portNumber)

This function makes an internet connection to the specified server/port combination. The host name is string that identifies the server, either using a host name such as www.maconomy.com or an IP number such as 192.38.244.201. The port number is a positive integer.

One problem with the use of net connections relates to the internal buffering that is done by the operating system. Since most network protocols work by first sending a request and then waiting for the answer, you must make sure that data is sent before you listen for the result—if you do not do that, the program would wait for a server response that never comes, because no request was sent. For this reason, a program must call io::flush before any read operation.

Return Value

An object with the following properties.

Property	Type	Description
ok	bool	Indicates success or failure.
error	string	Error message in case of failure, otherwise null.
data	stream	Input/output stream for sending and receiving data.

If exceptions are enabled, an exception of type error::stdlib::net that contains a standard message object is thrown instead.

Example

```
print("Creating socket connection to 'www.maconomy.com'\n");
var net = net::connect("www.maconomy.com", 80);
var data = net.data; print("Sending HTTP request\n"); file::print(data, "GET /
HTTP/1.0\r\n\r\n");
   // Now make sure data is sent before read io::flush(data);
print("Receiving answer\n");
var line = file::getline(data);
print("^1\n", line);
```

net::openurl(url [, args])

This function accepts a URL (string) and tries to fetch whatever data it points to. Currently, only HTTP 1.1 requests are supported. The HTTP 1.1 implementation does not support persistent connections and chunked data.

Return valueAn object with the following properties.

Property	Type	Description
ok	bool	Indicates success or failure.
error	string	Error message in case of failure, otherwise null.
data	stream	Input/output stream for sending and receiving data. The actual size of the data is protocol dependent, but for the

Property	Type	Description
		HTTP protocol it may be found in the http.contentSize property. For the HTTP protocol it furthermore makes no sense to send any data to the server—only reading should be done.
http	object	Various information returned from the web server. This field depends on the protocol specified in the URL, but since HTTP is the only protocol currently supported, this always exists. Future versions of M-Script may implement other protocols (for instance FTP or telnet).

The http return object contains the following properties (see to the HTTP protocol specification for an in-depth description of the fields).

Property	Type	Description
responseCode	int	The response code returned by the server.
responseMessage	string	The response message returned by the server.
contentType	string	Document content type, such as “text/html.”
contentLength	int	Size of the returned data.
headers	object	An object that contains all of the return headers. Each header has its own property (with the same name as the header) which stores the value of the header.

The URL consists of the following elements:

```
protocol://host:port/path
```

All elements are optional and default to the http protocol, localhost, port 80, and the empty path. Examples of valid URLs could be:

```
http://www.maconomy.com:80/root/index.html
//www.maconomy.com/root/index.html www.maconomy.com
```

The optional parameter args is an object that contains additional arguments. The following additional arguments are supported for the HTTP protocol.

Property	Type	Description
postData	data	Send the specified data to the remote host as POST data. The data can be a string or an output stream.

extraHeaders	array	Array of additional HTTP headers for the request. The array must be of objects with the properties “name” and “value.” Example:
		`{`
		`  extraHeaders:`
		`  [`

Property	Type	Description
		{ name: "SOAPAction", value: "add" }, ...] }
contentType	string	Content type specification. For instance, text/xml.
version	string	The HTTP protocol version. Versions 1.0 and 1.1 are supported. Version 1.1 ("1.1") is default.

Example

```
print("Make url connection to 'www.maconomy.com'\n");
var url = net::openurl("www.maconomy.com");
print("Get response\n");
var line = file::getline(url.data);
print("^1\n", line);
print("Returned data\n");
dumpvalue(url);
```

Output

```
Make url connection to 'www.maconomy.com'
Get response
<html>
Returned data
{
    ok:true,
    error:null,
    data:socket,
    http:{
        responseCode:200,
        responseMessage:"OK",
        contentType:"text/html",
        contentLength:0,
        documentDate:"Fri, 23 Nov 2001 14:16:42 GMT",
        headers:{
            server:"Microsoft-IIS/4.0",
            date:"Fri, 23 Nov 2001 14:16:42 GMT",
            content-type:"text/html",
            set-cookie:"ASPSESSIONX=Y; path=/",
            cache-control:"private"
        }
    }
}
```

net::parseurl(url)

This function parses an URL (string) and returns its elements. The return value is an object with the following properties.

Property	Type	Description
ok	bool	Indicates success or failure.
error	string	Error message in case of failure, otherwise null
protocol	string	The protocol.
host	string	The host name.
port	int	The port number.
path	string	The path.

Example

```
var urlData = net::parseurl("www.maconomy.com/index.html");
dumpvalue(urlData);
```

Output

```
{
   ok:true,
   error:null,
   protocol:"http",
   host:"www.maconomy.com",
   port:80,
   path:"/index.html"
}
```

XML Interface

M-Script offers an interface for reading and writing data formatted with XML. This functionality is found in the package xml, which contains the following functions:

```
xml::DOMParse(data)
xml::DOM2Parse(data)
xml::SAXParse(data, object)
xml::SAX2Parse(data, object)
xml::DOMPrint([ostream,] array)
xml::dumpvalue([ostream,] value)
xml::readvalue(data)
xml::dumpDTD([ostream])
xml::escape(data)
```

Context

The permitted context for functions and procedures in the xml:: package is All.

The next sections discuss the possibilities in this package.

Parsing XML-Data

XML-data can be read directly from a string, or from an M-Script input stream. There are two ways to parse XML-formatted data:

- xml::DOMParse(data [, options])
- xml::DOM2Parse(data [, options])

With DOMParse() and DOM2Parse() the data is read into one big data-structure that represents all of the XML-data. This data-structure is an array that contains the top-level nodes in the parse-tree. Each node in this array is a DOM object.

The optional argument options must be an object that contains zero or more of the following properties.

Name	Type	Description
whitespace	string	Specify how white space should be handled. Possible values are: ▪ **seep** — Keep all white space (default). ▪ **skip** — Skip strings that consist solely of white space. This eliminates all indentation strings between tags ▪ **trim** — Trim leading and trailing white space on all strings. In addition to eliminating indentation strings between tags (like skip) this option also ensures that each string within tags starts at the first, and ends at the last non-blank character.

xml::SAXParse(data, object)

xml::SAX2Parse(data, object)

With SAXParse() the data is read using a callback interface. This means that a callback object that contains functions that correspond to different events while parsing the data is handed to the XML parser. These functions are then called by the parser when the corresponding parse-event occurs.

Which of these parsers you should use is a trade-off between resources and flexibility. With the DOM parser, you get a data structure that represents the entire document. This allows for very flexible non-linear searching and manipulation of the data, but can also be very memory-consuming on large data-sets. The size of the DOM data structure can be as much as ten times the size of the original XML-document. If memory constraints are not an issue, the DOM parser is an obvious choice.

If memory is an issue, or you only need fast access to limited parts of the XML data, the SAX parser may be the better choice. It enables you to quickly discard the input that you do not need, extracting only the data relevant to you. The drawbacks of the SAX-parser is that parsing is a read only operation, and if you need context-information you must maintain it manually.

The M-Script DOM Format

The return value from a call to DOMParse() and DOM2Parse() is an array value that contains the resulting data structure, and a status object and an object error that contains status and error information. Note that as of M-Script 6 all standard libraries by default throw exceptions on errors so the status/error properties are only maintained for full backward compatibility.

If exceptions are disabled, and the parse failed for some reason, content is null, while error becomes an object that contains three fields: An error message is placed in the field message, while the line and row position of the error are placed in lineno and column respectively. When exceptions are enabled, the error object is instead thrown as an exception of type error::stdlib::xml::parse. If no error occurred, error is null.

The data-structure itself is an array where each element is either a text string, corresponding to characters that are read outside a data element, or an object that represents structured data. All such objects contain a string type that indicates the type of the data element. Possible values of type are pi for processing instructions, and tag for data elements. Processing instructions are of the form <?target ...?>, while data elements have the syntax <name [attrib]*>...</name>.

An object that represents a processing instruction contains two additional fields: target contains the target value of the processing instruction, while name contains the rest of the instruction up to, but not including, the ?>.

Example

The processing instruction <?foo val=bar?> results in an object that contains the following: {type: "pi", target: "foo", name: "val=bar"}.

An object that represents a data element contains three fields in addition to type: name is a string that contains the name of the field; attrib is an object that contains the key-value pairs of all attributes in the tag; while content is an array with all data contained within the data element. Therefore content has the same type as the value field in the outermost status object.

The DOM2Parse function returns more detailed information about the XML and includes the namespace values. The tag name is now an object with three properties (name, qname, and uri) and the same goes for the attribute name, which is also an object with the same properties.

The DOM2 name property contains the unqualified tag or attribute name (that is the name without any namespace qualifier) and the DOM2 qname property contains the same name, but with namespace qualifiers. The uri property contains the expanded namespace URI for the tag or attribute name.

Example

Consider the following XML-expression:

```
<employee company="Acme Inc.">
   <name>John Doe</name>
   <phoneno>555 123</phoneno>
</employee>
```

The resulting DOM object for DOMParse() becomes:

```
[
   {
      ok:true,
      content:[
         {
            type:"tag", name:"employee”,
```

```
            attrib:{
                company:"Acme Inc."
            },
            content:[
                "\n\t",
                {
                    type:"tag", name:"name", attrib:{},
                    content:["John Doe"]
                },
                "\n\t",
                {
                    type:"tag", name:"phoneno", attrib:{},
                    content:["555 123"]
                },
                "\n"
            ]
        }
    ],
    error:null,
}
]
```

The following XML code illustrates the return value of DOM2Parse():

```
<employee company="Acme Inc."
          xmlns="maconomy.com"
          xmlns:info="emplyeeInfo.com">
   <info:name>John Doe</info:name>
   <info:phoneno>555 123</info:phoneno>
   <manager>Sue Summers</manager>
</employee>
```

The employee XML contains two employee values from the “employeeInfo.com” namespace and one value from the “maconomy.com” namespace. The resulting DOM representation is:

```
{
   ok:true,
   error:null,
   content:[
      {
         type:"tag",
         name:{
            name:"employee",
            qname:"employee",
            uri:"maconomy.com"
         },
         attrib:{
            company:{
               value:"Acme Inc.",
               name:"company",
```

```
            qname:"company",
            uri:""
        },
        xmlns:{
            value:"maconomy.com",
            name:"xmlns",
            qname:"xmlns",
            uri:""
        },
        info:{
            value:"emplyeeInfo.com",
            name:"info",
            qname:"xmlns:info",
            uri:"http://www.w3.org/2000/xmlns/"
        }
    },
    content:[
        {
            type:"tag",
            name:{
                name:"name",
                qname:"info:name",
                uri:"emplyeeInfo.com"
            },
            attrib:{},
            content:[ "John Doe" ]
        },
        {
            type:"tag",
            name:{
                name:"phoneno",
                qname:"info:phoneno",
                uri:"emplyeeInfo.com"
            },
            attrib:{},
            content:[ "555 123" ]
        },
        {
            type:"tag",
            name:{
                name:"manager",
                qname:"manager",
                uri:"maconomy.com"
            },
            attrib:{},
            content:[ "Sue Summers" ]
        }
    ]
```

```
        }
    ]
}
```

The M-Script SAX Interface

The callback object that is passed to SAXParse() can contain any subset of the following methods. If a method is omitted, a default action, usually a no-op, for the corresponding event is executed.

startElement: procedure(name, attrib)

This method is called when an element opening tag is read. The parameter name is the name of the tag, while attrib is an object containing the key-value pairs of all attributes in the tag.

Example

The opening tag <entry a="10" b="foo"> results in a call to startElement("entry", {a:"10", b:"foo"}).

endElement: procedure(name)

This method is called when an element closing tag is read.

Example

The closing tag </entry> results in a call to endElement("entry").

characters: procedure(char)

This method is called to pass data read between the tags. The SAX-standard does not guarantee that all data is passed in one call, so several calls to this method might occur between the calls to beginElement() or endElement(). All non-tag data is reported through calls to characters(), even white-space characters between data elements.

Example

The data element <name>John Doe</name> could lead to the following call sequence:

```
beginElement("name", {})
characters("Joh")
characters("n Doe")
endElement("name")
```

processingInstruction: procedure(target, data)

In addition to data elements, XML can also contain processing instructions. These are of the form <?target ...?> and are used for a wide range of purposes. This method is called whenever such a processing instruction is read.

Example

The processing instruction <?foo val=bar?> results in a call to processingInstruction("foo", "val=bar").

In addition to these methods, three callback methods for error handling are defined. These all take a single parameter that is an error object that describes the nature of the error. This object consists of three fields: message, lineno, and column, where message is a text description of the error, and (lineno,column) is the position in the input where the error occurred.

error: procedure(errorObj)

This method is called if the XML input is well-formed but not valid. Such an error is considered non-fatal, so the parser continues after the call to error().

fatalError: procedure(errorObj)

This method is called when the XML input is not well-formed. If fatalError() is not defined, the XML parser throws a run-time exception. In most cases, a parser stops after the first fatal error that it finds.

warning: procedure(errorObj)

The method is called if the XML input is correct, but there is some other condition that the parser considers useful to report. Such conditions are non-fatal and the parser continues when warning() returns.

The M-Script SAX2 Interface

The callback object that is passed to SAX2Parse() can contain any subset of the methods that SAXParse() supports. If a method is omitted, a default action, usually a no-op, for the corresponding event is executed. The only difference between the SAX1 and SAX2 interface is that SAX2 passes more information to the following event handlers.

startElement: procedure(name, attrib)

This method is called when an element opening tag is read. The parameter name is the name of the tag represented as an object with the name, qname, and uri properties. The attrib parameter is an object that contains the key-value pairs of all attributes in the tag. Each attribute value is an object with the name, qname, and uri properties, as well as a value property with the actual attribute value.

Example

The opening tag <ns:entry a="10"> results in a call to startElement with the following parameters:

```
name:   { name:"entry", qname:"ns:name", uri:"namespace.com"}
attrib: { a:{value:"10", name:"a", qname:"ns:a", uri:"namespace.com"}
    })
```

endElement: procedure(name)

This method is called when an element closing tag is read. The name parameter is an object with the name, qname and uri properties.

Example

The closing tag </entry> results in a call to endElement({name:"entry", qname:"ns:entry",uri:"namespace.com"}).

Printing XML

xml::DOMPrint([ostream,] array)

The procedure xml::DOMPrint() converts a M-Script DOM object back to its textual XML representation and prints it on an output stream. If an output stream is not passed as the first argument, the standard output stream is used. The format of the DOM object corresponds to the value field in the status object returned by xml::DOMParse().

xml::dumpvalue([ostream,] value)

Most M-Script types can be converted to an XML representation. For each of the supported M-Script types, the procedure xml::dumpvalue() writes the XML representation on an output stream. If an output stream is not passed as the first argument, the standard output stream is used. The range of types supported by xml::dumpvalue() is identical to that of the built-in dumpvalue() procedure. When an M-Script value is converted to XML, it is placed inside a tag called value with the namespace specifier xmlns="http://maconomy.com/mscript."

xml::readvalue(data)

An M-Script value encoded in XML can be read back with the function xml::readvalue(). The data is either a string or an input stream, and readvalue() attempts to parse an XML representation of an M-Script value from the start of this data.

Return Value

The return value is a status object consisting of three fields.

ok	bool	A flag indicating if the read was successful.
value	any	The M-Script value encoded in the data.
error	object	An error object containing a string message, in addition to the line number lineno and column number column of where the read failed.

As of M-Script 6.0, errors are thrown as exceptions if standard library exception is enabled. Specifically syntax errors in the input cause an exception of type error::stdlib::xml::syntax, and the value of this exception is an object with the messages, lineno, and column properties.

xml::dumpDTD([ostream])

The function xml::dumpDTD() writes the DTD (Document Type Definition) for XML-encoded M-Script values on an output stream. If no argument is specified, the standard output stream is used. The DTD is useful if the receiver of XML-encoded M-Script values wants to validate that the XML is well-formed.

xml::escape(data)

The function xml::escape() takes text input from either a string or an input stream and converts it to a format that is suitable for XML representation. Special XML characters, such as angular brackets (<, >) and quotation marks that have special meaning in XML, are replaced by their standard XML encoding, such that '<' becomes "<" and so on.

The functions xml::dumpvalue() and xml::readvalue() handle these conversions automatically, so when using them there is no need to pay special attention to M-Script strings.

Return Value

The XML-escaped string.

Using XPath

XPath is a standard query language for retrieving elements of an XML structure, and it includes all sorts of expressions for manipulating and extracting whole subsets of data, as well allowing a user to use general relational expressions for pinpointing data—see, for instance, “Professional XML” from Wrox Press Ltd. for a more detailed description.

xml::xpath(array, string)

With xml::xpath(), M-Script implements a tiny subset of the XPath functionality—namely retrieving a single named tag or attribute.

Return Value

An array that holds the object that corresponds to the specified tag, or the text string of the attribute. The reason for storing the object in an array is that future extensions to the XPath implementation might return multiple tags.

The XPath expression is constructed more or less as a path specification of a file in a Windows (or Unix) file system. It consists of tag names that are separated by slashes and indexed by brackets, and references to attributes using the @ operator.

The return value depends on the expression. If it refers to an attribute, the attribute value is returned as a string, and if it refers to a tag, the object that defines the tag is returned.

Example

```
var xml = xml::DOMParse('XML');
   <inventory name="local">
      <item type="book">M-Script for dummies</item>
      <item type="book">M-Script for experts</item>
   </inventory>
XML
   // Get first item
xml::xpath(xml.value, "inventory/item");
   // Get second item
xml::xpath(xml.value, "inventory/item[2]");
   // Get attribute 'name' of inventory
xml::xpath(xml.value, "inventory/@name");
   // Get attribute 'type' of second item
xml::xpath(xml.value, "inventory/item[2]/@type");
```

The return values of the above examples are:

```
   // Path: inventory/item
[
   {
      type:"tag",
      name:"item",
      attrib:{
         type:"book"
      },
      content:[
         "M-Script for dummies"
      ]
```

```
    }
]
    // Path: inventory/item[2]
[
    {
        type:"tag",
        name:"item",
        attrib:{
            type:"book"
        },
        content:[
            "M-Script for experts"
        ]
    }
]
    // Path: inventory/@name
"local"
    // Path: inventory/item[2]/@type
"book"
```

Performance Tips

Prefer attributes for simple data instead of marked-up text. This saves you some memory when working with the DOM. Instead of using:

```
<employee>
    <name>John</name>
</employee>
```

You should use:

```
<employee name="John"/>
```

Use the SAX interface for large XML data sets.

The DOM interface is good for small amounts of data and makes it simple to access what has been parsed, but it does, unfortunately, consume a lot of memory. Therefore, you should prefer the SAX interface if you expect to be working with large amounts of data.

Technical issues

M-Script offers the xml package on the following platforms: Windows NT, IBM AIX, SUN Solaris, and Linux. It is Deltek's goal to offer XML parsing on all platforms that are supported by M-Script.

The XML parser that is used by the M-Script xml package is called Xerces, and is a project of the Apache Software Foundation5. It features full Unicode support and implements W3C's XML and DOM version 1 and 2 standards as well as the (de facto) SAX version 2 standard. Xerces is freely distributed under the license found in the Appendix.

You can find the home page of Xerces at http://xml.apache.org

Session Handling

One of the problems with HTML pages is the stateless nature of the HTTP protocol; every new execution of a CGI program such as the M-Script interpreter knows nothing of any prior pages the user might have seen, and it is not possible for one M- Script program to access variables used in another M-Script program.

This problem can be overcome in two different ways—either by explicitly using cookies on the client side, or by introducing a notion of sessions that allows certain variables to exist across multiple calls to the CGI program. In M-Script the last has been chosen and currently implemented by the passing of a session identifier in all URL links. Sessions could also have been implemented with cookies, but this was for various reasons discarded.

To set up a new session, the M-Script programmer must call the procedure newsession. This creates an internal storage for session variables and a session identifier to be used in all subsequent calls to the M-Script interpreter. Each of these M- Script invocations will then be running in the same session and have access to the same session variables. The session identifier must be passed to the interpreter via the special URL query variable named sessionid. If this query variable is set and a corresponding session storage exists on the server side, then the session variables in the session can be accessed.

Since the session identifier is passed via the URL, there must also exist some way of obtaining the identifier. This is done with a call to one of the functions currentsession or link. The first one returns a string containing exactly the session identifier, whereas the second takes a URL and adds the session identifier to it:

```
newsession();
print("<a href=\"^1\">click me!</a>", link("nextpage.html") );
```

The output of this program looks like the following:

```
<a href="nextpage.html?sessionid=N8Vl1WjI0E">click me!</a>
```

A session and its associated storage can be deleted with a call to the function deletesession.

Session Locking

The session file is locked by the M-Script program the first time a session variable is used in the script. This is done to avoid multiple M-Script programs overwriting each other's data. A program that tries to access a locked session file is put on hold until the lock is removed. On Unix, the program waits as long as necessary, and on Windows the program waits for a specific time—by default 30 seconds. The timeout value can be specified with the sessiontimeout setting in the initialization file.

If a session lock cannot be obtained in time, the system gives up and stops the execution of the program.

Session Variables

Now that you know how to create and stay in a session, we can see how the session can be utilized. To create a session variable, you use the keyword session. The following is a full M-Script that creates a session, defines a session variable, and then adds a link that increases this variable every time that it is clicked.

```
#version 15
if (!hassession())
    newsession();
```

```
session var x = 0;

++x;

print("x = ^1<br>\n", x);
print("<a href=\"^1\">Increase x</a>", link("scrap.ms") );
```

The variable initializer is used only when the variable is created—that is, at the first encountered declaration of the session variable in each session.

To use a session variable x, it must be declared like "session var x;" in all scripts where it will be used.

Some M-Script types cannot be stored in a session. The following list shows which types can and which cannot be stored in a session.

Type	Can be stored
Null	yes
Boolean	yes
Integer	yes
Real	yes
Amount	yes
Date	yes
Time	yes
String	yes
Object	yes
Array	yes
Server handle	yes
Package handle	yes
Regular expression	yes
Function	no
EnumRef	no
Enum	no

Sessions in Packages

A session is shared between all packages in a program. The session variables of each of these packages are set during its own initialization, so it must be ensured that a valid session is open at

the time of this initialization. As the sequence in which the initializations take place may change, it is generally not possible to rely on any given package for setting up the session.

The preferred solution to this problem is to put a guarded newsession() call in all packages that contain session variables:

```
if (!hassession())
   newsession();
```

It this way, you are sure that a valid session has been opened before the first attempt to access it.

Maconomy Interface

The main feature in M-Script is of course its seamless integration with the Maconomy application. With M-Script, it is possible to log on as a specific user, interact with Maconomy dialog windows, execute SQL select commands on the Maconomy server, and run Analyzer reports. To do so, and to sustain a virtual login across multiple calls to the M-Script interpreter, it is mandatory to run the M-script in a session.

All of the Maconomy functions exist in the M-Script module maconomy. This module name must be prepended to all Maconomy functions with the :: operator. To use the login function on the Maconomy server, you should for instance use:

```
maconomy::login(...)
```

All Maconomy functions are executed remotely on the Maconomy server via M-Script's built-in remote procedure calling. This functionality allows M-Script to execute selected server-side functions as if they were executed on the same machine as the M-Script interpreter.

To call any of the Maconomy functions, you must always make sure to be logged in using the login function.

A simple program that dumps the output of an SQL call would look like this:

```
#version 15
newsession();
maconomy::login("Administrator", "xyz");
var sql = maconomy::sql("select username from user");
dumpvalue(sql.result);
maconomy::logout();
 deletesession();
```

Each function in the Maconomy API is described in the M-Script Maconomy API Reference.

Multiple Server Connections

From version 5.0 of M-Script, it is possible to connect to several Maconomy applications simultaneously. To manage the available connections, the concept of a server handle has been introduced. Each server handle represents one server connection, and all functions in the maconomy library now accept a server handle as an optional first parameter. If no server handle is specified, the default connection is assumed, thereby ensuring full backward compatibility with previous versions of the maconomy interface.

Web-Centric Suff

Query Variables and CGI Data

M-Script automatically reads query variables and CGI data made available by a web server. HTML form data is supported using both the POST and GET transfer methods, including file transfers through multipart MIME (Multipurpose Internet Mail Extension).

Query variables in the URL received on the standard input are instantiated in an object as properties of the same name as the query variables. This object is again a property of the global query object that also includes the raw version of the query string. Files received as multipart MIME are each placed in a file object of the same name as the multipart message part it was received in.

Variable	Property	Properties	Description
query	raw		The full urlencoded query string. For backwards compatibility, this string contains a concatenation of both POST and URL data. Access to POST data through this property is considered deprecated and should be avoided since it may be removed from this property in a future version. Use rawPost instead.
	rawPost		Unparsed POST data represented as a string. This property is only useful in the case where M-Script does not parse the POST data (forms data, URL encoded or multi-part MIME data are all parsed). The rawPost property has been made available for use where the programmer wants to parse the data that M-Script does not handle separately—for instance incoming XML documents.
	values	A,b,c,...	Query string split into separate (urldecoded) properties
	files	a,b,c,...	File objects received as multipart MIME
	method		A string describing the HTTP operation. This may be one of the following three values: "POST" "GET" "unknown"
	status	ok, message	Status object indicating whether any problems were encountered while reading the CGI data

Only query variables that have valid names are stored in the query object. A valid name contains only the characters a-z, A-Z, 0-9 and _. Empty query variables (defined with “var=” in the URL) are defined in the values property as empty strings.

Each file object within the query.files object contains six fields:

- name — The name of the multipart message part through which the file was sent. Thus query.files[n].name = n.
- filename — A string containing the name of the file on the source machine.
- path — A string containing the path of the file on the source machine.
- content_type — A string containing the content type of the file, as reported by the source machine.
- size — An integer containing the size of the file in bytes.
- data — An input stream containing the received file.

URL Parameters with Namespace

It is possible to use URL parameters with nested namespaces. The parameters will be represented in M-Script in the query.value object as properties on objects named after the namespaces.

The syntax of an URL using namespaces is <url>?<param-list>, where <url> is the URL and <param-list> is a “&” delimited list of variable assignments of the form <variable>=<value>. <variable> must be enclosed in “$”s and each namespace must be followed by a dot ('.'). Namespaces can be arbitrarily nested.

Note that all variables are treated as strings. There is no formal way of specifying the types of the variables.

Example:

```
http://myurl.com?$a.b.c1=7&$a.b.c2=3&$b.c1=4
/*
query.values will contain:
{
    a:{
        b:{
            c1:"7", c2:"3"
        }
    }

        b:{
            c1:"4"
        }
    }
*/
```

URL Encoding and Decoding

It is possible to URL encode/decode, as well as escape/unescape strings.

Content-Type

It is possible to set the HTTP content type with a call to the function setcontenttype.

Error Handling

Run-time errors in M-Script can be caught and handled in a special M-script. This script is specified in the initialization file via the errorScript parameter.

If a run-time error occurs, the error handling script is executed, and three new global variables are defined by the interpreter. These are:

- error_filename — The name of the file in which the error occurred.
- error_lineno — The line number of the statement where the error occurred.
- error_message — The generated error message.

Example:

```
#version 15
print("An error occurred in file '^1' on line ^2\n", error_filename,
error_lineno);
print("The problem was: ^1\n", error_message);
print("Please contact the web administrator\n");
```

Debugging Facilities

Working with HTML, and especially WML, can be quite frustrating at times since you are unable to see the exact generated output from M-Script. It is for this reason possible to force M-Script to dump all its output to a separate file on the web server. To do so, the following line should be added to the initialization file:

```
DebugDumpFilename = /your/dump/file
```

The format of this file is quite simple:

```
******************************************************
   ****** Script name:       test.ms
   Execution started: Thu, 22 Feb 2001
   09:35:55 GMT Execution ended:   Thu, 22
   Feb 2001 09:35:55 GMT
******************************************************
<html>
<body>
Hello world
</html>
</body>
```

Another useful M-Script feature is the --ping command line option. When M-Script is invoked with this flag, it tries to connect to the Maconomy server while it at the same time printing some useful diagnostic information:

```
D:\bin>MaconomyMScript --ping
Connecting to WebDaemon on server.maconomy.dk:4101 ... ok.
Sending initial data to WebDaemon ... ok.
Receiving reply ... ok.
Checking reply ... ok.
Looking for free callback port on client ... ok.
```

```
Sending callback port (4200) to WebDaemon ... ok.
Closing connection to WebDaemon.
Waiting for callback from MaconomyServer ... ok.
Turn around time to server: 271ms.
Turn around time to server: 0ms.
Turn around time to server: 0ms.
Turn around time to server: 0ms.
Turn around time to server: 0ms.
Diagnostic data returned from server:
Maconomy server host: server(1622)
Server version: 36.00.0.0
Application: DK_8_0
Patch level: 0
Short name: dk80
Maconomy folder: dk80
Has M-Script reporting add-on: Yes
Has M-Script API add-on: Yes Has
Portal add-on: Yes
Reading mscript.ini: ok
M-Script is looking for M-Script files in: D:/inetpub/wwwroot/mscript
```

To contact a server other than the default server, a server label can be specified after the ping flag. The --pingall flag can be used to contact all servers.

Logging Facilities

In M-Script it is possible to specify various levels of logging of M-Script errors as well as server usage. First of all, it is possible to specify how much information should be logged as well as which kind of errors should be logged. This can be done in the initialization file with the ErrorLog and ErrorLogMask entries. In addition to this, it is possible to enable logging of server usage. This is done with the Log entry in the initialization file and makes M-Script log function names, parameters, and timing information of the specified functions.

The number of log entries can easily become overwhelming. With the LogInclude, LogExclude and LogOrder entries in the initialization file it is possible to specify filter rules to for example only log entries generated by a specific user or operations that took a very long time to complete.

It is possible to write directly to the log file from M-Script using the stream returned from the function io::log().

Standard Exceptions

error::stdlib

All functions in the standard library throw exceptions inherited from this one. It contains one property message in the value object of the exception.

Property	Description
message	A human readable message suitable for printing.

error::stdlib::file::<funcname>

Exception thrown by functions in the file library. The <funcName> part is the name of the function that threw the exception. It adds nothing to the parent exception type.

error::stdlib::file::copy

Exception thrown by file::copy if the source file or directory could not be found, or if permissions on a file or a drive prevent the operation from being completed. This exception adds nothing to the parent exception type.

error::stdlib::file::invalidCharacters

Exception thrown by file::open and other functions that try to access a file. It contains the following properties in addition to those from error::stdlib::file.

Property	Description
fileName	The name of the file being accessed.
invalidCharacters	A string containing the offending characters.

Error::Stdlib::Io::<Funcname>

Exception thrown by functions in the IO library. The <funcName> part is the name of the function that threw the exception. It adds nothing to the parent exception type.

Error::Stdlib::Pop3

Functions in the POP3 library throw this exception, which is inherited from error::stdlib. It adds nothing to the parent exception type.

Initialization File

When the stand-alone M-Script interpreter is started, it reads its parameters from a file named <name of executable>.I within the same directory as the executable. Thus if the name of the executable is maconomy, the parameters are found in the file maconomy.I6.

When executing M-scripts using the Maconomy server, the default location of the .I file is

```
<ApplicationHome>/MaconomyDir/Definitions/MaconomyMScript.I
```

However, you can specify a separate .I file using the server option --mscriptIFile. Some options are not available for specification in the .I file read by the Maconomy server executable (as opposed to the Maconomy M-Script executable). Those options, which all deal with CGI parameters, are marked with “Does not apply to server-side M-Script” in the descriptions following.

The parameter file consists of one text line per parameter. Each line can be of the following kind:

- A comment:

  If a line starts with either '//' or '##', it is ignored.
- A blank line:

  Blank lines are ignored.
- A section start
- A parameter line:

  A parameter line has the format <key>=<value>. Any leading or trailing blanks will be removed.

When the system looks up a key, it is case-insensitive, but it does not change the casing of the values.

The set of parameters is split into a group of CGI parameters and a group of M-Script parameters. The first group relates to the Maconomy interface, whereas the second group relates to M-Script specific configuration details.

From version 2.2 of M-Script, it is possible to split the initialization file into multiple sections that are applied to different file types. A section consists of a section start written as [filemask] followed by the parameters for that section. The file mask may contain the usual ? and * wildcards.

Example:

```
ServerIP = maconomy.home.dk
// No M-Script parsing of *.html files
[*.html]
scriptmode = plain
// *.hms files are parsed as embedded M-Script
[*.hms]
scriptmode = embedded
```

Note that although the name of an executable on the Windows platform usually has the file type “.exe” appended, the parameters should be placed in the file maconomy.I, even though the name of the executable is maconomy.exe

With the use of sections it is also possible to specify that all M-Script files in a Danish subdirectory should use one set of date and time formats, whereas the files in a US subdirectory should use another set of format specifiers.

The following parameters may not be part of a section:

- SearchPath
- SessionDir
- SessionLifeTime
- SessionCleanupInterval
- TmpDir

Required CGI Parameters

ServerIP=<IPAddress>

This is the IP address of the machine that runs the Web Daemon. Usually this is the same machine as the machine that holds the Maconomy Server, but it may be a machine anywhere on the (Inter)net. Does not apply to server-side M-Script.

Example

```
ServerIP=192.168.233.7
```

DaemonPort=<portnumber>

The port number is the TCP port number used to connect to the Web Daemon. It should be the same port number that is set up in the Web Daemon parameter file with the CGIPort command. Does not apply to server-side M-Script.

Example:

```
DaemonPort=4100
```

ListenFrom=<portnumber>

The connection between the Maconomy Server and M-Script is created by the Maconomy Server, but M-Script tells the Maconomy Server on which TCP port to connect.

Several M-Script interpreters can be running concurrently, so you cannot statically specify the TCP port to which the Maconomy Server should connect to get in touch with the actual interpreter. Instead, the interpreter allocates the next free port number equal to or larger that the ListenFrom port and communicates this port- number to the Web Daemon, which then tells the Maconomy Server to make the connection back to the interpreter on this port.

If a firewall is installed between the Maconomy server and the M-Script interpreter (the web server), the firewall should be configured to allow connections from the Maconomy server to the web server on port numbers in the range Listenfrom to Listenfrom+MaxListenPorts7+1. Does not apply to server-side M-Script.

The default value of MaxListenPorts is 10, but it can be changed—See the description of the command MaxListenPorts=<number>.

Optional CGI Parameters

connection = <name> : <host> : <port>

Adds the address of a new webdaemon to be used in the Maconomy interface. The value of this parameter is a colon separated string identifying the symbolic name of the connection, the host machine of the webdaemon, and its port number. The name is used to identify the connection in the M-Script program to which it can be referred using maconomy::getServerHandle(). Does not apply to server-side M-Script.

MaxListenPorts=<number>

See the description of the required command ListenFrom for a description of the use of MaxListenPorts. Does not apply to server-side M-Script.

By default, MaxListenPorts is set to 10. Usually this parameter should not be changed. You may want to change it for the following reasons:

- Decrease the number if you have fewer connections through a firewall available.
- Increase the number if you have many concurrent invocations of M-Script interpreters on the web server (and thus need many connections to Maconomy servers).
- Increase the number if the TCP/IP layer does not release the ports fast enough (the interpreter writes a message in its log file, if it cannot allocate a free port). If for instance the TCP/IP layer uses 0.1 second to release a port after use, 10 requests pr. second will use up all the ports (without taking the actual processing time in account).

WaitForServer=<milliseconds>

By default, the M-Script interpreter waits 30000 milliseconds (30 seconds) for answers from the Maconomy server before considering the link dead. Does not apply to server-side M-Script.

You may want to change the parameter for the following reasons:

- Decrease the time-out interval to get faster error detection.
- Increase the time-out interval if the requests to the server (in case of very high load) may take more than 30 seconds.
- Maximum timeout is 2,147,000 msec. (36 minutes).

WaitForDaemon=<milliseconds>

By default, the M-Script interpreter waits 30000 milliseconds (30 seconds) for answers from the Maconomy Daemon before considering the link dead. This should usually never be changed. Does not apply to server-side M-Script.

- Maximum timeout is 2,147,000 msec. (36 minutes).

LogFileName=<filename>

By default, any errors detected by M-Script are appended to the file <name of executable>.log, which normally would be maconomy.log. The file is placed in the same directory as the executable.

This filename can be changed with the LogFileName command

For server-side M-Script, the default location of the log file is

```
<MaconomyTmpDir>/MaconomyMScript.log
```

This path can be overwritten by the server option --mscriptLogFile.

LogInclude=<key=value>,... , LogExclude=<key=value>,...

Specify a log filter for server communication. Most server calls produce log entries which can be filtered on a number of keys. LogInclude allows you to specify rules for including log entries, while LogExclude describes rules for excluding entries.

The currently supported keys are:

- Time — The time spent on this log entry in milliseconds. You can specify either a maximum time (time<...), a minimum time (time>...) or write time=... and let the log choose the sensible default based on whether it is an inclusive or exclusive log rule.
- user — The user name under which the log entry was executed.
- script — The name of the script which caused the log entry.

Either the inclusive or the exclusive rules can have first priority when deciding whether a log entry should be logged. This is controlled by the setting of LogOrder.

LogOrder=include | exclude

Specify whether to use inclusive or exclusive logging.

- When using inclusive logging, entries will only be logged if they match an inclusive rule and do not also match an exclusive rule.
- When using exclusive logging, entries will be logged unless they match an exclusive rule and do not also match an inclusive rule.

The default setting is to use exclusive logging.

Required M-Script Parameters

Searchpath=<Searchpathlist>

The Searchpath command tells the M-Script interpreter where to look for M-Script files.

The search path consists of one or more paths. Each path is separated by semicolon ;. A path may be either relative or absolute. If the path is relative, the path starts at the default directory of the interpreter. This is usually the directory where the executable is placed.

For server-side M-Script, the default search path is:

```
<ApplicationHome>/MaconomyDir/MScripts
```

The default directory may depend on web server and/or OS.

PackageRoot=<PackageRootDir> [; <root>=<path>]*

The path to the root folder of the M-Script package hierarchy. It is also possible to specify one or more named package roots.

For server-side M-Script, the default package root is:

```
<ApplicationHome>/MaconomyDir/MScripts/PackageRoot
```

Optional M-Script Parameters

ScriptMode=[mscript | embedded | plain]

This parameter defines how exactly the loaded script should be handled. The different possibilities are described in the following table.

Name	Description
mscript	Interpret the script as M-Script.
embedded	Interpret the script as M-Script embedded in HTML.
plain	No interpretation of the script—it is simply printed as it is.

LoadScript=<scriptname>, EndScript=<scriptname>, ErrorScript=<scriptname>

These three M-Script files are executed in different predefined situations; the load script is always executed before the user specified script, the end script is always executed after execution of the user specified script, and the error script is always executed when an error occurs (except parse or syntax errors in the user defined script). If an error occurs inside the error script, a simple error message is printed, where after the interpreter aborts.

DefaultEncoding=<encoding>

The default encoding to use for input/output operations in all scripts.

Example:

```
DefaultEncoding = UTF-8
```

DefaultInputCharset=<encoding>

The default character set to use for parsing MScript source files.

Example:

```
DefaultInputCharset = ISO-8859-1
```

Boolformat=<formatstring>

The boolean format string specifies the input and output format of Boolean values. The string may contain the text Tnnn; as well as Fnnn; where nnn is the name of the true value as well as the false value. The values are terminated by semicolons.

Example:

```
BoolFormat = Tyes;Fno;
```

IntFormat=<formatstring>

The integer format string specifies the input and output format of integer values. The string may contain the symbols En, (), and -. The n value defines the thousands separator and may be any single character—a zero (0) means no separator. The () symbol means negative values should be printed in parentheses, and - means trailing minuses (instead of the usual leading minuses) should be used. The following example tells the interpreter to use a comma as thousand separator and use trailing minuses.

Example:

```
IntFormat = E,-
```

RealFormat=<formatstring>

The real format string specifies the input and output format of real values. The string may contain the symbols Enm, (), -, and .s. The n value defines the thousands separator and the m value defines the decimal separator. Both may be any single character—a zero (0) means no separator. The () symbol means negative values should be printed in parentheses, .s defines the number of decimals printed (s should be any 0..9 value) and - means trailing minuses (instead of the usual leading minuses) should be used. The following example tells the interpreter to use a comma as thou- sand separator, a period as decimal separator, and to print reals using four decimals precision.

Example:

```
RealFormat = .4E,.
```

AmountFormat=<formatstring>

The amount format string specifies the input and output format of amount values. The string may contain the symbols Enm, (), -, and .s. The n value defines the thou- sands separator and the m value defines the decimal separator. Both may be any single character—a zero (0) means no separator. The () symbol means negative values should be printed in parentheses, and - means trailing minuses (instead of the usual leading minuses) should be used. The following example tells the interpreter to use a comma as thousand separator and a period as decimal separator.

Example:

```
AmountFormat = E,.
```

DateFormat=<formatstring>

The date format string specifies the input and output format of date values. The string may contain the symbols:

- **En** — Use “n” as day/month/year separator
- **D** — Print days without leading zero
- **DD** — Print days with leading zero
- **M** — Print months without leading zero
- **MM** — Print months with leading zero
- **YY** — Print year without century
- **YYYY** — Print year with century

The sequence of the D, M, and Y elements defines the sequence in which they are printed. The following example makes M-Script print a date separated with “/” slashes, with days first (one digit), then months (two digits) and then years (all four digits).

Example:

```
DateFormat = E/DMMYYYY
```

TimeFormat=<formatstring>

The time format string specifies the input and output format of time values. The string may contain the symbols:

- **En** — Use “n” as time separator
- **H** — Print hours without leading zero
- **HH** — Print hours with leading zero

- **S** — Print seconds
- **Annn;** — Use “nnn” as AM string
- **Pnnn;** — Use “nnn” as PM string

The following example makes M-Script print a time separated with “-” dashes, include seconds, and write “AM” and “Pm.”.

Example:

```
TimeFormat = E-SAAM;PPM;
```

ErrorLog=[none | simple | full]

When M-Script encounters an error, it writes the current time and the error message to the log file. The default behavior is then to dump information about the URL the user requested which resulted in the error. By setting Errorlog to none, no extra information is written to the log file. By setting Errorlog to full, all available information about the request is written to the file. The values should be separated with a semicolon.

ErrorLogMask=[internal | load | parse | runtime | shutdown | all]

The error mask defines which errors that should be logged. The values should be separated with a semicolon.

Log=[option;option;...;option]

This entry is used to enable logging of server usage and it makes M-Script log called functions, parameters and timing information. The possible values for the Log entry are described in the following table.

Name	Description
none	No logging at all (default)
sql	Log only maconomy::sql calls.
analyzer	Log only maconomy::analyze calls.
login	Log only maconomy:: calls.
functions	Log all maconomy:: calls.
scripts	Log script execution load and time information.
serverstat	Log information about server-side timing.
query	Log information about received query variables.
postdata	Log all incoming POST data in a file with the same name as the M-Script executable appended with a .postdata.log.
cleanup	Log information about session cleanup activities.
localization	Log information about processing of localization dictionaries.

Name	Description
GUI_windowStatus	Log information about opening and closing of GUI windows.
GUI_warnings	Log extra information about the GUI.
client	Combines script, query and localization.
server	Combines sql, analyze, login, functions, and serverstat.
session	Log information about the creation and deletion of sessions.
GUI	Combines GUI_windowStatus and GUI_warnings.
all	Log all of the above.
performance	Log information about how various parts of M-Script perform (highly verbose!).
tab	Make log entries in a tabular format. See below for further info

The values should be separated with a semicolon. These values may be changed at run time by the setlogmask function.

The tabular data are printed in different formats. One for server-side function calls, one for script start, one for script end, and one for errors. Each tabular line of data begins with the string TAB followed by a semicolon and then the actual log data. Each entry consists of one or more semicolon separated fields, depending on the format. The fifth of these fields denotes the format of the line and may be one of the following.

Tag	Description
CALL	Server-side function call
SCRIPTSTART	Script start
SCRIPTEND	Script end
MAINEND	End of all scripts (including start and end scripts)
ERROR	Errors

The actual fields are listed in the following table.

CALL fields	
1	"TAB"
2	Date
3	Time

CALL fields	
4	Process identifier
5	Format
6	Script name
7	Session identifier
8	Transaction identifier
9	Client address
10	User name
11	Client call time. This is the time elapsed on the client while it waits for the server to answer.
12	Server call time. This is the time elapsed from the time when the client call is received on the server and until data is sent back. The difference between client call time and server call time corresponds roughly to the network delay and over-head involved with the M-Script network protocol.
13	Server transaction time. This is the time spent on the server waiting for one or more database calls.
14	Number of bytes sent from M-Script to server
15	Number of bytes sent from server to M-Script
16	Command

SCRIPTSTART fields	
1	"TAB"
2	Date
3	Time
4	Process identifier
5	Format
6	Empty
7	Empty
8	Empty

SCRIPTSTART fields	
9	Client address
10	Script name

SCRIPTEND and MAINEND fields	
1	"TAB"
2	Date
3	Time
4	Process identifier
5	Format
6	Script name
7	Empty
8	Transaction identifier
9	Client address
10	User name
11	Elapsed time in milliseconds.
12	Number of bytes sent to the browser
13	Description of operation.

DebugDumpFilename=<filename>

This defines a file to which all M-Script output is copied.

SessionDir=<dirname>

By default the values in a session are stored in a file in a directory called Sessions located in the current directory of the M-Script interpreter. This directory will be created if it does not exist. With the SessionDir parameter it is possible to specify another session directory.

TmpDir=<dirname>

When M-Script needs to create a temporary file, it first looks for one of the environment variables TMP, TEMP, or MaconomyTmpDir. One of these should normally be set by external means, but they may also be set by a TmpDir=<dirname> entry in the initialization file. The specified directory name is then used to store the temporary files.

SessionLifeTime=<seconds>

This entry defines the number of seconds a session file may exist on the web server. Session files which are older than the specified lifetime will be deleted in the next session cleanup. The <seconds> field is an integer that denotes the number of seconds. The default value is equal to 12 hours (43200 seconds).

SessionCleanupProbability

This entry defines the probability that M-Script will perform a session cleanup. The default probability is 0.01 (that is, 1%). On systems with a high load a lower value might be better, while systems with a very low load (for example, a few hits a day) might benefit from a higher value.

SessionCleanupInterval=<seconds> (Deprecated)

This entry defines the interval in which the M-Script cleanup daemon is going to scan the web server for old session files. The <seconds> field is an integer that denotes the number of seconds between each scan. The default value is equal to 15 minutes (900 seconds).

SessionIPCheck=1 | 0

This entry defines whether M-Script should verify the ownership of session files against the IP address of the client. In environments where the client's IP address can change dynamically, it may be necessary to omit this check. By default the check is performed.

SessionTimeOut=<seconds>

This entry defines the time a script will wait while trying to acquire a session lock (gaining access to a locked session file).

MScriptDaemonLogfile=<filename>

This entry defines the name of a file that the M-Script cleanup daemon can use for logging purposes. If this option is left out then all logging is disabled. This features has been included for debugging purposes—not for use in a live system.

FileSystemRoot=<path> [; <root>=<path>]*

This entry defines the root of the file system and is mandatory for all file:: functions. In addition to the default root it is possible to specify one or more named roots.

GuiFramesMax=<number>

This entry sets the maximum number of non-persistent GUI frames that can exist in a session at the same time.

Localization=<locale>, Localization=<locale>,<mapfile>, Localization=<locale>,<dictionary>,[<ignorelist>]

Specify dynamic localization for <locale> which must be a single uppercase letter. The first form can be used to make a locale known to M-Script without specifying a dictionary. The second form can be used if you have a pre-built map file but not the original dictionary (map files can be recognized by their .hash suffix). The third form should be used if you have the dictionary and optionally an ignore list with phrases that should never be translated.

See String Localization for an introduction to string localization in M-Script.

LazyPackageLoad=1 | 0

Set this entry to “1” to make M-Script load packages lazily. By default M-Script loads packages statically (see Scopes and Initialization).

FlushAndCloseOutput=1 | 0

When all output is generated, and before M-Script considers cleaning up session files, M-Script will flush stdout and stderr and close stdin, stdout, and stderr. However, this behavior has been seen to cause M-Script to be terminated by the web server, causing M-Script to leave files in the directory for temporary files.

Set this entry to “0” to disable this behavior if experiencing a build-up of files whose names begin with “MSCRIPT” in the directory for temporary files.

Environment=var_1=val_1;var_2=val_2;...;var_x=val_x

Several M-Script settings, such as the global package root and the encoding used by XML transcoding are configurable through environment variables. These variables are set for the system user that invokes the M-Script interpreter. This can be difficult to manage, and may also be inadequate, for example if different web services on the same web server needs different settings.

Using the Environment option, you can set environment variables for M-Script directly in the configuration file. The individual environment variable assignments can be placed on different lines, as long as each line is terminated by a semicolon:

```
Environment = var_1=val_1;
              var_2=val_2;
              ...;
              var_x=val_x
```

Note that it is important that the last line does not end with a semicolon, as the next setting will then be interpreted as an environment variable assignment.

Example:

This example sets the LC_ALL to UTF-8:

```
Environment = LC_ALL=en_US.UTF-8
```

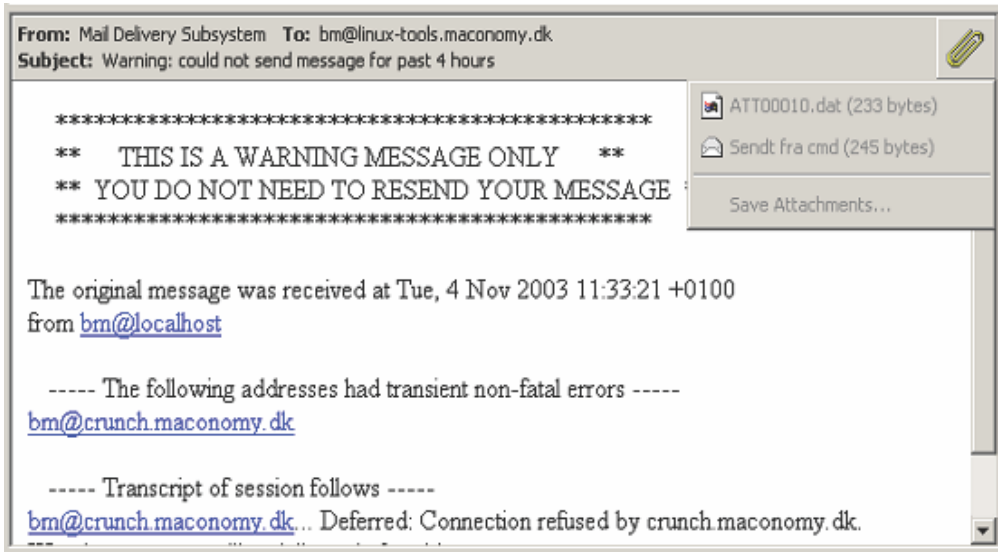
Appendix

The POP3 Package

Example 1 — A Delivery Status with an Attached Message/rfc822

The following is a complete example of the object returned by pop3::getmessage() when reading a message message delivery status with an attached message with Content-Type message/rfc822.

In Outlook Express, the message looks like the following figure.




```
{
   kind:"multipart",
   header:{
      Date:{
         UTCDate:04.11.2003,
         UTCTime:15:15:51
      },
      From:{
         name:"Mail Delivery Subsystem",
         email:"MAILER-DAEMON@linux-tools.maconomy.dk"
      },
      To:[
         {
            name:"",
            email:"bm@linux-tools.maconomy.dk"
         }
      ],
      MIME-Version:{
         major:1,
         minor:0
      },
```

```
        Content-Type:{
            media:"multipart",
            sub:"report",
            specifiers:@{
            report-type:"delivery-status",
            boundary:"QAA06413.1067958951/linux-tools.maconomy.dk"
        }
    },
    Subject:"Warning: could not send message for past 4 hours",
    cc:[
    ],
    bcc:[
    ],
    Reply-To:{
        name:"",
        email:""
    }
},
rawHeader:@{
    Return-Path:[
        {
            value:"",
            specifiers:@{
            }
        }
    ],
    Received:[
        {
            value:"from localhost (localhost)
by linux-tools.maconomy.dk (8.9.3/8.9.3) with internal id QAA06413; Tue, 4 Nov
2003 16:15:51 +0100",
            specifiers:@{
            }
        }
    ],
    Date:[
        {
            value:"Tue, 4 Nov 2003 16:15:51 +0100",
            specifiers:@{
            }
        }
    ],
    From:[
        {
            value:"Mail Delivery Subsystem ",
            specifiers:@{
            }
```

```
    }
],
Message-Id:[
    {
        value:"<200311041515.QAA06413@linux-tools.maconomy.dk>",
        specifiers:@{
        }
    }
],
To:[
    {
        value:"bm@linux-tools.maconomy.dk",
        specifiers:@{
        }
    }
],
MIME-Version:[
    {
        value:"1.0",
        specifiers:@{
        }
    }
],
Content-Type:[
    {
    value:"multipart/report", specifiers:@{
        report-type:"delivery-status",
        boundary:"QAA06413.1067958951/linux-tools.maconomy.dk"
    }
    }
],
Subject:[
    {
        value:"Warning: could not send message for past 4 hours",
        specifiers:@{
        }
    }
],
Auto-Submitted:[
    {
        value:"auto-generated (warning-timeout)",
        specifiers:@{
        }
    }
],
Status:[
    {
```

```
            value:"RO",
            specifiers:@{
            }
        }
    ]
},
parts:[
    {
        kind:"part",
        header:{
            Content-Type:{
                media:"text",
                sub:"plain",
                specifiers:@{
                    charset:"us-ascii"
                }
            }
        },
        rawHeader:@{
        },
        length:616,
        data:istream
    },
    {
        kind:"part",
        header:{
                Content-Type:{
                    media:"message",
                    sub:"delivery-status",
                    specifiers:@{
                    }
                }
            },
            rawHeader:@{
                Content-Type:[
                    {
                        value:"message/delivery-status",
                        specifiers:@{
                        }
                    }
                ]
            },
            length:311,
            data:istream
        },
        {
            kind:"multipart",
            header:{
```

```
    Content-Type:{
        media:"message",
        sub:"rfc822",
        specifiers:@{
        }
    }
},
rawHeader:@{
    Content-Type:[
        {
            value:"message/rfc822",
            specifiers:@{
            }
        }
    ]
},
parts:[
    {
        kind:"part",
        header:{
            Date:{
                UTCDate:04.11.2003,
                UTCTime:10:33:21
            },
            From:{
                name:"",
                email:"bm"
            },
            To:[
                {
                    name:"",
                    email:"bm@crunch.maconomy.dk"
                }
            ],
            Subject:"Sendt fra cmd",
            cc:[
            ],
            bcc:[
            ],
            Reply-To:{
                name:"",
                email:""
            },
            Content-Type:{
                media:"text",
                sub:"plain",
                specifiers:@{
                    charset:"us-ascii"
```

```
        }
    }
},
rawHeader:@{
    Return-Path:[
        {
            value:"",
            specifiers:@{
            }
        }
    ],
    Received:[
        {
            value:"(from bm@localhost)
by linux-tools.maconomy.dk (8.9.3/8.9.3) id LAA26660
for bm@crunch.maconomy.dk; Tue, 4 Nov 2003 11:33:21 +0100",
            specifiers:@{
            }
        }
    ],
    Date:[
        {
            value:"Tue, 4 Nov 2003 11:33:21 +0100",
            specifiers:@{
            }
        }
    ],
    From:[
        {
            value:"bm",
            specifiers:@{
            }
        }
    ],
    Message-Id:[
        {
            value:"<200311041033.LAA26660@linux-
        tools.maconomy.dk>",
            specifiers:@{
            }
        }
    ],
    To:[
        {
            value:"bm@crunch.maconomy.dk",
            specifiers:@{
            }
        }
```

```
                    ],
                    Subject:[
                        {
                            value:"Sendt fra cmd",
                            specifiers:@{
                            }
                        }
                    ]
                },
                length:0,
                data:istream
            }
        ]
    }
]
}
```

Example 2 — An HTML Message with Attachment

The following is a complete example of the object returned by pop3::getmessage() when reading a message containing a text and an HTML version of the body text. The HTML version has an inserted gif image which is referenced internally in the message. Finally, there is a PDF attachment.

In Outlook Express, the message looks like the following figure.


```
{
    kind:"multipart",
    header:{
        From:{
            name:"Test Testsen",
            email:"bm@crunch.maconomy.dk"
        },
        To:[
            {
```

```
            name:"",
            email:"hulla@børgesen.dk"
        }
    ],
    Subject:"Hvad med PDF-filer???",
    Date:{
        UTCDate:24.11.2003,
        UTCTime:12:21:24
    },
    MIME-Version:{
        major:1,
        minor:0
    },
    Content-Type:{
        media:"multipart",
        sub:"mixed",
        specifiers:@{
        boundary:"----=_NextPart_000_001B_01C3B28D.DBE3D420"
        }
    },
    cc:[
    ],
    bcc:[
    ],
    Reply-To:{
        name:"",
        email:""
    }
},
rawHeader:@{
    From:[
        {
            value:""Test Testsen" ",
            specifiers:@{
        }
        }
    ],
    To:[
        {
            value:"hulla@børgesen.dk",
            specifiers:@{
            }
        }
    ],
    Subject:[
        {
            value:" Hvad med PDF-filer???",
            specifiers:@{
```

```
            }
        }
    ],
    Date:[
        {
            value:"Mon, 24 Nov 2003 13:21:24 +0100",
            specifiers:@{
            }
        }
    ],
    MIME-Version:[
        {
            value:"1.0",
            specifiers:@{
            }
        }
    ],
    Content-Type:[
        {
            value:"multipart/mixed",
            specifiers:@{
            boundary:"----=_NextPart_000_001B_01C3B28D.DBE3D420"
            }
        }
    ],
    X-Priority:[
        {
            value:"3",
            specifiers:@{
            }
        }
    ],
    X-MSMail-Priority:[
        {
            value:"Normal",
            specifiers:@{
            }
        }
    ],
    X-MimeOLE:[
        {
            value:"Produced By Microsoft MimeOLE V6.00.2800.1165",
            specifiers:@{
            }
        }
    ],
    Status:[
        {
```

```
            value:"RO",
            specifiers:@{
            }
        }
    ]
},
parts:[
    {
        kind:"multipart",
        header:{
            Content-Type:{
                media:"multipart",
                sub:"related",
                specifiers:@{
                    type:"multipart/alternative",
                    boundary:"----=_NextPart_001_001C_01C3B28D.DBE3D420"
                }
            }
        },
        rawHeader:@{
            Content-Type:[
                {
                    value:"multipart/related",
                    specifiers:@{
                        type:"multipart/alternative",
                        boundary:"----=_NextPart_001_001C_01C3B28D.DBE3D420"
                    }
                }
            ]
        },
        parts:[
            {
                kind:"multipart",
                header:{
                    Content-Type:{
                        media:"multipart",
                        sub:"alternative",
                        specifiers:@{
                            boundary:"----=_NextPart_002_001D_01C3B28D.DBE3D420"
                        }
                    }
                },
                rawHeader:@{
                    Content-Type:[
                        {
                            value:"multipart/alternative",
                            specifiers:@{
```

```
                boundary:"----
            =_NextPart_002_001D_01C3B28D.DBE3D420"
            }
        }
    ]
},
parts:[
    {
        kind:"part",
        header:{
            Content-Type:{
                media:"text",
                sub:"plain",
                specifiers:@{
                    charset:"iso-8859-1"
                }
            }
        },
        rawHeader:@{
            Content-Type:[
                {
                    value:"text/plain",
                    specifiers:@{
                        charset:"iso-8859-1"
                    }
                }
            ],
            Content-Transfer-Encoding:[
                {
                    value:"quoted-printable",
                    specifiers:@{
                    }
                }
            ]
        },
        length:68,
        data:istream
    },
    {
        kind:"part",
        header:{
            Content-Type:{
                media:"text",
                sub:"html",
                specifiers:@{
                    charset:"iso-8859-1"
                }
            }
```

```
            },
            rawHeader:@{
                Content-Type:[
                    {
                        value:"text/html",
                        specifiers:@{
                            charset:"iso-8859-1"
                        }
                    }
                ],
                Content-Transfer-Encoding:[
                    {
                        value:"quoted-printable",
                        specifiers:@{
                        }
                    }
                ]
            },
            length:609,
            data:istream
        }
    ]
},
{
    kind:"part",
    header:{
        Content-Type:{
            media:"image",
            sub:"gif",
            specifiers:@{
                name:"Maconomy_M.gif"
            }
        }
    },
    rawHeader:@{
        Content-Type:[
            {
                value:"image/gif",
                specifiers:@{
                    name:"Maconomy_M.gif"
                }
            }
        ],
        Content-Transfer-Encoding:[
            {
                value:"base64",
                specifiers:@{
                }
```

```
                    }
                ],
                Content-ID:[
                    {
                        value:"<001a01c3b285$7a1a8a20$626410ac@cphpc353>",
                        specifiers:@{
                        }
                    }
                ]
            },
            length:1852,
            data:istream
        }
    ]
},
{
    kind:"part",
    header:{
        Content-Type:{
            media:"application",
            sub:"pdf",
            specifiers:@{
                name:"DocBox.pdf"
            }
        }
    },
    rawHeader:@{
        Content-Type:[
            {
                value:"application/pdf",
                specifiers:@{
                    name:"DocBox.pdf"
                }
            }
        ],
            Content-Transfer-Encoding:[
                {
                    value:"quoted-printable",
                    specifiers:@{
                    }
                }
            ],
            Content-Disposition:[
                {
                    value:"attachment",
                    specifiers:@{
                        filename:"DocBox.pdf"
                    }
```

```
                    }
                ]
            },
            length:5464,
            data:istream
        }
    ]
}
```

Licenses

The Apache Software License

The Apache Software License, Version 1.1

Copyright (c) 1999-2000 The Apache Software Foundation. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment:

   “This product includes software developed by the Apache Software Foundation (http://www.apache.org/).”

   Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.
4. The names “Xerces” and “Apache Software Foundation” must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.
5. Products derived from this software may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED “AS IS” AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDA- TION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCI- DENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUP- TION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation and was originally based on software copyright (c) 1999, International Business Machines, Inc., http://www.ibm.com. For more information on the Apache Software Foundation, please see http://www.apache.org.

The Boost Regex++ Library

Copyright (c) 1998-2001

Dr John Maddock

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copy- right notice appear in all

copies and that both that copyright notice and this permission notice appear in supporting documentation. Dr John Maddock makes no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

Part 2 – M-Script Maconomy API Reference

This is the reference document for the M-Script Maconomy API. This introduction aims at giving the reader an overview of what the M-Script Maconomy API is all about.

Prerequisites

To understand and use the M-Script Maconomy API it is necessary to have a good understanding of the M-Script programming language. Consequently, no basic M-Script language constructs are explained in this document. It is assumed that you already know the M-Script language well.

The M-Script Language Reference gives a thorough introduction to the M-Script language.

What is the M-Script Maconomy API?

The acronym API stands for "application programmer interface," and the M-Script Maconomy API is an interface that allows an M-Script programmer to access data in the Maconomy database.

The API consists of a collection of functions and procedures that can be called from M-Script. All of these subroutines reside in the M-Script module `maconomy`, which means that all subroutine names must be prefixed with "`maconomy::`".

The following is an example of a simple M-Script using the Maconomy API.

```
#version 14
newsession();
maconomy::login("Administrator", "123456");
var echoReply = maconomy::echo("Hello Maconomy!");
print("Echo from Maconomy: ^1\n", echoReply);
maconomy::logout();
deletesession();
```

This script performs a login to Maconomy, calls a function in the API, prints the result, and finally logs out from Maconomy.

Because the name "M-Script Maconomy API" is long, this document uses the term "API."

Contents of the M-Script Maconomy API

The M-Script Maconomy API is made up of many parts, which are described briefly in this section. In the following sections, all of the parts that make up the API are described in detail.

The M-Script Maconomy API is made up of:

- Login Session Interface — Logging in and out of Maconomy.
- Transaction Control Interface — Starting, ending, and rolling back database transactions.
- MQL Interface — Read-only access to the Maconomy database using MQL and universes.
- SQL Interface — Full SQL access to the Maconomy database.
- Analyzer Interface — Access to execution of Analyzer reports.
- RGL Report Interface — Access to execution of RGL reports.
- Value Storage Interface — Storage and retrieval of M-Script values in the database.

- Dialog Interface — Access to all windows in Maconomy.
- Pop-Up Interface — Handling of Maconomy's pop-up types and values.

Some of the subroutines that make up the API (such as `login`, `logout`, `sql`, …) are also documented in the M-Script Language Reference. However, because these subroutines are part of the API, they are also included in this document for completeness.

Reading this Section

In this section the term *iff* is used meaning "if and only if". The sentence "The return value is `true` *iff* the value exists" means that the return value is `true` if the value exists and `false` if it does not exist.

Throughout this section examples are given to illustrate the use of the API. To simplify the examples, not every possible error is checked when calling an API subroutine. This makes it easier to read the examples. However, you should be aware that when using the API for creating real applications, extensive error checking is necessary to produce robust M-Script applications.

Also, the example scripts are not complete scripts. To reduce the examples to a reasonable size, details about creating sessions, logging in, and so forth, are left out unless understanding the example require these details to be present.

An ellipsis (. . .) is used in many places to denote that some code that is irrelevant to the example has been left out. An ellipsis is both used to denote omitted M-Script statements.

```
// Function for getting user's max. rows
selection. function
readUserMaxRowsSelection()
{
  ...
}
```

and to denote omitted object or array complexity

```
{
  a : 42;
  b : [...], // Array details omitted.
  c : {...}, // Object details omitted.
...        // More properties omitted.
}
```

Finally, an ellipsis is used in subroutine prototypes to denote a subroutine that accepts an arbitrary number of arguments.

```
// 'sampleFunction' accepts 2 or more
arguments. function sampleFunction(arg1,
arg2, ...);
```

For every function or procedure in this section, a note of the contexts in which the procedure or function is permitted is indicated for each function or procedure. The allowed context is specified in a paragraph called "Context."

Error Handling in the Maconomy API

Until version 6.0 of the M-Script Maconomy API, errors received from the Maconomy application were reported through the `dialogStatus` return object. M-Script now offers support for exception handling.

To Throw or not to Throw

The Maconomy API can throw exceptions on errors that previously caused functions to return a `dialogStatus` object with the property `ok` set to `false`. Because this breaks backward compatibility with earlier version of this API, Maconomy offers a transition approach to enable existing M-Script code to continue to work as expected:

- Invocations of the API from scripts marked with `#version 6` or higher enable the newer exception mechanism.
- The old API behavior can be retained in a script marked with `#version 6` by putting the directive `#set stdlibStyle 5` after the `version` tag in that script.

Because existing code is most definitely marked with a version prior to 6, these scripts can continue working as they always have done.

Exception Types

The exceptions thrown from the Maconomy API are ordered according to the information that they contain. A few main exception classes exist:

- `...::maconomy` is thrown on the most general errors such as bad parameter values, and so on. These errors carry no other information than the error message in the `messageError` object.
- `...::maconomy::login` is thrown on a failed login attempt. The value of this exception is the `loginError` object.
- `...::maconomy::dialog` is thrown on a failed dialog operation. Its value is the `dialogError` object.
- `...::maconomy::license` is thrown when a Maconomy function is used for which there is no required add-on on the server.
- `...::maconomy::hllLock` is thrown on a failed high-level lock operation.

The common exception prefix is `error::stdlib`.

Login Session Interface

Before being able to access any other function in the API, a login session must be established. When the login session is no longer needed, the user must be logged out. This functionality is supplied by this part of the API.

Subroutines

The subroutines for handling login sessions are:

- function maconomy::getServerHandle([label])

  Returns a server handle for the specified server label. If a connection with that label is not declared in the initialization file, the operation will fail. If exceptions are enabled, this will cause a run-time exception, otherwise `null` is returned. If no label is specified, a handle for the default connection is returned.

- function maconomy::login(userName, password)

  Creates a login session in Maconomy. An M-Script session must exist before calling this function.

- function maconomy::login(userName)

  Creates a login session in Maconomy using stamped login privileges. An M-Script session must exist before calling this function.

- function maconomy::singleLogin()

  Performs a login based on the user authentication mechanisms of the host platform. Currently, Windows NT is the only platform supported by this function. In addition, web-based authentication is only supported by IIS.

- procedure maconomy::impersonateUser(userName)

  This procedure can be used during an administrator login session to temporarily assume the identity of another user on the system.

- procedure maconomy::revertToSelf()

  This procedure will discontinue a user impersonation, returning the original administrator identity to the login session.

- procedure maconomy::logout()

  Calling this procedure ends a login session.

- function maconomy::echo(value)

  Sends `value` to the server which sends back the same value as the return value of the function. This function exists only to test that a login session has been established.

Multiple Server Connections

From version 4.1 of M-Script it is possible to connect to several Maconomy applications simultaneously. To manage the available connections, the concept of a server handle has been introduced. Each server handle represents one server connection, and all functions in the `maconomy` library accept a server handle as an optional first parameter. If no server handle is specified, the default connection is assumed, thereby ensuring full backward compatibility with previous versions of the Maconomy interface.

Note that an invalid server handle, such as the `null` value returned by `maconomy::getServerHandle()` on an unknown server label in older scripts, can be a spurious source of parameter mismatch errors in scripts where exceptions are disabled. This is because server handles are stripped from the argument list before the function call is passed to the Maconomy API. Invalid server handles are not stripped, which causes the API functions to receive arguments which differ in cardinality or type from what was expected.

Because the optional server handle can be applied to all `maconomy` functions, it is omitted from the function descriptions throughout this document.

Example

The following example illustrates login session handling.

```
#version 1
newsession();
   var loginResult = maconomy::login("Administrator", "123456");
if (loginResult.ok)
{
  var echo = maconomy::echo("Hello Maconomy!");
  print("Echo from Maconomy: ^1\n", echo);
  maconomy::logout();
}
deletesession();
```

First, an M-Script session is created. This is needed to perform the login to Maconomy by calling the function `maconomy::login`. If the login succeeds, the function `maconomy::echo` is called. This function simply sends the M-Script value that is passed to it to the server, which sends the value back as the function return value. The return value is printed, and a logout is performed. Finally, the MScript session is deleted.

The output from the program is:

```
Content-Type: text/html
Echo from Maconomy: Hello Maconomy!
```

In this example, the default connection was used. A different server could have been chosen by first calling `getServerHandle()` to obtain a specific server handle, and then passing this server handle to all `maconomy` functions.

```
var srv = getServerHandle("srv1");
   maconomy::login(srv, "Administrator", "123456");
    ...
   maconomy::logout(sr
   v);
```

Transaction Control Interface

Many of the subroutines in the M-Script Maconomy API allow an M-Script programmer to update data in the Maconomy database. Such update operations are under transaction control by the underlying database system. This means that updates to the database must be committed before becoming visible to other Maconomy users. It is also possible to roll back all of the changes that were made during a transaction.

If a transaction is not already started, then a new transaction is started when calling an API subroutine that accesses the Maconomy database in some way.

Transactions can be ended automatically after each subroutine call, or they can be ended manually.

When the user logs out from the database system—calling `maconomy::logout`—the current transaction is rolled back.

Subroutines

The API includes the following subroutines for transaction control.

- function maconomy::autoCommit(yes)

  Controls whether or not all transactions should be automatically committed after each subroutine call that modifies the database. By default automatic commit is turned off.

- procedure maconomy::commit()

  Commits any changes that were made to the database since the current transaction was started. This also ends the current transaction.

- procedure maconomy::rollback()

  Rolls back all of the changes to the database since the current transaction was started. This also ends the current transaction.

Example

The following example illustrates the use of the transaction control subroutines. This example makes use of the Dialog Interface. What is interesting in this example is the use of the subroutines `rollback`, `commit`, and `autoCommit`.

Consider this code fragment:

```
// Open dialog 'Users' and get record for the user
'User'. var dialogId =
maconomy::dialogOpen("Users");

var dialogRet =

    maconomy::dialogGet(dialogId,

                        {NameOfUser : {value : "User"}});

print("Getting user ok? ^1\n", dialogRet.status.ok);

// Delete the entry.

dialogRet = maconomy::dialogDeleteUpper(dialogId);

print("Deleting user ok? ^1\n", dialogRet.status.ok);
```

```
// Regret and rollback
operation.
maconomy::rollback();
```

First, this opens the dialog Users in the Maconomy application. Then it gets the entry for a user named “User.” After getting the user entry, this deletes it. Then it performs a rollback. The result is that no changes are made to the database.

The generated output is:

```
Getting user ok? true

Deleting user ok? true
```

To verify that the user has not been deleted this tries to find it again.

```
// Try to get record for 'User' again.

// It should still be
there. dialogRet =

    maconomy::dialogGet(dialogI
                d,

                        {NameOfUser : {value : "User"}});

print("Getting user again ok? ^1\n", dialogRet.status.ok);
```

This code prints:

```
Getting user again ok? true
```

As this illustrates, the user was not deleted by the previous call to `dialogDeleteUpper`. Had this example replaced the call to `rollback` with a call to `commit` the change to the database would have been committed, and the attempt to look up the user again would have failed.

The function `autoCommit` can be used to avoid calling `commit` explicitly to commit changes to the database. Modifying the preceding code to use this function instead gives:

```
// Make sure every change is committed
automatically. maconomy::autoCommit(true);

// Try to delete user 'User' again.

dialogRet = maconomy::dialogDeleteUpper(dialogId);

print("Deleting user again ok? ^1\n", dialogRet.status.ok);
```

The call to `autoCommit` instructs M-Script to automatically commit all changes after each API subroutine call. This makes it unnecessary to call `commit`. However, there are some situations in which automatic commit is not appropriate. In these situations `commit` and `rollback` give you full control of when data is written to the database.

MQL interface

The M-Script Maconomy API includes an MQL interface that gives a programmer read-only access to the entire Maconomy database through the Maconomy Query Language. MQL is expected to replace the use of SQL completely in the future, although currently only data selection is supported in MQL.

The main difference between MQL and SQL is the separation of the data model and the command. In SQL, the data model is specified in each command, but in MQL, the data model is specified in a universe. This separation enables the reuse of the data model in multiple commands, and it enables the MQL command to be developed without prior knowledge of the data model. For more information about MQL and Universes, see the MQL Language Reference.

Subroutines

The MQL Interface includes the following subroutines:

- function maconomy::mqlOpen(mqlQuery [, mqlBind])

  This function defines a query handler from an MQL query command. The parameter `mqlQuery` can be any MQL `mselect` statement. Values for query parameters and page size information can be specified in the `mqlBind` parameter.

- procedure maconomy::mqlBind(mqlHandler, mqlBind)

  This function redefines the values for query parameters page size and format specification for the query that is defined by the query handler that is provided by the parameter `mqlHandler`.

- procedure maconomy::mqlClose(mqlHandler)

  This procedure releases resources that are used by the query handler that is provided by the parameter `mqlHandler`.

- function maconomy::mqlGetDef(mqlHandler)

  This function gets information about type and titles for fields selected in the query that is defined by the query handler `mqlHandler`.

- function maconomy::mqlGet(mqlHandler)

  This function executes the query that is defined by the query handler `mqlHandler` and returns all rows.

- function maconomy::mqlGetPage(mqlHandler)

  This function executes the query that is defined by the query handler `mqlHandler` and returns a specific page of rows.

- function maconomy::mqlGetPagePrev(mqlHandler)

  This function executes the query that is defined by the query handler `mqlHandler` and returns the previous page of rows.

- function maconomy::mqlGetPageNext(mqlHandler)

  This function executes the query that is defined by the query handler mqlHandler and returns the next page of rows.

- function maconomy::mql(mqlQuery [, mqlBind [, mqlParms]])

  This function executes an MQL query command. The parameter `mqlQuery` can be any MQL `mselect` statement. Values for query parameters can be specified in the `mqlBind`

parameter. Additional values can be supplied using the `mqlParms` parameter. All rows are returned.

- function maconomy::mqlUniverseQuery(universeName, searchBind, parms)

  This function selects a universe based on the `universeName` string parameter and the optional string property `interfaceName` in parms. A query is created from the field selections and restrictions given by the parameter `searchBind`, which is an object of type `searchBind`.Finally the values for query parameters page size and format specification are redefined for the query if `searchBind` contains an `mqlBind` object. In addition, the universe definition and the mql definition can be fetched.

- function maconomy::universeOpen(universeName[, interfaceName])

  This function creates a universe handler to the universe given by the parameter `universeName`. A universe handler can be used for dynamic queries, which are queries where field selection and restrictions are built at run time. A universe handler can also be used for getting field information about fields in a universe.

- procedure maconomy::universeClose(universeHandler)

  This procedure release resources that are used by the universe handler given by the parameter `universeHandler`.

- function maconomy::universeGetDef(universeHandler)

  This function gets information about fields in the Universe, defined by the universe handler given by the parameter `universeHandler`.

- function maconomy::universeBind(universeHandler, searchBind)

  This function defines a query handler from a universe handler, and field selections and restrictions given by the parameter `searchBind`. The query handler can be used with the `maconomy::mql*` set of functions.

Examples

The following examples use the MQL Interface. The first example extracts data using a query handler, while the others extract data using an MQL command directly. The second example also demonstrates the use of query parameters that are available in MQL. The third example extracts data using a universe handler.

Extract Data Using a Query Handler

This example extracts data by page from the Maconomy database using a query handler that is defined by the function `maconomy::mqlOpen`. Every call to `maconomy::mqlGetPageNext` returns a page of two rows, because the page size is set to two in the `mqlBind` parameter to the function `maconomy::mqlOpen`.

```
// Define a query using MQL
var mqlQuery = 'MQLQUERY';
<MQL 1.3>
MSELECT EmployeeNumber, Name1 FROM Employee ORDER BY
EmployeeNumber
MQLQUERY
// Execute the query and read rows
pagewise try  {
```

```
    var mqlBind    = {
      pageSize:2 };
  var  mqlHandler  =  maconomy::mqlOpen(mqlQuery,
  mqlBind);        var        mqlResult         =
  maconomy::mqlGetPageNext(mqlHandler);
  while(mqlResult.more) {
    ...
    mqlResult = maconomy::mqlGetPageNext(mqlHandler);
  }
  maconomy::mqlClose(mqlHandler);
}
catch(e) {
  ...
}
```

The return value saved in the variable `mqlResult` in the first pass of the loop looks like the following:

```
{
  mo
  re
  :t
  ru
  e,
  pa
  ge
  :1
  ,
  mq
  lD
  at
  a:
  {
    r
    o
    w
    s
    :
    [
      @
      {
        EmployeeNumber:{
          value:"1000"
        }
        ,

        N
        a
        m
        e
```

```
        1
        :
        {
          value:"James Hanson"
        }
      },
      @{
        EmployeeNumber:{
          value:"1001"
        }
        ,

        N
        a
        m
        e
        1
        :
        {
          value:"Bill Jackson"
        }
      }
    ]
  }
}
```

Extract Data Using a Query Parameter

This example extracts data without using a query handler. It defines a query that contains the query parameter `parmEmployeeNumber` and binds a value to this query parameter when executing the query.

```
// Execute the query without a query handler and use a query
parameter try  {
  var mqlQuery = 'MQLQUERY';
<MQL 1.3>
MSELECT Name1 FROM Employee WHERE EmployeeNumber =
parmEmployeeNumber
USING PARAMETERS parmEmployeeNumber : String
MQLQUERY
  var mqlBind    = { parameters:{
                       parmEmployeeNumber : { value : "1000"}
                     }
                }
                ;
```

```
  var mqlResult  = maconomy::mqlGet(mqlQuery, mqlBind);
  ...
}
catch(e) {
  ...
}
```

The return value that is saved in the variable `mqlResult` has the same structure as in the previous example. In this example only one row is returned.

```
{
  more:fal
  se,
  page:1,
  mqlData:
  {
    rows
     :[
      @
      {
        EmployeeNumber:{
          value:"1000"
        },
        Name
        1:{
          value:"James Hanson"
        }
      }
    ]
  }
}
```

Extract Data Using a Universe Handler

This example defines a universe handler from a universe name and uses the universe handler to build a query handler that can be used with the `maconmy::mqlGet` function. Note that every Maconomy relation is also available as a universe.

```
try  {
  var universeHandler = maconomy::universeOpen("Employee");
  var searchBind = {
        columnIndexOutput
           : [
           "EmployeeNumbe
           r",
           "Name1"
        ],
        restrictions : [
```

```
            { columnName:"EmployeeNumber", restriction:["1000..1010"]
            }
          ]
        };
    var mqlHandler =
    maconomy::universeBind(universeHandler,searchBind
    ); var mqlResult  = maconomy::mqlGet(mqlHandler);
    ...
    maconomy::mqlClose(mqlHandle
    r);
}
catch(e) {
    ...
}
```

SQL Interface

The M-Script Maconomy API includes an SQL interface that gives a programmer read-only access to the entire Maconomy database and restricted write access to specific database relations.

Subroutines

The SQL Interface includes the following subroutines:

- function maconomy::sql(statement, maxRecs, firstRec, ...)

  The parameter `statement` can be any SQL select statement. You can limit the number of records retrieved from the database.

- function maconomy::sqlRestricted(statement, maxRecs, firstRec, ...)

  This function works exactly like `maconomy::sql`, but Maconomy's access control system restricts which data can be retrieved from the database. This means that it is significant which user was logged on (using `maconomy::login`) before this function is called.

- function maconomy::sqlValidate(statement)

  This functions checks whether the specified SQL statement is valid in a call to `maconomy::sql`.

- function maconomy::sqlValidateRestricted(statement)

  This functions works exactly as `maconomy::sqlValidate` except that it checks whether the specified SQL statement is valid in a call to `maconomy::sqlRestricted`.

- function maconomy::sqlInsert(relName, values)

  Inserts a record into the relation named `relName`. Only very few relations can be modified using this function, but it is possible for all customer-specific relations.

- function maconomy::sqlUpdate(relName, values, whereClause)

  Updates existing records in the relation named `relName`. A `where` clause is used to specify which records to update. Only very few Maconomy relations can be modified using this function, but it is possible for all customer-specific relations.

- function maconomy::sqlDelete(relName, whereClause)

  Deletes records from the relation named `relName`. A `where` clause is used to specify which records to delete. Only very few relations can be modified using this function, but it is possible for all customer specific relations.

Examples

The following examples use the SQL Interface. One example extracts data, while the other updates existing records in the database

Extract Data

This simple example extracts some data from the Maconomy database using the function `maconomy::sql`.

```
// Get the name of the first 2 users in the
system. var sqlResult = maconomy::sql('SQL',
2, null);
```

```
  select nameofuser from userinformation order by nameofuser
SQL
```

The return value saved in the variable `sqlResult` looks like this:

```
{
  result:{
    columnin
      dex:[
      "nameo
      fuser"
    ],
    columns:{
      nameofuser:{
        title:"NAMEOF
        USER",
        index:0,
        type:"string"
      }
  },
  rows:[
      {
        nameofuser:{
          value:"Administrator"
        }
      },
      {
        nameofuser:{
          value:"John Doe"
        }
      }
    ]
  }
}
```

It is an object with a single property called `result`, which is a new object that contains all of the data that is returned from the Maconomy server. This new object has the following properties:

- columnindex — This property is an array that contains the names of all of the columns that are selected in the select statement. The columns are listed in the order in which they appeared in the select clause. This case selected only one column, nameofuser.
- columns — This property is an object that has properties that are named after the selected columns. Because this example selected only one column, there is only one property, nameofuser. This property is an object that holds information about the title, index, and type of the values in this column.
- rows — While the first two properties were descriptions of the columns, this property contains the data that is extracted from the database. The property is an array that has

an entry for each record that is fetched from the database. This example asked for a maximum of two records, and that is what is returned. What you can read is that the first two users in the system—ordered by name—are “Administrator ” and “John Doe.”

Regardless of which user logs into Maconomy (using `maconomy::login` earlier in the script), you get the same result using `maconomy::sql`. However, to impose access control that allows the user only to see what he or she is allowed to see, you can use the `maconomy::sqlRestricted` function instead.

Update Existing Records

The function `maconomy::sqlUpdate` allows you to change data in the Maconomy database, but only in the Enterprise Portal relations.

This example uses a fictitious relation `Rel` that has two columns `ItemNumber` and `ItemText`.

Assume that there is a record in this relation where `ItemNumber` is “1234” and that you want to change `ItemText` for this record to “Sofa, 2 persons, canvas.” You use the `sqlUpdate` function to do the job.

```
var values = { itemtext : "Sofa, 2 persons, canvas" };
var where  = "itemnumber = '1234'";
var sqlResult = maconomy::sqlUpdate("Rel", values, where);
```

The call to `sqlUpdate` generates an SQL `update` statement that is sent to the underlying database system. In this case, the generated statement is:

```
update Rel set itemtext = 'Sofa, 2 persons,
           canvas' where itemnumber =
           '1234';
```

From the return value in `sqlResult` you can see whether the operation succeeded and how many rows were updated in the database. In this example the value of `sqlResult` is:

```
{
  ok
  :
  true
  ,
  rows
  : 1
}
```

This means that the operation succeeded and that exactly one row was updated. Remember that the change must be committed via `maconomy::commit(); // Commit change to database.` This saves the change in the database.

Analyzer Interface

An integrated part of the Maconomy client is the Analyzer. The M-Script Maconomy API makes it possible to execute Analyzer reports from M-Script.

Note that some Analyzer reports can take a long time to execute and might consume a large amount of system resources on the Maconomy server. Executing such an Analyzer report might cause performance problems for the entire Maconomy system.

Subroutines

There is one subroutine in this part of the API:

- function maconomy::analyze(fileName, layoutName, rowSelection)

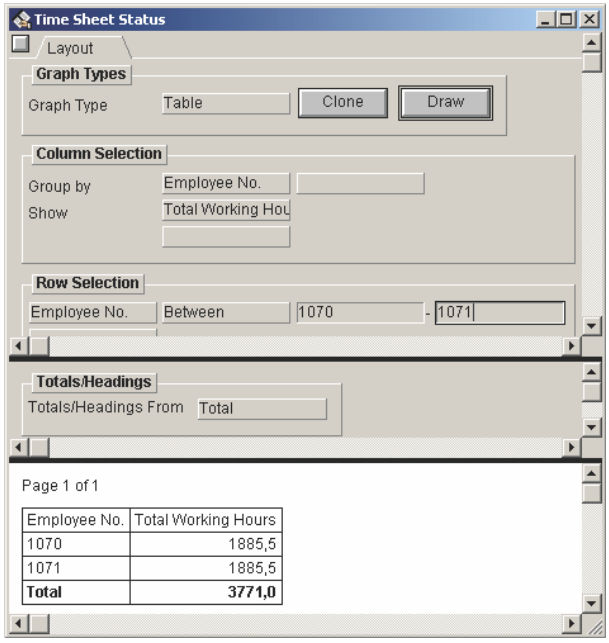
The parameter `fileName` is the name of an existing Analyzer report that is stored on the Maconomy server. The last two parameters are optional. The parameter `layoutName` tells the Analyzer which layout to choose in the report. If no layout name is given, the first layout is used. Using `rowSelection` it is possible to change the values of the row selection clause.

Examples

The following provides two examples of how to execute an Analyzer report from M-Script. The first example simply executes the default layout of a report. The second example illustrates how to modify the row selection of the pre-filled Analyzer report layout.

Execute a Report As-Is

This example executes the “Time Sheet Status” Analyzer report. For this example, there is a layout that finds the total number of working hours, grouped by the employee number.



To run the report from M-Script, use the following code:

```
var analyzeRes = maconomy::analyze("TimeSht.grf");
```

The result is found in `analyzeRes`, and the structure of the return value looks a lot like the type of data that is returned when you call `maconomy::sql`.

```
{
  result:{
    columnind
      ex:[
      "employ
      ee
      no.",
      "total working hours"
], columns:{
      'employee
        no.':{
        title:"Employ
        ee No.",
        index:0,
        type:"string"
        , isKey:true,
        sum:0.00,
        max:-999,999,999.00
      },
      'total working
        hours':{
        title:"Total Working
        Hours", index:1,
        type:
        "real
        ",
        isKey
        :fals
        e,
        sum:3
        ,749.
        00,
        max:1,885.50
      }
    }
    ,

    r
    o
    w
    s
    :
    [
   {
```

```
            'employee no.':{
              value:"1021"
            },
            'total working hours':{
              value:1,863.50
            }
          },
          {
            'employee no.':{
              value:"1028"
            },
            'total working hours':{
              value:1,885.50
            }
          }
        ]
      }
    }
```

Examining the return value reveals that the only differences between the result from a call to `analyze` and a call to `sql` are the extra properties `isKey`, `sum` and `max` inside the `rows` entries. In this example you can see that **Employee No.** is a key field in this report, whereas **Total Working Hours** is not. In addition, the maximum number of working hours for the selected employees and the total sum of working hours has been calculated.

Modify the Row Selection

Sometimes you need to be able to change the row selection of an Analyzer report without having to change the layout from a Maconomy client. You can do this using the last optional parameter to the `maconomy::analyze` subroutine.

```
var analyzeRes = maconomy::analyze("TimeSht.grf",
                                   "", // Use first
                                   layout.
                                   "rv1.1=1080&rv1.2=1080
                                   ");
```

The empty string that is passed as the second parameter means that you still want to use the first layout of the Analyzer report.

The third parameter tells `analyze` that you want to change the row selection. The first part of the string before the “&” character instructs `analyze` to change the value of row 1, column 1 of the row selection to “1080.” Likewise, you also want to change the value of row 1, column 2 of the row selection to “1080.” Comparing this to the figure in “Execute a Report As-Is,” you see that this example requested to get information about all employees with employee numbers between “1080” and “1080.” In other words: you want information about employee number “1080” only. Leaving out some details, the result is:

```
{
```

```
result
  :{
   columnindex : [...], // Details left
   out. columns : {...},       // Details
   left out. rows:[
     {
          'employee
           no.':{
         value:"108
             0"
           },
           'total working hours':{
             value:1,885.50
           }
         }
       ]
     }
  }
```

Value Storage Interface

It is useful to be able to store user-specific and application-specific values in the database from applications that are written in M-Script. Such values can be customized user settings or system-wide customized settings.

The Value Storage Interface provides access to storing named M-Script values in the Maconomy database. The stored values can either be user-specific values or system-wide values. User-specific values—also called user values—can only be read and written by the user who created those values. System-wide values—also called common values—can be read and written by any logged-in user. Recall that a user is logged in by calling the function `maconomy::login`.

You should be careful when choosing the names under which to store values. Remember that all stored values can be accessed by any application that is written in M- Script. Therefore, choosing value names improperly could mean that two M-Script applications both use the same stored value, but for completely different purposes. This could in the worst case lead to malfunction of both of the applications.

One naming convention could be to include the application name in the value. As an example, an M-Script application known as “Customer Portal” could choose to prefix the name of all stored values with `customerPortal` to ensure that the names are not used by other M-Script applications. This naming convention could lead to names like `customerPortalUserPrefs`, `customerPortalSettings`, and so forth.

Subroutines

The Value Storage Interface includes the following API subroutines:

- procedure maconomy::setUserValue(name, value)

  Stores or updates a named value for the current user in the Maconomy database.
- function maconomy::getUserValue(name)

  Retrieves a named value that is stored by the current user from the Maconomy database.
- function maconomy::userValuesExists(name)

  Returns `true` iff the named value exists for the current user in the Maconomy database.
- function maconomy::deleteUserValue(name)

  Deletes the named value for the current user from the Maconomy database.
- function maconomy::deleteAllUserValues(name)

  Deletes all values of the same name for all users from the Maconomy database.
- procedure maconomy::setCommonValue(name, value)

  Stores or updates a named system-wide value in the Maconomy database. The value can be read, changed, and deleted by any user.
- function maconomy::getCommonValue(name)

  Retrieves a named system-wide value from the Maconomy database.
- function maconomy::commonValuesExists(name)

  Returns `true` iff the named system-wide value exists in the Maconomy database.
- function maconomy::deleteCommonValue(name)

  Deletes the named system-wide value from the Maconomy database.

Examples

The following examples illustrate how to manipulate M-Script values in the database.

Use User Values

An example application called the Customer Portal stores user-specific preferences using the Value Storage Interface. For each user it stores the maximum number of rows that that user wants to have displayed in a table somewhere in the application. This value is named `customerPortalMaxRows`.

Somewhere in the application the user has can enter the appropriate value for this parameter, so the application has the user's choice stored in the variable `maxRows`. To store the value in the database, the application uses the procedure `setUserValue`. This also overwrites any previous value entered by the user.

```
// Name of the value holding max. no.
of rows. var maxRowsValueName =
"customerPortalMaxRows";

...
// Function for getting user's max. rows
selection. function
readUserMaxRowsSelection()

{
  ...
}

...
// Get user input.

var maxRows = readUserMaxRowsSelection();

...
// Store user selection in database.
maconomy::setUserValue(maxRowsValueName,
maxRows); maconomy::commit();
```

Somewhere else you want to retrieve the user's preference. If no value has been stored yet, the application provides a default value.

```
// Default value for max. rows if no user
selection. var defaultMaxRows = 5;

...
//Retrieve user's choice.
var valueReturn = maconomy::getUserValue(maxRowsValueName);
var currentMaxRows = valueReturn.ok
                      ? valueReturn.value // User's choice.
                      : defaultMaxRows;   // Default value.
```

If the value `4` is stored in the database, the return value `valueReturn` from `getUserValue` is an object that has the following contents:

```
{
  ok    :
  true,
  value : 4
}
```

Use Common Values

In the Customer Portal the administrator can change the maximum number of rows that SQL calls should return. The application reads this value before executing SQL calls and uses it as the second parameter to `sqlRestricted`. Because all users should use this value, the application stores it as a common value named `customerPortalSqlMaxRows`.

First you store the common value while you are logged in as administrator.

```
// Name of the value holding max. no. of SQL rows.
var sqlMaxRowsValueName = "customerPortalSqlMaxRows";
...
// Get the value entered by the
administrator. var maxSqlRows = ...;
// Store the value as a common value.
maconomy::setCommonValue(sqlMaxRowsValueName,
maxSqlRows); maconomy::commit();
```

Note that any user can change the value. Therefore, it is up to the application to control which users are allowed to change the value.

Now another user logs in and performs an SQL call. The common value is read from the database immediately before performing the SQL call (by calling `sqlRestricted`).

```
// Get user name and password and perform a
login. var userName = ...;
var password = ...;
maconomy::login(userName, password);
...
// Read max. no of SQL rows to fetch stored by admin.
var maxRows = maconomy::getCommonValue(sqlMaxRowsValueName);
// Execute SQL statement.
var statement = "..."; // Some SQL statement.
var sqlReturn = maconomy::sqlRestricted(statement,
                                        maxRows.val
                                        ue, null);
```

Dialog Interface

The M-Script Maconomy API must give access to read, update, insert, and delete data in the Maconomy database. However, this needs to be done in a controlled way to avoid API calls corrupting the data in the Maconomy database.

The SQL Interface was described earlier in this document. This section described how to read data from the Maconomy database. The SQL Interface also includes subroutines for modifying, inserting, and deleting data using the functions `sqlUpdate`, `sqlInsert`, and `sqlDelete`, respectively. These functions can do almost anything with the relations that they operate on. Consequently, the set of relations on which these functions are allowed to operate is very limited. This means that you need another way to modify data in Maconomy, and this is exactly the purpose of the Dialog Interface.

The Dialog Interface gives full access to Maconomy using the dialog model. This means that data is read, updated, inserted, and deleted as if it were done through dialog windows in the Maconomy client.

Dialog Access Control

Access to the M-Script Maconomy API is granted through two server add-ons. With add-on 62 (M-Script Reporting) provides access to `maconomy::sql`, `maconomy::analyze` and related functions. Add-on 63 (M-Script API) gives access to server-side functions for interfacing with the Maconomy Dialog Machine.

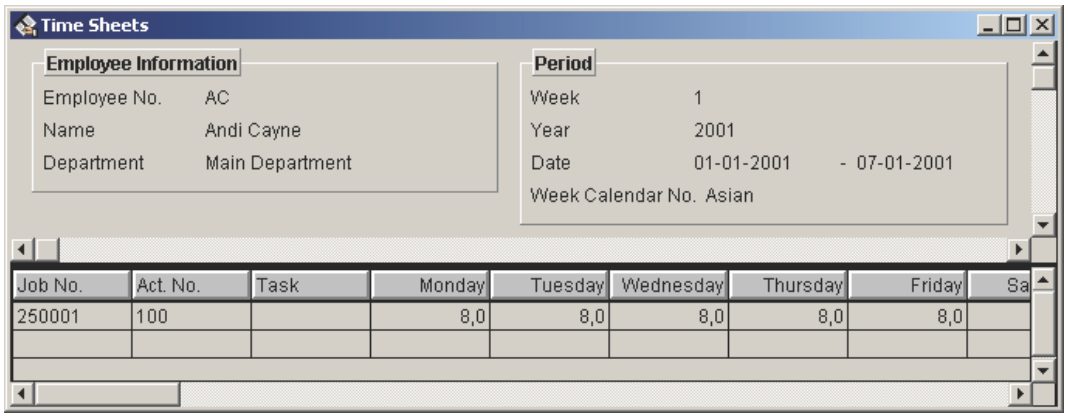
Starting with M-Script version 5.0, add-on 63 is supplemented with a dialog access file on the server. This file contains information about which dialogs can be accessed through the M-Script Maconomy API. This file is tied to a specific server registration number and is protected by an encryption scheme. Whenever a dialog is accessed in an unscrambled script through the M-Script Maconomy API, the dialog list is checked, and if the specified dialog is not in the list, the operation is denied. The dialog access list is not considered when running scrambled scripts.

The Maconomy Dialog Model

When working with data in dialog windows in the Maconomy client, a user unknowingly communicates with Maconomy through the dialog model. This model defines and restricts which operations can be performed on Maconomy data. This section attempts to provide a good understanding of the dialog model so that you are better able to take advantage of the Dialog Interface.

Anatomy of a Dialog

All operations in the dialog model are done through a Maconomy dialog. It is important to define the terms that this section uses when describing dialogs. The following figure shows a part of the Time Sheets dialog.



Upper and Lower Pane and Pane Types

The dialog contains two panes: the upper pane and the lower pane. The upper pane is a card pane, which means that data in this pane is placed freely in islands. The lower pane is a table pane, which means that data in this pane is arranged in a table that has columns and rows. A dialog like this is called a card/table dialog. The Dialog Interface also supports the card dialog, in which there is only one pane, the card pane.

Labels and Fields

The card pane contains labels and fields.

The texts that are displayed by the labels are static—that is, they never change. Examples of labels in this dialog's card pane are **Employee No.** and **Week**. The data that is taken from the Maconomy database is shown in fields, such as the data AC, Andy Cayne, and 2001.

In the table pane the labels are simply the column headings, and the fields are the cells of the table.

Only fields can be accessed by the Dialog Interface. The reason for this is that labels are really only static presentation data that is defined in layouts. Because the Dialog Interface does not know about Maconomy's MDL layouts, labels are also unknown to the Dialog Interface.

Key Fields

To tell Maconomy which data to display in a dialog, some fields in both panes are defined to be key fields. The set of all key fields in a pane is known as the key of the pane. The key of every entry in the Maconomy database must be unique.

In the Time Sheets dialog, the key fields of the upper pane are the fields **EmployeeNumber** and **PeriodStart**. The **EmployeeNumber** field is the field to the right of the label Employee No., and in the figure, it has the value AC. The **PeriodStart** field is the field just to the right of the label Date, and it has the value "01-01-2001."

(You cannot see the field names in the figure because these names are invisible to the user of the Maconomy client.)

Dialog Operations

The Dialog Interface can perform a number of operations on a dialog. These operations correspond almost one-to-one with operations that a user can perform using the Maconomy client.

These operations are:

- Open — The dialog is opened and initialized. This is the first operation that can be performed, and it must be performed before any other operation.
- Get — Data is fetched from the Maconomy database to the dialog.

  Opening a dialog window, such as Time Sheets, in the Maconomy client corresponds to performing an open operation immediately followed by a get operation.
- Update — After a get operation has been performed, the data in the dialog can be changed and updated in the Maconomy database. This is called an update operation.

  Changing data in an open dialog window and then pressing Enter corresponds precisely to performing an update operation.
- New — Creating an entry in the Maconomy database is a two-step process. First a new operation must be performed. This operation creates an empty and initialized entry.

  Selecting one of the New … menu items from the Index menu of the Maconomy client corresponds precisely to performing a new operation.
- Put — After creating an empty entry using the new operation and changing the data in the empty entry, you must save it in the database. This is called a put operation.

  Pressing Enter from the Maconomy client after selecting one of the New … menu items in the Index menu corresponds precisely to performing a put operation.
- Delete — Deleting data from the Maconomy database is done by performing a delete operation.

  Selecting one of the Delete … menu items from the Index menu of the Maconomy client corresponds precisely to performing a delete operation.
- Action — You can perform actions on a dialog. These actions are context-dependent. This means that the set of actions differs from dialog to dialog, and the set of enabled actions depends on the actual data that is in the dialog.

  Selecting any of the menu items from the Action menu of the Maconomy client corresponds precisely to performing an action operation.
- Initialize and Execute — Print dialogs are not handled in exactly the same was as normal dialogs are. Instead of updating data and performing actions on the dialogs, they are simply initialized to get default data and then executed to generate the actual print.
- Close — When the dialog is no longer used, it should be closed. This frees resources that are used to keep track of the dialog state. This operation is called close and is the last operation that can be performed on a dialog.

  Closing a dialog window in the Maconomy client corresponds precisely to performing a close operation.

Field Names

Danish characters are not allowed in field names.

Subroutines

The Dialog Interface provides the following subroutines:

- function maconomy::dialogOpen(dialogName [, modifiers])

  Opens a named dialog. The return value is a dialog ID which must be used as the first parameter to all other Dialog Interface subroutines. The optional parameter modifiers defines various special modes the dialog can be opened in.

- function maconomy::dialogGetDef(dialogId [,modifiers])

  Returns the definition of the dialog. The dialog definition includes field types for all fields, listing of key fields in both panes, and so on.

- function maconomy::dialogGet(dialogId, upperKey)

  Fetches and returns data from the specified dialog window based on an upper pane key.

- function maconomy::dialogGetFirst(dialogId)

  Fetches and returns the first record in the specified dialog window.

- function maconomy::dialogGetNext(dialogId)

  Fetches and returns the next record in the specified dialog window.

- function maconomy::dialogGetLast(dialogId)

  Fetches and returns the last record in the specified dialog window.

- function maconomy::dialogGetPrevious(dialogId)

  Fetches and returns the previous record in the specified dialog window.

- function maconomy::dialogRead(dialogId, upperKey, upperFields, lowerFields)

  Fetches and returns read-only data from the Maconomy database based on an upper pane key.

- function maconomy::dialogUpdateUpper(dialogId, record)

  Updates an upper pane dialog entry in the Maconomy database.

- function maconomy::dialogUpdateLower(dialogId, record, {rowNumber | key})

  Updates a lower pane dialog entry—that is, a table pane row—in the Maconomy database.

- function maconomy::dialogNewUpper(dialogId)

  Creates an empty upper pane dialog entry, which can be modified and used in a call to `dialogPutUpper` that follows immediately after this call.

- function maconomy::dialogPutUpper(dialogId, record)

  Puts a new upper pane dialog entry in the Maconomy database. The function`dialogNewUpper` must be called immediately before this function is called.

- function maconomy::dialogNewAndPutUpper(dialogId, record)

  Creates and puts an upper pane dialog entry in the Maconomy database. This functions combines the actions of the `maconomy::dialogNewUpper` and`maconomy::dialogPutUpper` functions.

- function maconomy::dialogNewLower(dialogId, rowNumber)

  Creates an empty lower pane dialog entry—that is, a table pane row—which can be modified and used in a call to `dialogPutLower` that follows immediately after this call.

- function maconomy::dialogPutLower(dialogId, record)

  Puts a new lower pane dialog entry—that is, a table pane row—in the Maconomy database. The function `dialogNewLower` must be called immediately before this function is called.

- function maconomy::dialogDeleteUpper(dialogId [, navigate])

Deletes the current upper pane entry along with all lower pane entries.

- function maconomy::dialogDeleteLower(dialogId, {rowNumber | key})

  Deletes the specified lower pane row of the current dialog entry.

- function maconomy::dialogAction(dialogId, action, rowNumber)

  Performs the specified action on the current dialog entry. The row number is used by very few actions. For other actions it is ignored.

- function maconomy::dialogGetPrint(dialogId)

  Return a handle for the print dialog associated with the specified dialog.

- function maconomy::dialogPrintThis(dialogId)

  Performs the equivalent of the “Print This…” command in the Maconomy client on the current dialog entry.

- function maconomy::dialogInit(dialogId)

  Initializes a print dialog. This function is similar to `maconomy::dialogGet()` and implies entering default data into the dialog and otherwise making ready for generating the print using `maconomy::dialogExecute()`.

- function maconomy::dialogExecute(dialogId, record)

  Executes a print dialog and generates a print based on the data entered in the record. This function is similar to `maconomy::dialogUpdateUpper()` except that it only applies to print dialogs.

- procedure maconomy::dialogClose(dialogId)

  Closes the dialog and frees all of the resources used to keep track of the dialog state.

- function maconomy::dialogGetState(dialogId)

  Returns the current internal state of the dialog state machine.

- function maconomy::dialogSearchOpen(searchSpec)

  Defines a foreign key search handler for a specified column in a dialog. A search handler can be used for getting static information about a foreign key search, and for getting the actual rows associated with the foreign key search.

- procedure maconomy::dialogSearchClose(searchHandler)

  This procedure releases resources used by the search handler given by the parameter`searchHandler`.

- function maconomy::dialogSearchGetDef(searchHandler)

  Get information about the foreign key search, defined by the search handler given by the parameter `searchHandler`.

- function maconomy::dialogSearchBind(searchHandler, searchBind)

  Bind field selection and restrictions to a search handler, and return a query handler that can be used for getting the actual rows associated with a foreign key search.

- function maconomy::setWarningReply(yes)

  Tells the Dialog Interface what to reply to application warnings that occur during execution of one the above operations. The default behavior is to answer “yes” to all warnings.

- function maconomy::fileSend(istream)

Creates a file on the server whose contents is read from the stream `istream`. A file handle of type `string` is returned. This is passed to other file-handling functions when referring to the file that is created.

- function maconomy::fileSendData(fileData)

  Creates a file on the server whose contents will be the string `fileData`. A file handle of type `string` is returned. This is passed to other file handling functions when referring to the created file.

- procedure maconomy::fileEnqueue(fileHandle, [fileName, [MIMEType]])

  The server file identified by `fileHandle` is queued for the next dialog operation. More than one file can be queued, but the following dialog operation must require as many input files as have been queued.

- function maconomy::fileGet(fileHandle, ostream)

  Retrieves a server file and writes it on the specified output stream. The server file can be a file generated by a dialog operation as well as a file sent to the server by calling `fileSendData`.

- function maconomy::fileGetData(fileHandle)

  Retrieves the contents of a server file. This can be a file generated by a dialog operation as well as a file sent to the server by calling `fileSendData`.

- procedure maconomy::fileDelete(fileHandle)

  Deletes a file on the server. See also fileDelete.

- function maconomy::fileGetHandles()

  Returns an array containing all file handles known in the current M-Script session.

- function maconomy::printGetPdf(printHandle, ostream)

  Converts a print on the server to PDF and writes it on the specified output stream.

- function maconomy::printDelete(printHandle)

  Deletes a print on the server.

- function maconomy::dialogMoveLowerBefore(dialogId, sourceKey, destinationRefKey)

  Moves a lower pane row to the position just above another lower pane row.

- function maconomy::dialogMoveLowerAftere(dialogId, sourceKey, destinationRefKey)

  Moves a lower pane row to the position just below another lower pane row.

- function maconomy::dialogMoveLowerFirstChild(dialogId, sourceKey, destinationRefKey)

  Moves a lower pane row so that it becomes the first child row of another lower pane row. Only valid if the table pane uses a tree structure.

Dialog Data in Session

A login session must be established before calling any other API subroutine. In addition, before establishing a login session by calling `maconomy::login`, an M-Script session must exist. (Recall that an M-Script session is created by calling the built-in M-Script function `newsession`.)

The subroutines in the Dialog Interface store data in the M-Script session. This data is not directly available to the M-Script programmer but is used to keep track of the state of each open dialog.

The data that is stored in the session by the Dialog Interface is:

- The current data in the dialog.
- High-level locks acquired by each dialog. (High-level locks prevent one user from modifying data in a window used by another user.)
- The current state of the dialog in the dialog state machine.
- References to files that are stored on the server. These can be files that are generated by a dialog operation, as well as a file sent to the server by calling `fileSendData`.

Only the first item in this list has direct impact on the M-Script programmer. It means that the Dialog Interface keeps an internal copy of the data that is received calling any of the Dialog Interface subroutines.

Error Handling

This description of the dialog API's error handling applies to scripts that are marked `#version 6` or higher.

The dialog API handles errors by throwing exceptions that you must catch to handle them gracefully. These exceptions are all of the type `error::stdlib::maconomy` or specializations of it such as `error::stdlib::maconomy::login`.

Note that the current transaction is usually rolled back when an error occurs during the execution of a Dialog Interface function.

Old-Style Error Handling

This description of the dialog API's error handling applies only to scripts that are marked `#version 5.1` or lower.

If an error occurs while executing a Dialog Interface function one of the following may happen:

- A run-time error occurs.
- The function returns an error code as part of the return value.

The following sections describe under which circumstances these two possibilities are used.

- Run-time error — If the error could have been avoided by the programmer, the Dialog Interface terminates M-Script with a run-time error. For example, if a value of the wrong type, such as a string value instead of an int value, is passed to a Dialog Interface function, the result is a run-time error.
- Error code — If the error is data-dependent, the Dialog Interface function returns an error code. For example, if a dialog entry cannot be created because the entry already exists, the Dialog Interface returns an error code.

Note that the current transaction is rolled back when an error occurs during execution of a Dialog Interface function.

Callbacks

A callback occurs when Maconomy needs more information during the execution of an operation in the dialog model. Examples of callbacks that most users have experienced when working with the Maconomy client are:

- Maconomy needs the user to select an input file.
- An output file is generated, and Maconomy needs to know where to save the file.
- A printout is generated.

- A warning is issued—for example, when creating an order for a customer and the customer 's credit has been exceeded. The user must click "OK" or "Cancel."

M-Script does not support callbacks. This means that all of these operations are not supported.

However, functionality exists to prepare for handling certain types of callbacks in advance:

- Warnings can be handled by the function `setWarningReply`, which allows the M-Script programmer to provide a default reply to give when a warning occurs.
- Requests for input files can be handled by sending the file contents to the server before performing the operation that requires the input file. Use the subroutines `fileSendData` and `fileEnqueue` for this purpose.
- If an operation creates an output file, the file contents can be retrieved from the server after the operation has been performed. Use the subroutine `fileGetData` for this purpose.

The Dialog State Machine

The dialog model imposes restrictions on the order of operations on a dialog. For example, it is not possible to perform a put operation without performing a new operation first. These restrictions are enforced by the dialog state machine.

A state machine is simply a device that keeps track of a state. From this state the state machine knows which operations are valid and which are not. Any open dialog is always in one of the following states:

- Initial — When a dialog has just been opened by calling `dialogOpen`, and no data is available, the dialog is in the Initial state. A dialog also ends up in this state if an error occurs during a dialog model operation.
- Exist — When data has been fetched from the server, a dialog is in the Exist state. A dialog is also in the exist state when a dialog entry has been created by performing a new operation and a put operation.
- NewUpper — When a new operation has been performed in the upper pane of a dialog, the dialog is in the NewUpper state. From this state, the new entry can be saved by performing a put operation on the upper pane. The only other operation that can be performed is a get operation, which cancels the new entry.
- NewLower — When a new operation has been performed in the lower pane of a dialog, the dialog is in the NewLower state. This state is completely analogous to the NewUpper state. The only operations that can be performed from this state are put (on the lower pane) and get.

  The dialog state machine is described by the following table. This table shows which subroutines can validly be called in which states. It also shows what the state is after calling each subroutine. Subroutines that are not included in this table are not restricted by the dialog state machine and can be called at any time. When an error occurs during a call to one of the subroutines in this table, the dialog state is Initial after the call.

Dialog State Before	Valid Subroutines	Dialog State After
Initial	`dialogGet`	Exist
	`dialogGetFirst`	Exist
	`dialogGetLast`	Exist

Dialog State Before	Valid Subroutines	Dialog State After
	dialogNewUpper	NewUpper
	dialogInit	Exist
Exist	dialogGetFirst	Exist
	dialogGetLast	Exist
	dialogGetPrevious	Exist
	dialogGetNext	Exist
	dialogUpdateUpper	Exist
	dialogUpdateLower	Exist
	dialogMoveLowerBefore	Exist
	dialogMoveLowerAfter	Exist
	dialogMoveLowerFirstChild	Exist
	dialogNewUpper	NewUpper
	dialogNewLower	NewLower
	dialogDeleteUpper	Initial
	dialogDeleteLower	Exist
	dialogAction	Exist
	dialogPrintThis	Exist
	dialogExecute	Initial
NewUpper	dialogPutUpper	Exist
	dialogGet	Exist
	dialogNewUpper	NewUpper
NewLower	dialogPutLower	Exist
	dialogGet	Exist
	dialogNewUpper	NewUpper
	dialogNewLower	NewLower

The following are examples of how to read this table:

- The first line in the table says that after opening the dialog by calling `dialogOpen`, the dialog is in the Initial state.
- The third line says that calling `dialogNewUpper` from the Initial state changes the state to NewUpper. The rows 12, 13, and 14 indicate that the only subroutines that can be called from this state are `dialogPutUpper`, `dialogGet`, and `dialogNewUpper`. (A call to `dialogGet` or `dialogNewUpper` from the NewUpper state will cancel creation of a dialog entry.)
- Because `dialogPutLower` is not listed in the Exist state, it cannot be called from this state.
- Because `dialogGetDef` is not listed in the table at all, it can always be called.

Print Dialogs

Print dialogs do not use the same dialog state machine as normal dialogs. Print dialogs have only two states—the initial state and the exist state. The initial state is entered when the dialog is opened, and the exist state is entered after a call to `dialogInit()`.

Examples

The Dialog Interface is by far the most complex part of the M-Script Maconomy API. To demystify it the following sections provide a range of examples that show how to perform the most common operations. The return values from the Dialog Interface functions are also very complex, so these are also discussed in the examples.

The examples in this section should be considered as one long example. This means that in each example it is assumed that you have read all of the previous examples in the section.

Getting the Dialog Definition

All that an M-Script programmer needs to know about a dialog is its internal name. When a programmer knows this, all other (relevant) information about the dialog can be retrieved using the function `dialogGetDef`.

This example opens the Time Sheets dialog and retrieves information about that dialog.

```
// Open the "Time Sheets" dialog.

// Please note: The internal name is
"TimeSheets". var dialogId =
maconomy::dialogOpen("TimeSheets");

// Get and print the dialog definition of this
dialog. var dialogDef =
maconomy::dialogGetDef(dialogId);
dumpvalue(dialogDef); print("\n");

// Close the dialog.
maconomy::dialogClose(di
alogId);
```

Before doing anything with the dialog you must open it. This is done by calling `dialogOpen`. This function takes the internal name of the dialog and returns a dialog ID that is a `string`. This dialog ID is used as the first parameter to almost all other Dialog Interface subroutines.

The easiest way to find the internal name of a dialog is to use the Maconomy client:

- Open the Window Layouts window.
- Select **Find Window** from the Find menu.
- Search for the dialog under its external name in the Window Name column.
- When you find the dialog, its internal name can be read from the Internal Name column in the Find window. Note that the internal window type must be Dialog Window.

Following this procedure you find that the internal name of the Time Sheets dialog is TimeSheets (with no space between the two words.)

To get the dialog definition you call `dialogGetDef` with `dialogId` as the only parameter.

After you print the information you close the dialog, because you are not going to use it anymore.

Remember to close a dialog when it is not used anymore. Failing to do so may cause problems for other users of the Maconomy system, because high-level locks are not freed. In addition, some resources that are associated with the dialog are never freed from the session.

Reading the Dialog Definition

The sample code in the previous section prints out the dialog definition for the Time Sheets dialog. The return object is very complex. Actually the output is more than 2,200 lines long. Consequently, it is not included in its entirety here. The following briefly describes the parts of this return object.

The overall structure of the return object from `dialogGetDef` is:

```
{
  actions         : [...],
  printName       :
  "Print_TimeSheets",
  printLayoutName : "Standard",
  type            :
  "card/table", name
  : "TimeSheets", title
  : "Time Sheets",
  highLevelLock   : true,
  readAccess      : true,
  newAccess       :
  true, updateAccess
  : true, deleteAccess
  : true, upperPane
  : {...}, lowerPane
  : {...}
}
```

This return object provides the following information about the dialog:

- A list of the available actions in this dialog.
- The name and layout of the printout used by this dialog.
- The type, internal name, and external name of this dialog.
- Information that indicates whether the dialog uses high-level locking or not.

- The access rights to the dialog for the current user.
- A description of the upper and lower panes of the dialog.

The following provides a closer look at the pane description for the upper pane that is contained in the property `upperPane`:

```
{
  type         :
"card", columnIndex
: [...], columns
: {...},

relationName :
"TimeSheetHeader", keyDef
: [...],

newSpelling  : "New Time
Sheet", name         :
"TimeSheetHeader", title
: "Time Sheet"

}
```

The description of the upper pane contains the following information:

- The type of the pane.
- A description of all of the fields in the pane. This information—contained in the properties `columnIndex` and `columns`—corresponds to the information that is contained in the properties of the same name in the return objects of the SQL Interface and the Analyzer Interface.
- Information about which relation delivers data to this pane and what the key fields are.
- The external description of a new entry in the pane.
- The internal and external names of the pane.

Finally, the following field descriptions are contained in the property `columns`.

The property is an object where the properties are named after the fields. Each property contains a description of the field whose name it bears.

The following shows the description of the first field in the upper pane. This property is `dialogDef.upperPane.columns.EmployeeNumbe` and its value is:

```
{
  index      : 0,

  title      :
  "EmployeeNumber", type
  : "string",

  kind       :
  "database",
  secret     :
  false, mandatory
  : true, openNew
  : true,
  openUpdate :
  false
```

```
}
```

You can recognize a part of the information from the description of the columns of an SQL result:

- The field's index.
- The external name of the field.
- The field's type.

You can also see some additional information:

- Whether the field is a database field or a variable field.
- Whether the field is a secret field, such as a password field whose contents should not be displayed on a screen.
- Access rights of the field.

Print Dialogs

The `dialogGetDef` function also works for print dialogs, but with a reduced amount of information, because not all of the data is relevant to print dialogs (for instance, the access rights).

Getting Data

To retrieve data from the dialog you use the function `dialogGet`.

```
// Open the "Time Sheets" dialog.
var dialogId = maconomy::dialogOpen("TimeSheets");
// Build a key for the entry
to get. var key = {
            EmployeeNumber : {value :
            "AC"}, PeriodStart :
            {value : 01.01.2001}
          };
// Get a time sheet entry.
var dialogReturn = maconomy::dialogGet(dialogId, key);
```

You must open the dialog again, because you closed it before. Note that it is not necessary to close the dialog after calling `dialogGetDef`. It was simply closed it in the preceding example to make clear the point that a dialog should be closed when not used anymore. However, if you plan to call `dialogGet` and other Dialog Interface functions, the dialog should not be closed until it is not used anymore.

Almost all Dialog Interface functions use the same return object. Consequently, looking closely at the return value from `dialogGet` is a very good idea. The object looks like this:

```
{
  status     :
  {...},
  dialogData :
  {...}
}
```

The property `status` indicates the status of the function call: Did the operation succeed? Were any errors returned? The property `dialogData` contains the data that is fetched from the Maconomy database. This section looks into the latter of the two properties.

The `dialogData` property looks like this:

```
{
  actionsEnabled      :
  {...},       readOnly
  :   false,   upperPane
  :   {...},   lowerPane
  : {...}
}
```

From the first two properties you receive information about which of the dialog actions are enabled and if you received a read/write or a read-only copy of the data.

The data that is retrieved from the Maconomy database is contained in the last two properties. The following is a closer look at the `upperPane` property:

```
{
  access : {
  newAccess    : true, updateAccess : false, deleteAccess : true,
  findAccess   : true
           },
  rows   :
  [...]
}
```

The Maconomy application can dynamically control which operations can be performed on a dialog. This information is received in the `access` property. You can see that the upper pane cannot be modified because `updateAccess` is `false`. This corresponds to the knowledge of the Time Sheets window in the Maconomy client. After a time sheet has been created, data can only be entered in the lower pane.

The `rows` property is an array of entries that are fetched from the database. Because the upper pane is always a card pane, there is exactly one entry in `rows`. This array entry is an object whose properties are named after the fields in the dialog—just like the `columns` property described previously

The following illustrates the object `rows[0]`:

```
{
  EmployeeNumber :
      {
        value : "AC"
      },
      PeriodStart :
      {
        value : 01.01.2001
      },
      ...
    }
```

These are the values that you expected for the upper pane and the same values that were displayed in the earlier figure.

Updating a Lower Pane Row

As the example figure shows, 8 hours of work is entered for Monday in the selected week. You want to change the number of hours to 9 for that day. For that purpose you use the `dialogUpdateLower` function.

```
// We only need the changed value.
var changed = { NumberOfDay1 : {value : 9.0} };
// Update row no. 0.
dialogReturn = maconomy::dialogUpdateLower(dialogId,
                                           changed,
                                                   0);
// Commit
changes.
maconomy::c
ommit();
```

You only need to send changed values to Maconomy, so you create a simple record that contains only the value that you want to change: the hours for Monday. When you call `dialogUpdateLower` you must tell it which lower pane row to update—row number 0 in this case. Finally, you commit the changes to the database.

The data that is returned from `dialogUpdateLower` reveals that you have changed the hours that were entered for Monday:

```
{
  status :
  {
    ok
    : true,
    messages
    : []
  },
  dialogData :
  {
    actionsEnabled
    :         {...},
    readOnly
    :         false,
    upperPane
    :         {...},
    lowerPane :
    {
      access
      :
      {...},
      rows :
```

```
            [
              {
                ...
                NumberOfDay1 : {value
                : 9.0}, NumberOfDay2
                : {value : 8.0},
             NumberOfDay3 : {value : 8.0},
             ...
           }
        ]
     }
  }
}
```

Inserting a New Lower Pane Row

You need to enter another time sheet line for the employee “AC” in the existing time sheet. The line should enter hours for the job “250002” and the activity “110.” You want to enter 4 hours of work on Saturday.

```
// Create a new row.
maconomy::dialogNewLower(dialogId,
1);

// Enter the desired
data. var newRec = {
                JobNumber      : {value : "250002"},
                ActivityNumber : {value : "110"},
                NumberOfDay6   : {value : 4.0} //
                Saturday.
              };
// Put the new row.
dialogReturn = maconomy::dialogPutLower(dialogId, newRec);
maconomy::commit();
```

The function `dialogNewLower` actually returns the same kind of object as the other Dialog Interface functions that you have seen so far. It contains all of the data in the dialog, including the new empty, initialized row. However, in this example you do not need these values, so you simply discard the return value. Remember that the Dialog Interface stores its own copy of the return value in the M-Script session. This means that the Dialog Interface still knows that you are about to put a new row number 1 in just a moment. (The first row is number zero, so row number 1 is the second row in the table.)

Now, you create a record that contains only the values that you want to set to be different from the values of the empty row. All that you want to change is the job number, the activity number, and the number of hours for Saturday.

After that you put the new row into the system and commit the changes.

The return value shows that a new second row has been entered in the table:

```
{
```

```
status :
{
  ok       :
  true,
  messages : []
},
dialogData :
{
  ...
  upperPane :
  {...},
  lowerPane :
  {
    ...
        r
        o
        w
        s
        :

        [
          {...}, // First row.
          {      // Second new row.
            ...
            JobNumber:{value :
            "250002"},
            ActivityNumber:{value
            :"110"},

            ...
            NumberOfDay5:{val
            ue:0.0},
            NumberOfDay6:{val
            ue:4.0},
            NumberOfDay7:{val
            ue:0.0},

            ...
          }
        ]
      }
    }
  }
```

Deleting a Lower Pane Row

As it turns out, the row that you just entered should never have been entered. Therefore, you can use the function `dialogDeleteLower` to remove it.

```
// Delete row number 1. (The second row.)
dialogReturn = maconomy::dialogDeleteLower(dialogId, 1);
maconomy::commit();
```

A look at the return value shows that the time sheet now again has only one lower pane row:

```
{
  status
  : {...},
  dialogData
  :
  {
...
upperPane :
    {
      .
      .
      .
      r
      o
      w
      s
      :
      [
        {
          ...
          Submitted : {value : false}, // Not submitted.
          ...
        }
      ]
    },
    lowerPane :
    {
      .
      .
      .
      r
      o
      w
      s
      :
      [
```

```
            {...} // Only one row.
          ]
        }
      }
    }
```

Performing an action

The time sheet is now ready for submission. You do this by calling the dialogAction function.

```
// Submit the time sheet.
dialogReturn = maconomy::dialogAction(dialogId, 0);
maconomy::commit();
// We're done: Close the
dialog.
maconomy::dialogClose(dialogI
d);
```

The second parameter to `dialogAction` is the action number. In this case you want to perform action number zero. you could also choose to give the internal name of the action. From the result of calling `dialogGetDef` you can learn that action number zero has the external name "Submit Time Sheet" and the internal name "Deliver_TimeSheet." What you need is the internal name, which would make the call to `dialogAction` look like this:

```
// Submit the time sheet.
dialogReturn = maconomy::dialogAction(dialogId,
                                      "Deliver_TimeSheet
                                      ");
maconomy::commit();
```

In any case you close the dialog because you are not going to use it anymore. The return value from the call to `dialogAction` is:

```
{
  status     :
  {...},
  dialogData :
  {
    ...
    upperPan
    e :
    {
      ...
      row
      s :
      [
        {
          ...
          Submitted : {value : true}, // Submitted!
          ...
```

```
        }
      ]
    },
    lowerPane :
    {
      ...
      row
      s :
      [
        {...} // Still only one row.
      ]
    }
  }
}
```

You see that the time sheet is now submitted.

Using Input and Output Files

Certain dialog operations (mostly “actions”) can use input files or produce output files. An example of the usage of input and output files are the actions “Import Text” and “Export Text” of the dialog Events in the Contact Management module.

When a dialog action needs an input file, the Maconomy client prompts the user to provide this file. When a dialog action needs to produce an output file, the Maconomy client prompts the user for a location of the file. However, this is handled a bit differently when using the Dialog Interface.

```
// Open the "Events" dialog and get an entry.
var dialogId = maconomy::dialogOpen("Contacts");
maconomy::dialogGet(dialogId,
                    {ContactNumber : {value:"40272"} });
// Perform the action "Export Text".
var dialogData = maconomy::dialogAction(dialogId,
"Export_Text");
// Retrieve the data from the file generated by the
action. var fileHandle =
dialogData.status.files[0];
var fileData = maconomy::fileGetData(fileHandle);
print("fileData is:\n^1\n", fileData);
// Delete the file on the
server.
maconomy::fileDelete(file
Handle);
```

The preceding M-Script code opens the Events dialog (whose internal name is “Contacts”), fetches the event identified by the key “40272,” and performs the action “Export_Text” on this dialog entry. This action generates an output file that can be retrieved by calling the function `fileGetData`. This function takes a file handle as its first parameter and returns the file data as a string.

The file handle is taken from the files property of the dialog status. This property is an array that contains file handles for all of the files that are produced by the dialog operation. It is the responsibility of the programmer to remember to delete the file after retrieving it from the server. The file is deleted by calling `fileDelete`.

The output generated by this code is:

```
            fileData is:
ContactLine:Change      40272           #
$$                                      T
This is the first line,                 E
and this is the second.                 X
$$                                      T
```

The file data that is received corresponds to the lines in the lower pane of the dialog. The exact meaning of the file data is not relevant in this context. What is interesting is how the file data is retrieved from the Maconomy server.

To input data back into the application you perform the “Import Text” action. You use the same data that you received, but you add another line of text.

```
            // Change file data.
            fileData = 'DATA';
            ContactLine:Change
            40272  #TEXT

            $$

            This is the first line,

        and this is the
        second. We add a
        third line

        $$
        DATA

        // Create file on server and enqueue it for the next

        // dialog operation.

        fileHandle = maconomy::fileSendData(fileData);

        maconomy::fileEnqueue(fileHandle);

        // Perform the action "Import Text" and commit changes.
        dialogData = maconomy::dialogAction(dialogId,
        "Import_Text"); maconomy::commit();

        // Delete the server file.
        maconomy::fileDelete(fileHandl
        e);
```

First you add another line to the string in `fileData`. Before performing the “Import Text” action you must send the file to the server and queue it for the next dialog operation. This is done by calling the subroutines `fileSendData` and `fileEnqueue`. When performing the action the queued file is used as input.

After performing the action you must delete the file on the server by calling `fileDelete`.

Using the Documents Archive

The following example illustrates how a file can be stored in the document archive on the server:

```
            #version 15
```

```
newsession();
maconomy::login("Administrator", "123456");
  // Open document archive window
var id = maconomy::dialogOpen("DocumentArchives");
  // Create a new empty archive
maconomy::dialogNewUpper(id);
maconomy::dialogPutUpper(id,{DocumentArchiveNumb
er:
                                {value:"MScript"}});
  // Now open and send the file we want to store in the archive
  // (always open as binary file; mode="rb")
var f = file::open("readme.pdf", "rb");
var handle = maconomy::fileSend(f);
  // Create a lower row for the file
... maconomy::dialogNewLower(id, 0);
  // Enqueue it for the next operation ...
  // (here we supply file name and MIME-
type). maconomy::fileEnqueue(handle,
                     "readme.pdf",
                     "application/pdf
                     ");
    // and put it in the
  database.
  maconomy::dialogPutLower(
  id, {});
  maconomy::commit();
  maconomy::logout();
  deletesession();
```

Assuming that the preceding example has been executed without errors you can now retrieve the PDF file again with the following code:

```
#version 15
newsession();
maconomy::login("Administrator", "123456");
  // Open document archive window
var id = maconomy::dialogOpen("DocumentArchives");
  // Get our archive
var data = maconomy::dialogGet(id, {DocumentArchiveNumber:
                                    {value:"MScript"}});
  // Get file name (and prepend "new_")
var filename = "new_"
```

```
                                    +
                        data.dialogData.low
                              erPane
                          .rows[0].DocumentName.value;
    // Open the file for local
  storing. var f =
  file::open(filename, "wb");
    // Export our file (stored in row 0)
  data = maconomy::dialogAction(id, "Export_Document",0);
    // Retrieve the file
  maconomy::fileGet(data.status.files
  [0], f);
  maconomy::logout();
  deletesession();
```

Using a Print Dialog

The following example opens a print dialog, retrieves its definition, initializes it, adds user-defined data, and executes it. The output of the program is a PDF file that is stored on the web server.

```
  #version 15
    // Create a session and login
  newsession();
  maconomy::login("Administrator",
  "123456");
    // Open print dialog and get its
  definition print("Open dialog 'Print
  Order List'.\n");
  var id = maconomy::dialogOpen("Print Order List");
  print("Get definition.\n");
var definition = maconomy::dialogGetDef(id);
  // Obs: getting the definition is optional
  // and only showed as an illustration of its use
  // Initialize the dialog and execute
it print("Initialize it.\n");
var data = maconomy::dialogInit(id);
print("Execute it.\n");
data = maconomy::dialogExecute(id, {FROMCUSTOMER:
                                     {value:"31003100
                                   "}, TOCUSTOMER:
                                     {value:"31003100"}});
  // Retrieve the PDF file from the Maconomy server and
  // store it on the web server.
```

```
var f = file::open("print.pdf", "wb");
maconomy::printGetPdf(data.status.prints[0],
f); file::close(f);

maconomy::dialogClose(id);

maconomy::logout();

deletesession();
```

Improving Performance

In cases where the return value for the dialog functions can be ignored, or used as read-only data, it is possible to improve performance by using the dialog API in a read-only mode. This does not mean that access is read only; `dialogPutUpper`, `dialogUpdateUpper` and so on may still be used, but the data that is returned from the API is for reading only.

This read-only mode is enabled for each dialog with a second optional modifier parameter to `dialogOpen`. The modifier is an object which may contain various optional properties—in this case you use `readonlyData`, which should be set to `true`.

The data that is returned from the dialog API is now meant for reading only; it resembles the normal dialog data, but a closer look at the type of the records (entries in the row data) reveals that the records are of an internal M-Script type `DialogRecord`. These records cannot be distinguished from normal objects, as long as they are used for reading only.

Example:

```
#version 15

// Read-only data requires version 5 or higher.

newsession();

maconomy::login("Administrator", "123456");

  // Make dialog return values read-only for
efficiency var id =
maconomy::dialogOpen("TimeSheets",

  { readonlyData:true });

  // Get a record

    var data = maconomy::dialogGet( id,

                                    {PeriodStart:{value:29.10.
                                     2001},
                                     EmployeeNumber:{value:"12
                                     95"} });

    var record = data.dialogData.upperPane.rows[0];

      // What is the record type?

    print("Record type = ^1\n", typeof(record));

      // It is possible to read data as usual

    print("Employee number: ^1\n", record.EmployeeNumber.value);

    dumpvalue(record);

      // But we cannot change it

    new record.EmployeeNumber.mySetting = "something"; // Error!!!

    record.EmployeeNumber.value = 0; // Error!!!
```

```
maconomy::logout();
deletesession();
```

RGL Report Interface

The M-Script API includes an interface to RGL reports that enables a programmer to run RGL reports on the server and retrieve the generated prints as PDF files.

Subroutines

- maconomy::rglOpen(rglReportName)

  Opens a named RGL report. The return value is some status information and a report identifier that must be used as the first parameter to all other RGL Interface subroutines.

- maconomy::rglInit(rglId)

  Fetches and returns default data for the report dialog.

- maconomy::rglExecute(rglId, data)

  Executes the report and returns the generated print and data file.

- maconomy::rglClose(rglId)

  Closes the report and frees all of the resources that are used to keep track of the report state.

- maconomy::rglGetDef(rglId)

  Returns the definition of the report dialog. The dialog definition includes field types for all fields and access rights information, and so forth.

- maconomy::rglList()

  Returns a list of all known RGL reports.

- maconomy::rglList(groupName)

  Returns a list of all known RGL reports in a specific group.

RGL Reporting Setup

RGL reports are stored in files named `*.grn` and must be placed in the `Analyze` directory of the Maconomy server. In addition to this, the server must also be notified about the reports. This is done by adding one line of text for each report in the file `StandAloneList.txt` in the server's `Definitions` directory. Each line should follow this format:

```
RGLReport; internal-name; external-name; group; filename
```

That is—a semicolon-separated record identifying the report's internal and external name, its associated group, and its filename. The first field identifies the record as being an RGL report. The internal name is the name by which the report is identified when using `maconomy::rglOpen`. The external name is the name that an end-user should see. The group is used to combine a set of reports into a logical group. The last field is the exact name of the report file, as it is stored in the `Analyze` directory. Whitespace before and after the semicolons is removed.

The `StandAloneList.txt` file must begin with the text "StandAloneList 1" to identify it as the standalone program list (RGL reports are a special case of Maconomy's standalone MSL programs). Example:

```
StandAloneList 1

RGLReport; MScript1; M-Script 1; M-Script; MScript1.grn
RGLReport; MScript2; M-Script 2; M-Script; MScript2.grn
RGLReport; MScript3; M-Script 3; M-Script; MScript3.grn
```

RGL Reporting Compared to the Dialog API

The RGL report interface has been carefully designed to work in the same way as the dialog API. This means that you get the same requirements to and use of the session and login information, the same data layout, and the same error handling.

The only exception to this is the return value of `maconomy::rglOpen`, which returns an object instead of a dialog identifier. The returned object contains both some status information and the RGL dialog ID.

Examples

The following is an example that illustrates how RGL reports can be executed and how the resulting data can be retrieved from M-Script:

```
#version 15
try // Check for all errors
{
    // Open RGL report
    // - here we use the internal name from the
    //   stand alone list.
     var rgl = maconomy::rglOpen("MScriptTest1");
    // Get RGL dialog
  identifier var rglId =
  rgl.id;
    // Get default report dialog data
  var rglData = maconomy::rglInit(rglId);
    // Print some of the
  data print("Field
  'DateFirst' = ^1\n",
       rglData.dialogData.upperPane.rows[0].DATEFIRST.value);
    // Change 'myField' to 42 and execute the report
  rglData = maconomy::rglExecute(rglId, {myField:{value:42}});
    // Now get the print and store it in a
  PDF file. var p =
  rglData.status.prints[0];
  var f = file::open("print.pdf", "wb"); // Binary output!
  maconomy::printGetPdf(p, f);
  file::close(f);
    // And do the same with the generated
  data file. p = rglData.status.files[0];
  f = file::open("file.txt", "wb");
maconomy::fileGet(p,
f); file::close(f);
maconomy::rglClose(rglI
d);
}
```

```
catch error::stdlib::maconomy(e)
  dumpvalue(e);
```

Pop-Up Interface

The Dialog Interface can return Maconomy pop-up values. These values are handled by the Popup Interface. Each pop-up value has two components:

- The pop-up type, which is a string.
- The pop-up ordinal value, which is an integer.

An example of a pop-up value is (printed using the built-in M-Script procedure `dumpvalue`)

```
maconomy::CurrencyType(3)
```

This is a pop-up value of the pop-up type "CurrencyType," and the value has ordinal value 3 in this type. Pop-up ordinal values start at zero, so this is the fourth literal in the "CurrencyType" pop-up type.

A pop-up value can also be a `null` value, which corresponds to a blank popup entry in the Maconomy client.

Working with Pop-Ups

M-Script has built-in primitives that allow the handling of pop-up type values.

Use the `typeof` operator to get the type name of a pop-up value. To get the ordinal value, you type cast the pop-up value to `int`. Using the pop-up value from the preceding example, you apply the `typeof` operator and perform a type cast to `int`:

```
var popupValue = maconomy::popup("CurrencyType",3);

var typeName = typeof(popupValue);

var ordinalValue = int(popupValue);

var ordinalValue2 = ordinal(popupValue); // Same
as above print("Type name is:     ^1\n",
typeName);

print("Ordinal value is: ^1\n", ordinalValue);
```

The output from these program lines is:

```
Type name is:     maconomy::CurrencyType

Ordinal value is: 3
```

Pop-ups can be converted to strings as illustrated here:

```
  // Create a popup

var p = maconomy::popup("CurrencyType",0);

  // Implicit string
conversion print("popup
= ^1\n", p);

  // Explicit string
conversion var ps =
string(p);

print("popup string = ^1\n", ps);
```

The output is:

```
popup = CurrencyType(0)
```

```
popup string = CurrencyType(0)
```

The conversion can be used for debugging and does not require a call to the Maconomy server.

The actual image of the pop-up can be obtained using `maconomy::popupTitle()`.

Subroutines

To create, decode, and manipulate pop-up values, the Popup Interface provides the following subroutines:

- function maconomy::popup(typeName, ordValue)

  Returns a pop-up value of the given type and with the given ordinal value. A more elaborate name for this function is `maconomy::createPopupLiteral()`.

- function maconomy::popupNull(typeName)

  Returns a `null` pop-up value of the given type. A more elaborate name for this function is `maconomy::createPopupNullLiteral()`.

- function maconomy::popupTitle(…)

  Returns the display title of a pop-up literal. The literal can be identified by its type name and ordinal value or by a pop-up value. A dialog identifier is needed to handle pop-ups of the type “LayoutNameType” since the literals of this pop-up depends on the current dialog. A more elaborate name for this function is `maconomy::getPopupLiteralTitle()`.

- function maconomy::popupTitles(…)

  Returns an array containing all the display titles of all literals in the given pop-up type.

  The pop-up type can be identified by the type name or can be derived from a pop-up value. A more elaborate name for this function is `maconomy::getPopupLiteralTitles()`.

Example

Assume that you have received a pop-up value as part of the return value from a call to `maconomy::dialogGet` in the Dialog Interface. You want to display this value to the user, so you need the display title.

```
var popupValue = ...; // CurrencyType(3);
// Get and print display title.
var title = maconomy::popupTitle(popupValue);
print("Display title is: ^1\n", title);
```

The output is:

```
Display title is: USD
```

This is the way to display the value to a user as a read-only value. However, if the user needs to be able to change the value you must create a drop-down list. To get the list of texts to appear in this drop-down list, you can use the function `popupTitles`. This example does not create a drop-down list; it simply prints the titles:

```
// Get titles of all literals in this type.
var titles = maconomy::popupTitles(popupValue);
// Print them.
for (var i in titles)
```

```
  print("Literal ^1: ^2\n", i, titles[i]);
```

The output is:

```
Literal
0: DKK
Literal
1: SEK
Literal
2: NOK
Literal
3: USD
Literal
4: GBP
Literal
5: NLG
Literal
6: DEM
Literal
7: FRF
Literal
8: BEF
Literal
9: FIM
Literal
10: CHF
Literal
11: EUR
Literal
12: CDN
Literal
13: ITL
Literal
14: ESP
```

When the user has selected one of these values, you use the function `popup` to create the pop-up value that should be sent back to the Maconomy server.

```
// Get user
selection. var
index = ...;

// Create new popup value.

var typeName = typeof(popupValue);

var newVal = maconomy::popup(typeName, index);
```

If the user selected the blank entry, a `null` literal is created using the function `popupNull`.

Accessing Print Layouts using Pop-Ups

The following example opens a print dialog and specifies a specific print layout when executing it:

```
newsession();

maconomy::login("Administrator", "123456");

  // Open a print dialog
```

```
    var id = maconomy::dialogOpen("Print Job Report");
    var data = maconomy::dialogInit(id);
      // Get and print all available print layouts
    var titles = maconomy::popupTitles(id, "LayoutNameType");
    print("Titles
    = \n");
    dumpvalue(tit
    les);
    print("\n");
  // Create a "LayoutNameType" popup
  // for selecting a specific layout
var p = maconomy::popup("LayoutNameType",0);
  // Print the popup as "LayoutNameType(0)"
  // (this does not print its title)
print("popup = ^1\n", p);
  // Get the title of this layout
  // - here we use the dialog ID to achieve
it var title = maconomy::popupTitle(id, p);
print("Title = ^1\n", title);
  // Select the specified print layout
  // when executing the print
data = maconomy::dialogExecute(id, {LAYOUTNAME:{value:p}});
```

Preprocessing Interface

The preprocessing functionality allows M-Script code sections to be dependent on add-ons as well as system parameters and system information. For instance, you can use this functionality to show a message if a certain system parameter has been marked, or to run a subroutine if a certain add-on has been installed.

Subroutines

To preprocess M-Script code, the following subroutines are available:

- function maconomy::getSystemParameter(relationName, fieldName)

  Tests whether a given system parameter is set or not on the server.

- function maconomy::hasAddOn(integer)

  Tests whether an add-on number is enabled on the server.

- function maconomy::preprocess(output, input)

  Allows M-Script code sections to be dependent on add-ons as well as system parameters and system information.

- function maconomy::readvalue(input)

High-Level Lock Interface

In the Dialog API, Maconomy automatically sets high-level locks on relations and releases them again when the dialog is closed. The idea behind “high-level locking” is to prevent two users from making changes to the same database record at the same time. However, if you are using custom relations in Maconomy (that is, relations that are specified in MOL files), the M-Script programmer must implement manual high-level locking. The subroutines in this interface can be used to implement a protocol such as the following for accessing a custom relation:

- Before making changes to a record, try to acquire a high-level lock.
- If successful: make the changes and release the lock.
- Else: do not make changes to the record; instead, try again, or show the user an error message.

A lock for a record cannot be acquired when the record is already locked.

There is a timeout-value for high-level locks (it defaults to five minutes). This means that even though a user has obtained a lock, it may have timed out and been obtained by another user. To follow the locking-protocol, the first user must be able to recognize that the lock is gone.

For more information about MOL and custom relations, see the MOL Language Reference.

Subroutines

To implement a manual high level lock routine, the following subroutines are available:

- function maconomy::hllLock(relationName, key, comment)

  Tries to obtain a lock for the record that is specified by `key` and `relationName`.
- function maconomy::hllRenewLock(lockId)

  Tries to renew the lock that is specified by `lockId`.
- function maconomy::hllUnlock(lockId)

  Releases a high-level lock that is specified by `lockId`.

Example

The following example of the use of the high-level lock interface assumes an external relation as defined in the following MOL file:

```
<MOL 1.0>
  <Object abc_testRelation>
    .Salary       :Amount
    .Componentid  :String  :Key+
    .Target       :Integer :Key+
    .SalaryGroup  :String  :"Salary level"
  <End Object>
```

The following M-script first obtains a lock on a record and then releases it:

```
try
{
  var lockId = maconomy::hllLock("abc_testRelation",
```

```
    { Componentid:"hello",
    Target:53 }, "this is a
    lock comment");
}
catch error::stdlib::maconomy::hllLock::locked(e)
{
 ...
}

.

.

.


t

r

y
{
  var wasLockRetaken = maconomy::hllRenewLock(lockId);
}
catch error::stdlib::maconomy::hllLock::lockLost(e)
{
  ...
}
maconomy::hllUnlock(lockId);
```

Subroutine Reference

This section gives a complete, alphabetically ordered reference description of all of the subroutines in the M-Script Maconomy API.

Each subroutine is described in a standardized way in its own section. The first section of this section is a sample entry that illustrates how to read a subroutine reference entry. Please read this entry first to get the most out of this section.

Sample Entry

Prototype

Every subroutine is described by its prototype. The prototype tells:

- Whether the subroutine is a procedure (that is, a subroutine that has no return value) or a function (that is, a subroutine that has a return value),
- What the name of the subroutine is, and
- Which parameters the subroutine takes.

A prototype for a function taking one parameter could look like this:

```
function maconomy::dialogOpen(dialogName)
```

Sometimes a function has a few mandatory parameters but accepts an unlimited number of parameters. In this case "..." is used to denote zero or more parameters. A prototype for such a function could look like this:

```
function maconomy::sql(statement, maxRecs, firstRec, ...)
```

This function takes at least three parameters, but there is no upper limit to the number of parameters that can be passed.

If a subroutine can be called in more than one way, more than one prototype is given.

API Part

Each subroutine belongs to a specific part of the API, such as the Dialog Interface.

License

Most subroutines require that specific add-ons are enabled on the server. This could, for example, be M-Script Reporting (Add-on 62). If more than one license requirement is stated, it means that any one of these licenses gives access to the subroutine.

Description

A brief description of the subroutine is provided.

Parameters

All subroutine parameters are described. For each parameter, its name, type, and a description of its meaning are provided. The type simple is used for a parameter that can be of any simple type—that is, not object or array. If either of two types could be used, | is used to separate the types.

A parameter description could look like this:

Name	Type	Description
`statement`	string	Any SQL select statement to be executed on the Maconomy server. The statement may contain placeholders `^1`, `^2`, …. These placeholders are replaced by the arguments following `firstRec`. The placeholder `^1` is replaced by the first argument following `firstRec`, `^2` is replaced by the second argument following `firstRec`, etc.
`maxRecs`	int | null	The maximum number of records the function call should return. If this parameter is omitted or `null` is passed all records are returned.
`firstRec`	int | null	The number of the first record to return. The index of the first record is zero. If this parameter is omitted or `null` is passed the function call returns records starting from the first available record.
`...`	simple	Placeholder strings `^1`, `^2`, … in `statement` are replaced by the corresponding of the remaining parameters. The placeholder `^1` is replaced by the first of these arguments, `^2` by the second, etc.

Return Value

If the subroutine is a function, the return value is described here.

Context

This field specifies the permitted contexts for the function or procedure. See the M-Script Language Reference section.

Throws

This field is relevant when the API is invoked from scripts with version 6 or later, and exceptions have not been disabled.

If the subroutine has previously returned status information in the form of an ok property or a dialogStatus object, it may now throw an exception in situations where the ok property is false. The value of the exception that is thrown is described here.

In addition to this, all API subroutines can throw exceptions of the type messageError if there are simple errors such as a parameter mismatch.

Required State

This section applies to Dialog Interface subroutines only. It descirves in which state the dialog must be to allow calling the described subroutine.

State After Call

This section applies to Dialog Interface subroutines only. It describes which state the dialog is in after calling the described subroutine. This information applies only if the operation succeeds. If the operation fails, the dialog goes into the Initial state.

Remarks

This section provides information about how to use the subroutine. It also describes common pitfalls and special considerations to be taken.

Example

Finally, a small code example of how to use the function is provided. An example could look like the following:

```
var dialogId = maconomy::dialogOpen("TimeSheets");
if (dialogId != null)
{
  print("Opened dialog 'Time Sheets'.\n");
  maconomy::dialogClose("TimeSheets");
}
```

Analyze

Prototype

```
function maconomy::analyze(fileName)
function maconomy::analyze(fileName, layoutName)
function maconomy::analyze(fileName, layoutName, rowSelection)
```

API Part

Analyzer Interface

License

M-Script Reporting (add-on 62)

Description

Executes an Analyzer report on the Maconomy server.

Parameters

Name	Type	Description
	string	The name of the Analyzer report file. This file must be located on the server. Analyzer files have the extension .grf.
`layoutName`	string	The name of the Analyzer layout to use. If this parameter is omitted or if it is the empty string, the first layout in the report is used.
`rowSelection`	string	A string that describes changes in the row selections already completed in the selected Analyzer report layout. This parameter consists of key/value pairs in the format of a query string: `key1=value1&key2=value2...` A key must be named “rv<n>,” “rv<n>.1,” or “rv<n>.2.” The first two kinds of key names designate the first field of the nth row in the row selection of the Analyzer report. The third kind designates the second field of the nth row in the row selection.

Return Value

The return value is an object of type `analyzeReturn`.

Context

Standalone, server command-line

Throws

If the call fails, an exception with value `messageError` is thrown.

Remarks

At least one layout must exist for the Analyzer report to be executed. The layout that is selected using the parameter `layoutName` must exist.

Layouts are created using an ordinary Maconomy client. After the layout has been saved, the `.lay` file must be moved to the Maconomy server’s `Analyze` folder for the relevant application.

Note that executing certain Analyzer reports can consume a large amount of system resources on the Maconomy server. Executing such reports might result in a major performance degradation of the entire Maconomy system.

Example

```
var analyzeRes = maconomy::analyze("TimeSht.grf",
                                                    "", // Use first
                                                    layout.
                                                    "rv1.1=1080&rv1.2=
                                                    1080");
```

analyzeValidate

Prototype

```
function maconomy::analyzeValidate(reportName)
```

API Part

Analyze Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Validates the name of an Analyzer report.

Parameters

Name	Type	Description
`reportName`	string	Analyzer report name to be verified.

Return Value

A `validateReturn` object

Context

Standalone, server command-line

autoCommit

Prototype

```
function maconomy::autoCommit(yes)
```

API Part

Transaction Control Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Turns automatic commit on or off.

Parameters

Name	Type	Description
`yes`	bool	If this parameter is `true` automatic commit is enabled. If it is `false` automatic commit is disabled.

Return Value

The function returns the state of automatic commit before the call. If `true` is returned automatic commit was enabled before the call. Otherwise `false` is returned.

Context

Standalone, server command-line

Remarks

When automatic commit is disabled all changes to the database must be explicitly committed or rolled back by calling `maconomy::commit` or `maconomy::rollback` respectively.

When automatic commit is enabled all changes to the database are immediately committed. There is no need to call `maconomy::commit` to commit changes to the database.

Automatic commit is disabled by default.

Example

```
// Make sure every change is committed automatically. var
                oldVal = maconomy::autoCommit(true);
```

commit

Prototype

```
procedure maconomy::commit()
```

API Part

Transaction Control Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Commits changes to the database.

Context

Standalone, server command-line

Remarks

If automatic commit is disabled (which it is by default) it is necessary to commit all changes to the database before they take effect.

To turn on automatic commit call `maconomy::autoCommit`.

Example

```
// Open dialog 'Users' and get record for the user 'User'. var
               dialogId = maconomy::dialogOpen("Users");
               var dialogRet =
                   maconomy::dialogGet(dialogId,
                                       {NameOfUser : {value : "User"}});
               print("Getting user ok?: ^1\n", dialogRet.status.ok);
               // Delete the entry.
               dialogRet = maconomy::dialogDeleteUpper(dialogId);
               print("Deleting user ok? ^1\n", dialogRet.status.ok);
               // Commit the change to the
               database. maconomy::commit();
```

commonValueExists

Prototype

```
function maconomy::commonValueExists(name)
```

API Part

Value Storage Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Checks whether a named common value exists in the Maconomy database.

Parameters

Name	Type	Description
`name`	string	The name of the value whose existence should be checked.

Return Value

The return value is of type `bool`. The function returns `true` *iff* a common value with the specified name exists.

Context

All

Remarks

This function is provided for performance reasons since `getCommonValue` can provide the same information. However, calling this function does not transfer the value (if it exists) from the server

to the M-Script interpreter. If the value is a complex compound value then the overhead can be significant. Consequently, this function should be used instead of `getCommonValue` if only the question of value existence is interesting.

Call the function `userValueExists` to check for the existence of a user-specific value.

Example

```
var valueName = "customerPortalSettings";
                if (maconomy::commonValueExists(valueName))
                {
                  ...
                }
```

createPopupLiteral

See `popup`.

createPopupNullLiteral

See `popupNull`.

deleteAllUserValues

Prototype

```
function maconomy::deleteAllUserValues(name)
```

API Part

Value Storage Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Deletes a named user value for all users.

Parameters

Name	Type	Description
`name`	string	The name of the user value that should be deleted for all users.

Return Value

The return value is an object of type `valueDeleteReturn` that indicates whether any values were deleted or not.

Context

Standalone, server command-line, custom action

Remarks

This function deletes a named value for all existing users. Call the function `deleteUserValue` to delete a named value for the current user only.

Example

```
var valueName = "customerPortalUserPrefs";
                maconomy::deleteAllUserValues(valueName);
```

deleteCommonValue

Prototype

```
function maconomy::deleteCommonValue(name)
```

API Part

Value Storage Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Deletes a named common value from the Maconomy database.

Parameters

Name	Type	Description
`name`	string	The name of the common value to delete.

Return Value

The return value is an object of type `valueDeleteReturn` that indicates whether a value was deleted or not.

Context

Standalone, server command-line, custom action

Remarks

This function deletes a common value from the Database. Call the function `deleteUserValue` to delete a user specific value.

Example

```
var valueName = "customerPortalSettings";
                maconomy::deleteCommonValue(valueName);
```

deleteUserValue

Prototype

```
function maconomy::deleteUserValue(name)
```

API Part

Value Storage Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Deletes a named user value for the current user from the database.

Parameters

Name	Type	Description
`name`	string	The name of the user value to delete.

Return Value

The return value is an object of type `valueDeleteReturn` that indicates whether a value was deleted or not.

Context

Standalone, server command-line, custom action

Remarks

This function deletes a named user value for the current user only. Call the function `deleteAllUserValues` to delete a named value for all users. Call the function `deleteCommonValue` to delete a common (system-wide) value from the Maconomy database.

Example

```
var valueName = "customerPortalUserPrefs";
                maconomy::deleteUserValue(valueName);
```

dialogAction

Prototype

```
function maconomy::dialogAction(dialogId, actionName) function
maconomy::dialogAction(dialogId, actionIndex) function
maconomy::dialogAction(dialogId, actionName, rowNo) function
maconomy::dialogAction(dialogId, actionIndex, rowNo)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Performs an action on a dialog.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`actionName`	string	The internal name of the action to perform.
`actionIndex`	int	The index of the action to perform. This is the index in the array property `actions` in the type `dialogDef`.
`rowNo`	int	The index of a lower pane row. Note the following remarks.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a The function returns an object of type `dialogReturn` exception of type `error::stdlib::maconomy::dialog::Action` is thrown.

Required State

Exist

State After Call

Exist

Remarks

Very few actions use the parameter `rowNo`—such as the Import Layout action of the Window Layouts window. If this parameter is not given, the result is the same as passing zero.

Example

```
// Submit the time sheet.
                var dialogReturn = maconomy::dialogAction(dialogId, 0);
                maconomy::commit();
```

dialogClose

Prototype

```
procedure maconomy::dialogClose(dialogId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Closes a dialog and frees all of the resources that are allocated for the dialog in the M-Script session.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Context

Standalone, server command-line, custom action

Remarks

This must be the last subroutine called when a dialog is no longer used.

Forgetting to call this function leads to the loss of resources in the current M-Script session for as long as the session occurs.

A more serious potential issue is that if the dialog holds a high-level lock for a specific dialog entry, this locks out other users for as long as it takes for the user to time out. The default time-out value is five minutes.

Example

```
// We're done: Close the dialog.
               maconomy::dialogClose(di
               alogId);
```

dialogDeleteLower

Prototype

```
function maconomy::dialogDeleteLower(dialogId, {rowNumber | key})
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Deletes a lower pane row in a dialog.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`rowNumber`	int	The index of the row to delete. The first row has index zero.
`key`	object	A key that matches at least one row in the lower pane.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Exist

State After Call

Exist

Remarks

This function can only be used for two-pane dialogs.

When specifying a row by key, the fields in each row are tested for equality against the fields that are specified in the key; thus, the key must contain “value” properties exactly as in the dialog record. The first row where all of these fields match is the target of the update operation.

Example

```
// Delete row number 1. (The second row.)
                dialogReturn = maconomy::dialogDeleteLower(dialogId, 1);
                maconomy::commit();
```

dialogDeleteUpper

Prototype

```
function maconomy::dialogDeleteUpper(dialogId [, navigate])
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Deletes the current dialog entry.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`
`navigate`	bool	Optional flag that indicates whether a navigation should be attempted after the deletion. If omitted it is assumed to have the value `false`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Exist

State After Call

Initial or Exist

Remarks

If `navigate` is set to `true` this function attempts to fetch the next record from the server. If the deleted record was the only record in the dialog, the navigation fails with a `RecordNotFound` error. Otherwise, the next or the previous record (if the deleted record was the last) is returned.

If `navigate` is set to `false` no new data is fetched from the server, and in this case there is no `dialogData` property in the return value, because the current entry has been deleted.

Example

```
// Delete the current entry and get the next.
                dialogReturn = maconomy::dialogDeleteUpper(dialogId, true);
                maconomy::commit();
```

dialogExecute

Prototype

```
function maconomy::dialogExecute(dialogId, record)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Executes a print dialog. This corresponds to pressing Enter in the Maconomy client when the print dialog is open. The record contains the values that are supplied by the user.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`record`	object	The new data for the print. This property is an object of type `record`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Exist

State After Call

Initial

dialogGet

Prototype

```
function maconomy::dialogGet(dialogId, upperKey)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Fetches data for a specific dialog from the Maconomy database.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`upperKey`	object	The key identifying which dialog entry to fetch. The object is of type `record`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Initial, Exist, NewUpper, NewLower

State After Call

Exist

Remarks

The parameter `upperKey` is of type `record`. However, only the fields that are defined as key fields for the upper pane of the dialog need to have a value defined in this object.

Call the function `dialogGetDef` to get the list of key fields in the upper pane.

Example

```
// Build a key for the entry to get. var key
                = {
                            EmployeeNumber : {value :
                            "AC"}, PeriodStart : {value
                            : 01.01.2001}
                          }
                          ;
                // Get a time sheet entry.
                var dialogReturn = maconomy::dialogGet(dialogId, key);
```

dialogGetDef

Prototype

```
function maconomy::dialogGetDef(dialogId)
function maconomy::dialogGetDef(dialogId, modifiers)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Returns the static description of a dialog. This description includes the number and type of panes, field names, access rights, and so forth.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`modifiers`	object	Optional modifiers to the dialog. This is an object that contains one property for each modifier.

Return Value

The return value is an object of type `dialogDef`.

Context

Standalone, server command-line, custom action

Remarks

The `modifiers` object may contain the following properties:

- `getFavorites` — bool — Specifies whether favorite values should be returned as part of the dialog definition. Favorite values are described in the Maconomy product note on favorite values setup. Please note that including favorites adds an extra performance overhead, because it requires an M-Script to be run on the Maconomy server. The default value is false.

It is not necessary to call this function to access a dialog, but it enables an M-Script application not to assume any knowledge of a specific dialog before using it.

Example

```
// Get and print the dialog definition of this dialog. var
              dialogDef = maconomy::dialogGetDef(dialogId);
              dumpvalue(dialogDef); print("\n"");
```

dialogGetFirst

Prototype

```
function maconomy::dialogFirst(dialogId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Fetches data for the first record in a specific dialog from the Maconomy database.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Initial, Exist, NewUpper, or NewLower

State After Call

Exist

Example

```
// Get the first time sheet entry.
                var dialogReturn = maconomy::dialogGetFirst(dialogId);
```

dialogGetLast

Prototype

```
function maconomy::dialogLast(dialogId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Fetches data for the last record in a specific dialog from the Maconomy database.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Initial, Exist, NewUpper, or NewLower

State After Call

Exist

Example

```
// Get the last time sheet entry.
                var dialogReturn = maconomy::dialogGetLast(dialogId);
```

dialogGetNext

Prototype

```
function maconomy::dialogNext(dialogId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Fetches data for the record immediately after the current one in a specific dialog from the Maconomy database.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Exist, NewUpper, or NewLower

State After Call

Exist

Remarks

A dialog record must have already been fetched when this function is called. If the current record is the last in the dialog, a `RecordNotFound` error is issued.

Example

```
// Get the next time sheet entry.
                var dialogReturn = maconomy::dialogGetNext(dialogId);
```

dialogGetPrevious

Prototype

```
function maconomy::dialogPrevious(dialogId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Fetches data for the record immediately before the current one in a specific dialog from the Maconomy database.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Exist, NewUpper, or NewLower

State After Call

Exist

Remarks

A dialog record must have already been fetched when this function is called. If the current record is the first in the dialog, a `RecordNotFound` error is issued.

Example

```
// Get the previous time sheet entry.
                var dialogReturn = maconomy::dialogGetPrevious(dialogId);
```

dialogGetPrint

Prototype

```
function maconomy::dialogGetPrint(dialogId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Opens the print dialog associated with the specified dialog.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Return Value

This function returns a dialog ID identifying the print dialog.

Context

Standalone, server command-line, custom action

Throws

If there is no print associated with the given dialog a `messageError` exception is thrown. If the associated print cannot be opened, the behavior is defined by `dialogOpen`.

Remarks

Not all dialogs have an associated print dialog. Calling dialogGetPrint on such a dialog results in an error. The print dialog name can be found as the printName property in the dialogDef object.

Example

```
// Open a dialog:
                var id = maconomy::dialogOpen(dialogName);
                // Open the associated print dialog:
                var print_id = maconomy::dialogGetPrint(id);
                // Initialize and execute the print dialog:
```

See Using a Print Dialog.

dialogGetState

Prototype

```
function maconomy::dialogGetState(dialogId)
```

API Part

Dialog Interface

License

No license required

Description

Returns the current state of the dialog state machine.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Return Value

The return value is of type `string`. Possible return values are “Initial,” “Exist,” “NewUpper,” and “NewLower.”

Context

Standalone, server command-line, custom action

Example

```
// Print the current state.
                print("state = ^1", maconomy::dialogGetState(dialogId));
```

dialogInit

Prototype

```
function maconomy::dialogInit(dialogId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Initializes a parameter (print) dialog for use with `dialogExecute()`.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Return Value

The function returns an object of type `dialogReturn` that contains the default values defined by the Maconomy application.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Initial

State After Call

Exist

MoveLowerBefore

Prototype

```
function maconomy::dialogMoveLowerBefore(dialogId, key, destinationReferenceKey)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Moves a lower pane row to the position just above another lower pane row.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`key`	object	The key that identifies the lower pane row to move. This property is an object of type `record`.
`destinationReferenceKey`	object	The key that identifies the row that determines where to move the row. This property is an object of type `record`

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown. If the move operation is disabled for the table relation, a `dialogError` of type `OperationDisabled::MoveLower` is thrown.

Required State

Exist

State After Call

Exist

Remarks

This function can only be used if the table relation has a position context and a line number field.

Example

```
// Move row with key theKey before the row with key destKey. var
               dialogReturn =

                  maconomy::dialogMoveLowerBefore(dialogId, theKey, destKey);

               maconomy::commit();
```

dialogMoveLowerAfter

Prototype

```
function maconomy::dialogMoveLowerAfter(dialogId, key, destinationReferenceKey)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Moves a lower pane row to the position just below another lower pane row.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`key`	object	The key that identifies the lower pane row to move. This property is an object of type `record`.
`destinationReferenceKey`	object	The key that identifies the row that determines where to move the row. This property is an object of type `record`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action.

Throws

In case of failure, a `dialogError` exception is thrown. If the move operation is disabled for the table relation, a `dialogError` of type `OperationDisabled::MoveLower` is thrown.

Required State

Exist

State After Call

Exist

Remarks

The function can only be used if the table relation has a position context and a line number field.

Example

```
// Move row with key theKey after the row with key destKey. var
               dialogReturn =

                  maconomy::dialogMoveLowerAfter(dialogId, theKey, destKey);

               maconomy::commit();
```

dialogMoveLowerFirstChild

Prototype

```
function maconomy::dialogMoveLowerFirstChild(dialogId, key,
destinationReferenceKey)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Moves a lower pane row to become the first child row of another lower pane row.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`key`	object	The key that identifies the lower pane row to move. This property is an object of type `record`.
`destinationReferenceKey`	object	The key that identifies the row of which the row identified by `key` should be made a child. This property is an object of type `record`.

Return Value

The function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown. If the move operation is disabled for the table relation, a `dialogError` of type `OperationDisabled::MoveLower` is thrown.

Required State

Exist

State After Call

Exist

Remarks

The function can only be used if the table relation has a position context and a line number field, and if the lower pane has a tree structure.

Example

```
// Move row with key theKey to become first child of row destKey. var
              dialogReturn =

                  maconomy::dialogMoveLowerFirstChild(dialogId, theKey,
                                          destKey);

              maconomy::commit();
```

dialogNewAndPutUpper

Prototype

```
function maconomy::dialogNewAndPutUpper(dialogId, record)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Creates and inserts a dialog entry in the Maconomy database.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`record`	object	The data for a new upper pane entry. This property is an object of type `record`.

Remarks

This function combines the operations of `dialogNewUpper` and `dialogPutUpper`.

The `record` object needs not contain values for all fields in the pane. Only fields that need to be changed—compared to the return value of `dialogNewUpper`—should be included.

Example

```
// Set the mandatory fields for the dialog. var
              rec = {

                          EmployeeNumber : {value
                          : "AC"}, WeekNumber
                          : {value : 5}

                        };

              // Create and put the new entry.

              var dialogReturn = maconomy::dialogNewAndPutUpper(dialogId,
              rec);

              maconomy::commit();
```

dialogNewLower

Prototype

```
function maconomy::dialogNewLower(dialogId, rowNumber)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Creates an empty and initialized lower pane row

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`rowNumber`	int	The index of the row to create. The first row has index zero. If the index of an existing row is passed, this row and all rows below it are moved down. A value of -1 can be used to add a row at the bottom of the table without having to calculate the actual row index.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Exist or NewLower

State After Call

NewLower

Remarks

The return value contains all data in the dialog including the new empty row. To insert the new row call the function `dialogPutLower`.

Be aware of performance when using this function. It is always more efficient to add a row at the bottom of the table, rather than inserting a row somewhere in the table.

This function can only be used for two-pane dialogs.

Example

```
// Create a new row.
                maconomy::dialogNewLower(dialogI
                d, 1);

                // Enter the desired
                data. var newRec = {

                                JobNumber      : {value : "250002"},
                                ActivityNumber : {value : "110"},
                                NumberOfDay6   : {value : 4.0} //
                                Saturday.

                            }
                            ;

                // Put the new row.

                dialogReturn = maconomy::dialogPutLower(dialogId, newRec);

                maconomy::commit();
```

dialogNewUpper

Prototype

```
function maconomy::dialogNewUpper(dialogId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Creates a dialog entry

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Initial, Exist, NewUpper, or NewLower

State After Call

NewUpper

Remarks

To insert the new dialog entry, call the function `dialogPutUpper`.

Example

```
// Create a new time sheet.
               maconomy::dialogNewUpper(dial
               ogId);

               // Change the mandatory
               fields. var rec = {
                           EmployeeNumber : {value :
                           "AC"}, WeekNumber     :
                           {value : 5}
                         }
                         ;
               // Put the new entry.
               var dialogReturn = maconomy::dialogPutUpper(dialogId, rec);
               maconomy::commit();
```

dialogOpen

Prototype

```
function maconomy::dialogOpen(dialogName)
function maconomy::dialogOpen(dialogName, modifiers)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Opens a dialog.

Parameters

Name	Type	Description
`dialogName`	string	The internal name of the dialog.

Name	Type	Description
`modifiers`	object	Optional modifiers to the dialog. This is an object that contains one property for each modifier.
`readonlyData`	bool	Improve performance by enforcing read-only return data.
`disableHLL`	bool	Disable high-level locking for this dialog. This modifier is for experts only and cannot be used without specific settings in the `mscript.ini` file in the server's `definitions` directory. Contact Maconomy Development for more information.
`IgnoreDialogAccessControl`	bool	Setting this property to `true` makes it possible to open dialogs to which the logged-in user has no access. This option can be used only if the script is stamped with the TAC-field `AllowIgnoreDialogAccessControl` set to `true`.

Return Value

If the dialog could be opened, the return value is of type `string` and is a unique ID that identifies this instance of the dialog.

If exceptions are disabled a `null` value is returned if the current user does not have access to the requested dialog.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `messageError` exception is thrown.

Required State

No state exists before the call.

State After Call

Initial

Remarks

The dialog name that is passed to this function is evaluated case-insensitively, and the characters "_" and " " (space) are considered equal.

If an invalid dialog name is passed to this function a run-time error occurs.

Access to dialog windows is controlled by a file in the `definitions` directory of the server.

The easiest way to find the internal name of a dialog is to use the Maconomy client:

- Open the Window Layouts window.
- Select **Find Window** from the Find menu.

- Search for the dialog under its external name in the Window Name column.
- When the dialog is found its internal name can be read from the Internal Name column in the "Find" window. Note that the internal window type must be Dialog Window.

This function allocates space in the current M-Script session that is freed by the procedure `dialogClose`. Consequently, it is important to remember to call `dialogClose` when the dialog is no longer used.

In addition, forgetting to call `dialogClose` may lock out other users.

Example

```
// Open the "Time Sheets" dialog.
                // Please note: The internal name is
                "TimeSheets". var dialogId1 =
                maconomy::dialogOpen("TimeSheets");

                // Do the same, but with read-only return data.

                var dialogId2 = dialogOpen("TimeSheets", { readonlyData:true });
```

dialogPrintThis

Prototype

```
function maconomy::dialogPrintThis(dialogId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Generates a print file from the current dialog record.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Exist

State After Call

Exist

Remarks

This function produces a print file on the server and returns the corresponding print handle in the `status.prints` array in the `dialogReturn` object. To obtain this print from the server as a PDF file, see `printGetPdf`.

Example

```
// Generate a print for the current record dialogReturn =
                maconomy::dialogPrintThis(dialogId);
                // Get the corresponding print handle
                var printHandle = dialogReturn.status.prints[0];
                // Write the PDF print to a file
                var f = file::open("print.pdf", "wb");
                dialogReturn = maconomy::printGetPdf(printHandle, f);
                file::close(f);
                // Remove the print from
                the server
                maconomy::printDelete(print
                Handle);
```

dialogPutLower

Prototype

```
function maconomy::dialogPutLower(dialogId, record)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Inserts a new lower pane row in a dialog.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.

Name	Type	Description
`record`	object	The data for new lower pane row. This property is an object of type `record`.

Return Value

The function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

NewLower

State After Call

Exist

Remarks

It is necessary to call the function `dialogNewLower` (section 14.30) before this function to enter the NewLower state.

The `record` object needs not contain values for all fields in the pane. Only fields that need to be changed—compared to the return value of `dialogNewLower`—should be included. However, it imposes no run-time overhead to pass all values.

This function can only be used for two-pane dialogs.

Example

```
// Create a new row.
               maconomy::dialogNewLower(dialo
               gId, 1);

               // Enter the
               desired data. var
               newRec = {
                              JobNumber      : {value :
                              "250002"}, ActivityNumber : {value
                              : "110"}, NumberOfDay6   : {value :
                              4.0} // Saturday.

                            }
                            ;

               // Put the new row.

               dialogReturn = maconomy::dialogPutLower(dialogId, newRec);

               maconomy::commit();
```

dialogPutUpper

Prototype

```
function maconomy::dialogPutUpper(dialogId, record)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Inserts a new dialog entry in the Maconomy database.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`record`	object	The data for new upper pane entry. This property is an object of type `record`.

Return Value

The function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

NewUpper

State After Call

Exist

Remarks

It is necessary to call the function `dialogNewUpper` before this function to get into the NewUpper state.

The `record` object needs not contain values for all fields in the pane. Only fields that need to be changed—compared to the return value of `dialogNewUpper`—should be included. However, it imposes no run-time overhead to pass all values.

Example

```
// Create a new time sheet.
                maconomy::dialogNewUpper(dial
                ogId);

                // Change the mandatory
                fields. var rec = {

                            EmployeeNumber : {value :
                            "AC"}, WeekNumber     :
                            {value : 5}

                          }
                          ;

                // Put the new entry.

                var dialogReturn = maconomy::dialogPutUpper(dialogId, rec);

                maconomy::commit();
```

dialogSearchBind

Prototype

```
function maconomy::dialogSearchBind(searchHandler, searchBind)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Bind field selection and restrictions to a search handler, and returns a query handler that can be used for getting the actual rows that are associated with a foreign key search window.

Parameters

Name	Type	Description
`searchHandler`	string	A search handler that is created by a call to the function `dialogSearchOpen`.
`searchBind`	object	Selection of fields and definition of restrictions. This property is an object of type `searchBind`.

Return Value

This functions returns a query handler that are to be used with the `maconomy::mql*` set of functions for data retrieval.

Context

All

Throws

The exception `error::stdlib::maconomy::dialogSearch::invalidHandler` is thrown if the search handler do not exists. The exception `error::stdlib::maconomy::mql::invalidHandler` is thrown if the query handler could not be created. Information about an error is available in the exception.

Remarks

The call does not execute the query. Always remember to call the `maonomy::mqlClose` at some point after a successful call to `maconomy::dialogSearchBind`.

Example

```
try  {
                var searchHandler = maconomy::dialogSearchOpen(...);
                var searchBind = {
                      columnIndexO
                         utput: [
                         "Employee
                         Number",
                         "Name1"
                      ],
                      restrictions : [
                        { columnName: "EmployeeNumber",
                        restriction:["1000..1010"] }
                      ]
                    };
        var mqlHandler = maconomy::dialogSearchBind(searchHandler,
                                                    searchBind);
                ... maconomy::mqlClose(mqlClose);
                maconomy::dialogSearchClose(searchH
                andler);
              }
              catch(e) {
                ...
              }
```

dialogSearchClose

Prototype

```
procedure maconomy::dialogSearchClose(searchHandler)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

This procedure release resources used by a search handler. Always remember to call `dialogSearchClose` when a search handler is no longer needed.

Parameters

Name	Type	Description
`searchHandler`	string	A search handler created by a call to the function `dialogSearchOpen`.

Context

All

Throws

The exception `error::stdlib::maconomy::dialogSearch::invalidHandler` is thrown if the search handler does not exist. Information about an error is available in the exception.

Example

```
try  {
        var searchHandler = maconomy::dialogSearchOpen(...);
        ...
        maconomy::dialogSearchClose(searchHan
        dler);
    }
    catch(e) {
        ...
    }
```

dialogSearchGetDef

Prototype

```
function maconomy::dialogSearchGetDef(searchHandler)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

This function gets information about the foreign key search that is associated with the search handler.

Parameters

Name	Type	Description
`searchHandler`	string	A search handler created by a call to the function `dialogSearchOpen`.

Return Value

The return value is an object of type `searchDef`.

Context

All

Throws

The exception `error::stdlib::maconomy::dialogSearch::invalidHandler` is thrown if the search handler does not exist. Information about an error is available in the exception.

Example

```
try  {
        var searchHandler = maconomy::dialogSearchOpen(...);
        var searchDef     =
        Maconomy::dialogSearchGetDef(searchHandler);
        ...
        maconomy::dialogSearchClose(searchHan
        dler);
    }
    catch(e) {
        ...
    }
```

dialogSearchOpen

Prototype

```
function maconomy::dialogSearchOpen(searchSpec)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

This function creates a foreign key search handler for a specified column in a dialog.

A search handler can be used for getting static information about a foreign key search, and for getting the actual rows that are associated with the foreign key search.

Parameters

Name	Type	Description
searchSpec	object	Selection of a column in a dialog, and specification of values of current record in the dialog. This property is an object of type searchSpec.

Return Value

The return value is a search handler that are to be used with the other maconomy::dialogSearch* functions.

Context

All

Throws

The exception error::stdlib::maconomy::dialogSearch::invalidHandler is thrown if the search handler could not be created. Information about an error is available in the exception.

Remarks

Always remember to call the maonomy::dialogSearchClose at some point after a successful call to maconomy::dialogSearchOpen.

Example

```
try  {
                  var searchSpec = {
                        dialogName    :
                        "Employees", pane
                        : "upper", columnName
                        : "CompanyNumber",
                        currentRecord : @{ EmployeeNumber : {value:"11"} }
                  };
                  var searchHandler = maconomy::dialogSearchOpen(searchSpec);
                  ...
                  maconomy::dialogSearchClose(searchH
                  andler);
               }
               catch(e) {
                  ...
               }
```

dialogRead

Prototype

```
function maconomy::dialogRead(dialogId, upperKey, upperFields, lowerFields)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Fetches read-only data for a specific dialog from the Maconomy database.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`upperKey`	object	The key that identifies which dialog entry to fetch. The object is of type `record`.
`upperFields`	array	An optional list of upper pane fields that should be returned.
`LowerFields`	array	An optional list of lower pane fields that should be returned.

Return Value

This function returns an object of type `dialogReturn`.

CXontext

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Initial, Exist, NewUpper, or NewLower

State After Call

Initial

Remarks

This function is a read-only version of `dialogGet`. In addition, it makes it possible to fetch only specific fields from the upper and lower panes, thereby reducing the amount of data that is passed over the network. Using `dialogRead` instead of `dialogGet` wherever possible could greatly reduce the resource needs for an M-Script program and speed up execution.

The `upperFields` and `lowerFields` arrays can be used to specify the names of the fields that should be returned. An empty array means “no fields,” while an omitted parameter or a `null` value means “all fields.”

The parameter `upperKey` is of type `record`.However, only the fields that are defined as key fields for the upper pane of the dialog need to have a value defined in this object.

Call the function `dialogGetDef` to get the list of key fields in the upper pane.

Example

```
// Build a key for the entry to get. var key
                = {
                            EmployeeNumber : {value :
                            "AC"}, PeriodStart : {value
                            : 01.01.2001}
                          }
                          ;
                // Build array of upper pane fields to be read.
                var upper = [ "EmployeeNumber", "PeriodEnd", "EmployeeName" ];
                // Get a time sheet entry with the specified upper pane data
                and
                // all lower pane rows.
                var dialogReturn = maconomy::dialogRead(dialogId, key, upper);
```

dialogUpdateLower

Prototype

```
function maconomy::dialogUpdateLower(dialogId, record, {rowNo | key})
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Updates an existing lower pane row.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`record`	object	The updated data for lower pane row. This property is an object of type `record`.
`rowNo`	int	The index of the row to update. The index of the first row is zero.
`key`	object	A key that matches at least one row in the lower pane.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Exist

State After Call

Exist

Remarks

The `record` object need not contain values for all fields in the pane. Only fields that need to be changed should be included. However, it imposes no run-time overhead to pass all values.

This function can only be used for two-pane dialogs.

When specifying a row by key, the fields in each row are tested for equality against the fields that are specified in the key; thus, the key must contain “value” properties exactly as in the dialog record. The first row where all of these fields match is the target of the update operation.

Example

```
// We only need the changed value.
                var changed = { NumberOfDay1 : {value : 9.0} };
                // Update row no. 0.
  dialogReturn = maconomy::dialogUpdateLower(dialogId,
                                             changed,
                                                  0);
                // Commit
                changes.
                maconomy::c
                ommit();
```

dialogUpdateUpper

Prototype

```
function maconomy::dialogUpdateUpper(dialogId, record)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Updates the upper pane of a dialog.

Parameters

Name	Type	Description
`dialogId`	string	Identifies the dialog. This ID is obtained by calling `dialogOpen`.
`record`	object	The updated data for upper pane. This property is an object of type `record`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

In case of failure, a `dialogError` exception is thrown.

Required State

Exist

State After Call

Exist

Remarks

The `record` object needs not contain values for all fields in the pane. Only fields that need to be changed should be included. However, it imposes no run-time overhead to pass all values.

Example

```
// Change the job name. var rec
                = {
                            JobName :
                            {
                              value : "M-Script Maconomy API Reference"
                            }
                          };
              // Update the upper pane.
              var dialogReturn = maconomy::dialogUpdateUpper(dialogId, rec);
              maconomy::commit();
```

dialogValidate

Prototype

```
function maconomy::dialogValidate(dialogName)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Validates the name of a dialog.

Parameters

Name	Type	Description
`dialogName`	string	Dialog name to be verified.

Return Value

A `validateReturn`object.

Context

Standalone, server command-line, custom action

Echo

Prototype

```
function maconomy::echo(value)
```

API Part

Login Session Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Returns the value passed to the function.

Parameters

Name	Type	Description
`value`	any	Any M-Script value.

Return Value

This function returns a deep copy of the value that was passed to it.

Context

All

Remarks

This function exists for diagnostic purposes only. Note that the return value is a deep copy of the passed parameter. This means that if the parameter value is an object or array, all properties or array elements are copied recursively to the return value. This corresponds to the behavior of the built-in M-Script operator `copyof`.

Example

```
var echo = maconomy::echo("Hello Maconomy!");
               print("Echo from Maconomy: ^1\n", echo);
```

fileDelete

Prototype

```
procedure maconomy::fileDelete(fileHandle)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Deletes a server file.

Parameters

Name	Type	Description
`fileHandle`	string	Identifies the server file. This handle is obtained by calling `fileSendData` or from the files property of the `dialogStatus` object.

Context

Standalone, server command-line, custom action.

Remarks

Only files that are sent to the server by calling `fileSendData` or produced by a dialog operation can be deleted using this procedure.

When a dialog operation produces a file, the file handle can be found as an array element in the `files` property of the `dialogStatus` object.

All existing server files are deleted when the login session is ended. This happens when `maconomy::logout` is called or when the M-Script session is deleted by calling `deletesession`. However, server files should be deleted by calling `fileDelete` when no longer needed to avoid taking up unnecessary space on the server.

Example

```
// Retrieve the data from the file generated by the action. var
                fileHandle = dialogData.status.files[0];

                var fileData = maconomy::fileGetData(fileHandle);

                print("fileData is:\n^1\n", fileData);

                // Delete the file on the
                server.
                maconomy::fileDelete(file
                Handle);
```

fileEnqueue

Prototype

```
procedure maconomy::fileEnqueue(fileHandle)

procedure maconomy::fileEnqueue(fileHandle, fileName)

procedure maconomy::fileEnqueue(fileHandle, fileName, MIMEType)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Queues a server file to be used for the next dialog operation.

Parameters

Name	Type	Description
`fileHandle`	string	Identifies the server file. This handle is obtained by calling `fileSendData` or from the `files` property of the `dialogStatus` object.
`fileName`	string	Optional name of the queued file.
`MIMEType`	string	Optional MIME type of the file, such as “application/pdf.”

Context

Standalone, server command-line, custom action.

Remarks

Before you use an input file in a dialog operation, the file must be queued using this procedure. Before you queue a file it must be sent to the server by calling `fileSendData`.

Alternatively, the output file of a previous dialog operation can be queued. When a dialog operation produces a file, the file handle can be found as an array element in the `files` property of the `dialogStatus` object.

More than one file can be queued, and the same file can be queued multiple times.

An error results if a dialog operation does not use all queued server files.

After a dialog operation has used an enqueued file it still exists and can be enqueued again for another dialog operation.

All existing server files are deleted when the login session is ended. This happens when `maconomy::logout` is called or when the M-Script session is deleted by calling `deletesession`. However, server files should be deleted by calling `fileDelete` when no longer needed to avoid taking up unnecessary space on the server.

Example

```
// Create file on server and enqueue it for the next
                // dialog
                operation. var
                fileData =
                ...;

                fileHandle = maconomy::fileSendData(fileData);

                maconomy::fileEnqueue(fileHandle);

                // Perform the action "Import Text" and commit
                changes. dialogData =

                    maconomy::dialogAction(dialogId, "Import_Text");

                maconomy::commit();

                // Delete the server file.
                maconomy::fileDelete(fileHa
                ndle);
```

fileGet

Prototype

```
function maconomy::fileGet(fileHandle, ostream)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Retrieves file data from a server file.

Parameters

Name	Type	Description
`fileHandle`	string	Identifies the server file. This handle is obtained by calling `fileSendData` or from the `files` property of the `dialogStatus` object.

Name	Type	Description
ostream	stream	An open output stream to which the file data will be written.

Context

Standalone, server command-line, custom action

Remarks

It is only possible to get the contents of a file sent to the server by calling `fileSendData` or produced by a dialog operation.

When a dialog operation produces a file, the file handle can be found as an array element in the `files` property of the `dialogStatus` object.

Example

```
// Perform the action "Export Text". var
               dialogData =

                       maconomy::dialogAction(dialogId, "Export_Text");

               // Create an file stream for receiving
               the data. var fname = file::tmpname();

               var f = file::open(fname, "w");

               // Retrieve the data from the file generated by the
               action. var fileHandle =
               dialogData.status.files[0];
               maconomy::fileGet(fileHandle, f);

               file::close(f);

               // Delete the file on the
               server.
               maconomy::fileDelete(file
               Handle);
```

fileGetData

Prototype

```
function maconomy::fileGetData(fileHandle)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Retrieves file data from a server file.

Parameters

Name	Type	Description
`fileHandle`	string	Identifies the server file. This handle is obtained by calling `fileSendData` or from the `files` property of the `dialogStatus` object.

Return Value

The function returns the contents of the file as a string.

Context

Standalone, server command-line, custom action.

Remarks

It is only possible to get the contents of a file sent to the server by calling `fileSendData` or produced by a dialog operation.

When a dialog operation produces a file, the file handle can be found as an array element in the `files` property of the `dialogStatus` object.

Example

```
// Perform the action "Export Text". var
               dialogData =

                       maconomy::dialogAction(dialogId, "Export_Text");

               // Retrieve the data from the file generated by the
               action. var fileHandle =
               dialogData.status.files[0];

               var fileData = maconomy::fileGetData(fileHandle);

               print("fileData is:\n^1\n", fileData);

               // Delete the file on the
               server.
               maconomy::fileDelete(fileHa
               ndle);
```

fileGetHandles

Prototype

```
function maconomy::fileGetHandles()
```

API Part

Dialog Interface

License

None

Description

Returns an array that contains handles to all server files.

Parameters

None

Return Value

The return value is an array of strings. Each element is a handle to a server file.

Context

Standalone, server command-line, custom action

Remarks

This function can be used to delete all existing server files.

All existing server files are deleted when the login session is ended. This happens when `maconomy::logout` is called or when the M-Script session is deleted by calling `deletesession`. However, server files should be deleted when no longer needed to avoid taking up unnecessary space on the server.

Example

```
// Delete all server files.
               var fileHandles = maconomy::fileGetHandles();
               for (var i in fileHandles)
                 maconomy::fileDelete(fileHandles[i]);
```

fileGetName

Prototype

```
function maconomy::fileGetName(handle)
```

API Part

Dialog Interface

License

None

Description

Returns the server-side path and file name for the file that is identified by `handle`.

Parameters

None

Return Value

The return value is a string with the name and path of the file on the server.

Context

Standalone, server command-line, custom action

Remarks

Information about generated files are stored in the M-Script session. For this reason it is not possible to obtain information about files that have not been created within the current session.

Throws

If the specified file handle does not exist in the session, a `messageError` exception is thrown.

fileSend

Prototype

```
function maconomy::fileSend(istream)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Creates a file on the server.

Parameters

Name	Type	Description
`istream`	pointer	An input stream that contains the data of the file to be created.

Return Value

The function returns a file handle of type `string`.

Context

Standalone, server command-line, custom action

Remarks

Before using an input file in a dialog operation, the file must be queued by calling `fileEnqueue`. To queue a file it must exist on the server. This function can be used to create a file on the server. An input stream with the contents of the new file is passed as a parameter to the function call.

The return value is a file handle, which can be used in subsequent calls to other file handling functions.

All existing server files are deleted when the login session is ended. This happens when `maconomy::logout` is called or when the M-Script session is deleted by calling `deletesession`. However, server files should be deleted by calling `fileDelete` when no longer needed to avoid taking up unnecessary space on the server.

Example

```
// Create file on server and enqueue it for the next
```

```
// dialog
operation.
var
fileStream =
...;

fileHandle = maconomy::fileSend(fileStream);

maconomy::fileEnqueue(fileHandle);

// Perform the action "Import Text" and commit
changes. dialogData =

    maconomy::dialogAction(dialogId, "Import_Text");

maconomy::commit();

// Delete the server
file.
maconomy::fileDelete(file
Handle);
```

fileSendData

Prototype

```
function maconomy::fileSendData(fileData)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Creates a file on the server.

Parameters

Name	Type	Description
`fileData`	string	The contents of the file to create.

Return Value

The function returns a file handle of type `string`.

Context

Standalone, server command-line, custom action

Remarks

Before you use an input file in a dialog operation, the file must be queued by calling `fileEnqueue`. To queue a file it must exist on the server. This function can be used to create a file on the server. The contents of the new file is passed as a string parameter to the function call.

The return value is a file handle that can be used in subsequent calls to other file-handling functions.

All existing server files are deleted when the login session is ended. This happens when `maconomy::logout` is called or when the M-Script session is deleted by calling `deletesession`. However, server files should be deleted by calling `fileDelete` when no longer needed to avoid taking up unnecessary space on the server.

Example

```
// Create file on server and enqueue it for the next
                // dialog
                operation. var
                fileData =
                ...;

                fileHandle = maconomy::fileSendData(fileData);

                maconomy::fileEnqueue(fileHandle);

                // Perform the action "Import Text" and commit
                changes. dialogData =

                    maconomy::dialogAction(dialogId, "Import_Text");

                maconomy::commit();

                // Delete the server file.
                maconomy::fileDelete(fileHa
                ndle);
```

getCommonValue

Prototype

```
function maconomy::getCommonValue(name)
```

API Part

Value Storage Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Retrieves a named common value from the Maconomy database.

Parameters

Name	Type	Description
`name`	string	The name of the common value to retrieve.

Return Value

The return value is an object of type `valueGetReturn` that holds the retrieved value.

Context

All

Remarks

This function retrieves a common system-wide value from the Maconomy database.

Call the function `getUserValue` to retrieve a user specific value.

Example

```
var valueName = "customerPortalSettings";
                var valueRet = maconomy::getCommonValue(valueName);
```

getConnectionInfo

Prototype

```
function maconomy::getConnectionInfo(connection)
```

License

None

Description

Returns information about the specified server connection.

Parameters

Name	Type	Description
`connection`	server handle	A handle to the server connection, as returned by `maconomy::getServerHandle`.

Return Value

The return value is an object of type `connectionInfo`.

Context

All

Remarks

None

Example

```
var sslOn =
                    maconomy::getConnectionInfo(maconomy::getServerHandle()).ssl;
```

getPopupLiteralTitle

See `popupTitle`.

getPopupLieralTitles

See `popupTitles`.

getServerHandle

Prototype

```
function maconomy::getServerHandle([serverLabel])
```

Description

Returns a handle to the server with the specified label. The available servers are specified with the `connection` keyword in the M-Script configuration file.

If no label is specified, a handle to the default server is returned.

Because the optional server handle can be applied to all `maconomy` functions, it has been omitted from the function descriptions throughout this document.

Return Value

The return value is a server handle that can be used in subsequent operations to specify the server.

Context

All

Throws

If the server label is unknown, a standard M-Script exception of type `error::runtime` is thrown.

Example

```
var serverHandle1 = maconomy::getServerHandle();
var serverHandle2 = maconomy::getServerHandle("server-label");
```

getServerInfo

Prototype

```
function maconomy::getServerInfo()
```

Description

Retrieves miscellaneous information about the server. It is not required to be logged in to call this function.

Return Value

The return value is an object of type `serverInfo` that holds the server information.

Context

All

Example

```
var serverInfo = maconomy::getServerInfo();
```

getSSOProperties

Prototype

```
function maconomy::getSSOProperties()
```

Description

This function reads and returns the status of the Single Sign On functionality. SSO is implemented in Maconomy using the Kerberos protocol. For more information, see “Single Sign On with Kerberos” in the System Administrator’s Guide.

Return Value

The return value is an object of type `ssoProperties` that holds the SSO status information.

Throws

If SSO is not active for web clients, or if SSO has not been configured on the server, an M-Script exception of type `error::stdlib::maconomy` is thrown.

Context

All

Example

```
var ssoInfo = maconomy::getSSOProperties();
```

getSystemParameter

Prototype

```
function maconomy::getSystemParameter(relationName, fieldName)
```

API Part

Preprocessing Interface

Description

Tests whether a given system parameter is set or not on the server. This means that you can make portions of your code depend on whether a given parameter is set at run time.

Parameters

Name	Type	Description
`relationName`	string	The name of a relation in Maconomy.
`fieldName`	string	The internal name of a field in Maconomy. The values of `relationName` and `fieldname` together point to a single Boolean field in the Maconomy database. Use the “Database Description” document or MDoc to see all relations and fields in Maconomy.

Return Value

An object of the type `systemParameterReturn`.

Throws

Exception of type `messageError`.

Context

All

Example

```
var b = maconomy::getSystemParameter("SystemInformation",
                  "DifferentialVAT");
```

getUserInfo

Prototype

```
function maconomy::getUserInfo()
```

Description

Retrieves miscellaneous information about the user who is currently logged in.

Return Value

The return value is an object of type `userInfo` that holds the user information.

Context

All

Example

```
var userInfo = maconomy::getUserInfo();
```

getUserValue

Prototype

```
function maconomy::getUserValue(name)
```

API Part

Value Storage Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Retrieves a named user value from the Maconomy database.

Parameters

Name	Type	Description
`name`	string	The name of the user value to retrieve.

Return Value

The return value is an object of type `valueGetReturn` that holds the retrieved value.

Context

All

Remarks

This function retrieves a user specific value from the Maconomy database. Call the function `getCommonValue` to retrieve a common system-wide value.

Example

```
var valueName = "customerPortalUserPrefs";
                var valueRet = maconomy::getUserValue(valueName);
```

hasAddOn

Prototype

```
function maconomy::hasAddOn(int)
```

API Part

Preprocessing Interface

Description

This function is used to test whether an add-on number is enabled on the server. This means that you can make portions of your code depend on whether a given add-on is installed at run time.

Parameters

Name	Type	Description
`int`	integer	The name of the user value to retrieve.

Return Value

`Boolean` – `true` if the add-on is installed, `false` otherwise.

Context

All

Example

```
if (maconomy::hasAddOn(42))
                    .
                    .
                    .
```

hllLock

Prototype

```
function maconomy::hllLock(relationName, key, comment)
```

API Part

High-Level Lock Interface

License

M-Script Dialog API (add-on 63)

Description

This function tries to obtain a lock for the record specified by `key` and `relationName`.

See also “High-Level Lock Interface.”

Parameters

Name	Type	Description
`relationName`	string	The name of a custom (MOL) relation.
`key`	object	An object that contains (at least) properties that have the same names as the key fields of the relation. If `key` is a case-insensitive object (for example `@{ key1:42}`) the matching of key fields from the custom relation and properties in the `key` object is done case-insensitively; otherwise, the matching is case-sensitive.
`comment`	string	The comment entered will be displayed when running the `mmem` utility.

Return Value

The return value is an object of type `lockId`.

Context

All

Throws

If `relationName` is not the name of a custom relation, the function throws an exception of type `invalidRelation`. If `key` does not have values for all key fields in the custom relation, `invalidKey` is thrown. If the record is already locked, an exception of type `locked` is thrown.

Example

See “High-Level Lock Interface” for an example.

hllRenewLock

Prototype

```
function maconomy::hllRenewLock(lockId)
```

API Part

High-Level Lock Interface

License

M-Script Dialog API (add-on 63)

Description

This function tries to renew the lock specified by `lockId`. If the lock still exists, the timeout is reset, and `false` is returned. If the lock no longer exists (it has timed out), the function tries to obtain a new lock for the same record. If this new lock can be obtained, `true` is returned. If the lock could not be reobtained, an exception is thrown.

Parameters

Name	Type	Description
`lockId`	object	The return value from `hllLock`.

Return Value

A Boolean value that specifies whether the lock has been lost since `hllLock` (or `hllRenewLock`) was called. If `true` is returned, it means that another user could potentially have changed the record.

Context

All

Throws

If the `lockId` argument is an invalid `lockId` object, the function throws an exception of type `invalidLockId`. This does not occur if the return value from `hllLock` is passed as an argument to `hllRenewLock`.

If the lock was lost and could not be obtained because it is locked by another user, the function throws an exception of type `lockLost`.

Example

See “High-Level Lock Interface” for an example.

hllUnlock

Prototype

```
procedure maconomy::hllUnlock(lockId)
```

API Part

High-Level Lock Interface

License

M-Script Dialog API (add-on 63)

Description

This procedure releases a lock on a record in a custom relation. If the lock does not exist, nothing is done; that is, it is not considered an error to release the same lock twice.

Parameters

Name	Type	Description
`lockId`	object	The return value from `hllLock`.

Context

All

Throws

If the `lockId` argument is an invalid `lockId` object, the procedure throws an exception of type `invalidLockId`. This does not occur if the return value from `hllLock` is passed as argument to `hllUnlock`.

Example

See “High-Level Lock Interface.”

impersonateUser

Prototype

```
procedure maconomy::impersonateUser(userName)
```

API Part

Login Session Interface

License

No license required

Description

This function enables users with administrator privileges to assume the identities of other users. See also `revertToSelf`.

Parameters

Name	Type	Description
`userName`	string	The name of the user to impersonate.

Context

Standalone, server command-line

Example

```
maconomy::login("Administrator", ...);
```

```
maconomy::impersonateUser("OtherUser"); // Now we are
"OtherUser"

...

maconomy::revertToSelf(); // Now we are "Administrator"
again maconomy::logout();
```

isLoggedIn

Prototype

```
function maconomy::isLoggedIn()
```

API Part

Login Session Interface

License

No license required

Description

Function used to detect Maconomy login status.

Return Value

A `boolean` that indicates whether a user is logged in or not.

Context

All

Example

```
if (!maconomy::isLoggedIn())
                  maconomy::login("John", "123456");
```

login

Prototype

```
function maconomy::login(userName, password)
function maconomy::login(userName)
```

API Part

Login Session Interface

License

No license required

Description

Creates a login session in Maconomy. If a password is not given, the calling script must be stamped, and there must be a valid login privilege for the specified user in the stamp. See the MStamper Reference for further information.

Parameters

Name	Type	Description
`userName`	string	The name of the user to log in.
`password`	string	The password of the user.

Return Value

The return value is an object of type `loginReturn`.

Context

Standalone, server command-line

Throws

In case of failure, a `loginError` exception is thrown.

Remarks

It is necessary to have an M-Script session before calling this function. An M-Script session is created by calling the built-in M-Script procedure `newsession`.

The password is encrypted before being sent over the network.

The login is in effect for as long as a session lasts or until `maconomy::logout()` is called or until the user is timed out on the server. The default timeout delay is five minutes, but it is possible to change this in the Maconomy server configuration file. A user cannot time out for as long as a single M-Script is running, but if more than five minutes passes between two executions of an M-Script, which uses a logged-in session, the user is automatically logged out by the server.

If the returned error code indicates that the user's password is expired, access to the SQL and Analyzer functions is blocked.

Example

```
#version 1
                newsession();
                var loginResult = maconomy::login("Administrator", "123456");
                if (loginResult.ok)
                {
                  var echo = maconomy::echo("Hello Maconomy!");
                  print("Echo from Maconomy: ^1\n", echo);
                  maconomy::logout();
                }
                deletesession();
```

logout

Prototype

```
procedure maconomy::logout()
```

API Part

Login Session Interface

License

No license required

Description

Ends a Maconomy login session.

Remarks

Deleting the current M-Script session does not log out the user, but leaves a “ghost user”on the Maconomy server. Consequently, it is strongly recommended to log out the current user by calling this function.

Ghost users time out after five minutes by default. This timeout value can be changed. See the M-Script Language Reference.

Context

Standalone, server command-line

Example

```
#version 1
                newsession();
                var loginResult = maconomy::login("Administrator", "123456");
                if (loginResult.ok)
                {
                  var echo = maconomy::echo("Hello Maconomy!");
                  print("Echo from Maconomy: ^1\n", echo);
                  maconomy::logout();
                }
                deletesession();
```

mql

Prototype

```
function maconomy::mql(mqlQuery)
function maconomy::mql(mqlQuery, mqlBind)
function maconomy::mql(mqlQuery, mqlBind, mqlParms)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used in the query

Description

This function defines and executes an MQL query command. For more information on MQL, see the MQL Language Reference. This function returns the query result as an object for further manipulation. All rows are returned, so use the function with care. See `Maconomy::mqlGetPage*` set of functions for page-oriented data retrieval.

Parameters

Name	Type	Description
`mqlQuery`	string	Any MQL query command. An MQL version tag can be supplied as part of the command. If no version tag is supplied, it is derived from the M-Script version.
`mqlBind`	object | null	Bind data for the query, for example, values for query parameters. This property is an object of type `mqlBind`.
`mqlParms`	object	Bind data for the query, for example, values for query parameters. This property is an object of type `mqlBind`.

Return Value

The return value is an object of type `mqlResult`.

Context

All

Throws

The exception `error::stdlib::maconomy::mql::parseError` is thrown if the query contains an error. Information about an error is available in the exception. The exception `error::stdlib::maconomy::mql::dbError` is thrown if a database error occurs while executing the query.

Remarks

Aggregate information can be requested in queries, but it is not available in the MQL Interface.

Example

```
try  {
                  var mqlQuery = 'MQLQUERY';
                <MQL 1.3>
                MSELECT Name1 FROM Employee WHERE EmployeeNumber =
                parmEmployeeNumber
                USING PARAMETERS parmEmployeeNumber : String
                MQLQUERY
                  var mqlBind = { parameters:{ parmEmployeeNumber : { value:
                  "1000"}
                                  }
                               }
                               ;
```

```
    var mqlParms = { mqlGetDef: true };
    var mqlResult  = maconomy::mql(mqlQuery, mqlBind, mqlParms);
    ...
}
catch(e) {
    ...
}
```

mqlBind

Prototype

```
procedure maconomy::mqlBind(mqlHandler, mqlBind)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used in the query

Description

This procedure redefines the values for query parameters, page size, and format specification for the query defined by the query handler given by the parameter `mqlHandler`.

Parameters

Name	Type	Description
`mqlHandler`	string	A query handler that could be obtained from a call to the function `mqlOpen`.
`mqlBind`	object | null	Bind data for the query, e.g. values for query parameters. This property is an object of type `mqlBind`.

Context

All

Throws

The exception `error::stdlib::maconomy::mql::invalidHandler` is thrown if the query handler could not be found. Information about an error is available in the exception.

Example

```
try {
    // Defines a query handler
    var mqlHandler  = maconomy::mqlOpen(...);
    ...
```

```
  // Redefines values for query
  parameter var mqlBind    = {
  parameters:{
                      parmEmployeeNumber : { value : "1000"}
                    }
                  }
                  ;
  maconomy::mqlBind(mqlHandler, mqlBind);
  ...
}
catch(e) {
  ...
}
```

mqlClose

Prototype

```
procedure maconomy::mqlClose(mqlHandler)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used in the query

Description

This procedure releases resources used by a query handler. Always remember to call `mqlClose` when a query handler is no longer needed.

Parameters

Name	Type	Description
`mqlHandler`	string	A query handler that could be obtained from a call to the function `mqlOpen`.

Context

All

Throws

The exception `error::stdlib::maconomy::mql::invalidHandler` is thrown if the query handler could not be found. Information about an error is available in the exception.

Example

```
try {
```

```
    // Defines a query handler
    var mqlHandler  = maconomy::mqlOpen(...);
    ...
    maconomy::mqlClose(mqlHan
    dler);
}
catch(e) {
    ...
}
```

mqlGet

Prototype

```
function maconomy::mqlGet(mqlHandler)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used in the query

Description

This function executes the query associated with a query handler. It returns the query result as an object for further manipulation. All rows are returned, so use this function with care. See the `Maconomy::mqlGetPage*` set of functions for page-oriented data retrieval.

Parameters

Name	Type	Description
`mqlHandler`	string	A query handler that could be obtained from a call to the function `mqlOpen`.

Return Value

The return value is an object of type `mqlResult`.

Context

All

Throws

The exception `error::stdlib::maconomy::mql::invalidHandler` is thrown if the query handler could not be constructed. Information about an error is available in the exception. The exception `error::stdlib::maconomy::mql::dbError` is thrown if a database error occurs while executing the query.

Remarks

Aggregate information can be requested in queries, but it is not available in the MQL Interface.

Example

```
{
                  var mqlHandler = maconomy::mqlOpen(...);
                  var mqlResult  = maconomy::mqlGet(mqlHandler);
                  ...
                }
                catch(e) {
                  ...
                }
```

mqlGetDef

Prototype

```
function maconomy::mqlGetDef(mqlHandler)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used in the query

Description

This function obtains information about type and titles for fields selected in the query defined by a query handler.

Parameter

Name	Type	Description
`mqlHandler`	string	A query handler that could be obtained from a call to the function `mqlOpen`.

Return Value

The return value is an object of type `mqlDef`.

Context

All

Throws

The exception `error::stdlib::maconomy::mql::invalidHandler` is thrown if the query handler could not be constructed. Information about an error is available in the exception.

Example

```
try  {
                  var mqlQuery = 'MQLQUERY';
                <MQL 1.3>
                MSELECT Name1 FROM Employee
                MQLQUERY
                  var mqlHandler  = maconomy::mqlOpen(mqlQuery, mqlBind);
                  var mqlDef      = maconomy::mqlGetDef(mqlHandler);
                  ...
                }
                catch(e) {
                  ...
                }
```

mqlGetPage

Prototype

```
function maconomy::mqlGetPage(mqlHandler, pageNumber)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used in the query

Description

This function executes the query associated with a query handler. It returns the query result as an object for further manipulation. The function returns one specific page of rows.

Parameters

Name	Type	Description
`mqlHandler`	string	A query handler that could be obtained from a call to the function `mqlOpen`.

Return Value

The return value is an object of type `mqlResult`.

Context

All

Throws

The exception `error::stdlib::maconomy::mql::invalidHandler` is thrown if the query handler could not be constructed. Information about an error is available in the exception. The exception `error::stdlib::maconomy::mql::dbError` is thrown if a database error occurs while executing the query. If the requested page is not available, the exception `error::stdlib::maconomy::mql::invalidPage` is thrown.

Remarks

The number of rows in a page is specified when defining the query handler. Aggregate information can be requested in queries, but it is not available in the MQL Interface.

A call to this function sets the current page to the requested page. This means that a subsequent call to `maconomy::mqlGetPageNext` or `maconomy::mqlGetPagePrev` returns the next/previous page of rows relative to the requested page.

Example

```
{
                  var mqlHandler = maconomy::mqlOpen(...);
                  var mqlResult  = maconomy::mqlGetPage(mqlHandler, 1);
                  ...
                }
                catch(e) {
                  ...
                }
```

mqlGetPageNext

Prototype

```
function maconomy::mqlGetPageNext(mqlHandler)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used in the query

Description

This function executes the query associated with a query handler. It returns the query result as an object for further manipulation. The function returns the next page of rows.

Parameters

Name	Type	Description
`mqlHandler`	string	A query handler that could be obtained from a call to the function `mqlOpen`.

Return Value

The return value is an object of type `mqlResult`.

Context

All

Throws

The exception `error::stdlib::maconomy::mql::invalidHandler` is thrown if the query handler could not be constructed. Information about an error is available in the exception. The exception `error::stdlib::maconomy::mql::dbError` is thrown if a database error occurs while executing the query. If the next page is not available, the exception `error::stdlib::maconomy::mql::invalidPage` is thrown.

Remarks

The number of rows in a page is specified when defining the query handler. Aggregate information can be requested in queries, but it is not available in the MQL Interface.

A call to this function sets the current page to the returned page. This means that a subsequent call to `maconomy::mqlGetPageNext` or `maconomy::mqlGetPagePrev` returns the next/previous page of rows relative to the returned page.

Example

```
try {
    var mqlBind = {
    pageSize:2 };
    var mqlHandler = maconomy::mqlOpen(mqlQuery,
    mqlBind); var mqlResult =
    maconomy::mqlGetPageNext(mqlHandler);
    while(mqlResult.more) {
      ...
      mqlResult = maconomy::mqlGetPageNext(mqlHandler);
    }
    maconomy::mqlClose(mqlHandler);
  }
  catch(e) {
    ...
  }
```

mqlGetPagePrev

Prototype

```
function maconomy::mqlGetPagePrev(mqlHandler)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used in the query

Description

This function executes the query associated with a query handler. It returns the query result as an object for further manipulation. This function returns the previous page of rows.

Parameters

Name	Type	Description
`mqlHandler`	string	A query handler that could be obtained from a call to the function `mqlOpen`.

Return Value

The return value is an object of type `mqlResult`.

Context

All

Throws

The exception `error::stdlib::maconomy::mql::invalidHandler` is thrown if the query handler could not be constructed. Information about an error is available in the exception. The exception `error::stdlib::maconomy::mql::dbError` is thrown if a database error occurs while executing the query. If the next page is not available, the exception `error::stdlib::maconomy::mql::invalidPage` is thrown.

Remarks

The number of rows in a page is specified when defining the query handler. Aggregate information can be requested in queries, but it is not available in the MQL Interface.

A call to this function sets the current page to the returned page. This means that a subsequent call to `maconomy::mqlGetPageNext` or `maconomy::mqlGetPagePrev` returns the next/previous page of rows relative to the returned page.

Example

```
try  {
                var mqlBind    = { pageSize:2 };
                var mqlHandler = maconomy::mqlOpen(mqlQuery, mqlBind);
                var mqlResult  = maconomy::mqlGetPage(mqlHandler, 10);
                ...
                // Getting page 9
                mqlResult = maconomy::mqlGetPagePrev(mqlHandler);
                ...
                // Getting page 8
                mqlResult = maconomy::mqlGetPagePrev(mqlHandler);
```

```
    ...
    maconomy::mqlClose(mqlHan
    dler);
}
catch(e) {
    ...
}
```

mqlOpen

Prototype

```
function maconomy::mqlOpen(mqlQuery)
function maconomy::mqlOpen(mqlQuery, mqlBind)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used in the query

Description

This function defines a query handler from an MQL query command. For further information on MQL, see the MQL Language Reference.

Parameters

Name	Type	Description
`mqlQuery`	string	Any MQL query command. An MQL version tag can be supplied as part of the command. If no version tag is supplied, it is derived from the M-Script version.
`mqlBind`	object | null	Bind data for the query, e.g. values for query parameters. This property is an object of type `mqlBind`.

Return Value

The return value is a query handler to be used with the other `maconomy::mql` functions.

Context

All

Throws

The exception `error::stdlib::maconomy::mql::parseError` is thrown if the query contains an error. The exception `error::stdlib::maconomy::mql::invalidHandler` is thrown if the query handler could not be constructed. Information about an error is available in the exception.

Remarks

The call does not execute the query, but the query is validated for syntax, type and semantic errors.

Always remember to call the `maconomy::mqlClose` at some point after a successful call to `maconomy::mqlOpen`.

Example

```
// Defines a query handler for a query containing a query parameter try {
                  var mqlQuery = 'MQLQUERY';
                <MQL 1.3>
                MSELECT Name1 FROM Employee WHERE EmployeeNumber =
                parmEmployeeNumber
                USING PARAMETERS parmEmployeeNumber : String
                MQLQUERY
                  var mqlBind    = { parameters:{
                                       parmEmployeeNumber : { value : "1000"}
                                     }
                                   }
                                   ;
                  var mqlHandler  = maconomy::mqlOpen(mqlQuery, mqlBind);
                  ...
                }
                catch(e) {
                  ...
                }
```

mqlUniverseQuery

Prototype

```
function maconomy::mqlUniverseQuery(universeName, searchBind[,
                parms])
```

API Part

MQL Interface

License

No license required

Description

This function selects a universe based on the `universeName` string parameter and the optional string property `interfaceName` in `parms`. A query is created from the field selections and restrictions given by the parameter `searchBind`, which is an object of type `searchBind`. Finally, the values for query parameters, page size, and format specification are redefined for the query if `searchBind` contains an `mqlBind` object. In addition, the universe definition and the MQL definition can be fetched.

Parameters

Name	Type	Description
`universeName`	string	A universe identifier.
`searchBind`	object	Selection of fields and definition of restrictions. This property is an object of type `searchBind`.
`parms`	object	An object with additional settings. The properties of the object are described in the following. The type of `parms` is an object with any of the following optional properties: ▪ `interfaceName` – string | null – A interface identifier in the selected universe. If no interface identifier is specified, then the default interface is used. ▪ `universeGetDef` – bool – A flag that tells the server whether to include the universe definition in the response object. ▪ `mqlGetDef` – bool – A flag that tells the server whether to include the MQL definition in the response object.

Return Value

None

Context

All

Example

```
var universeName = "MyUniverse";

                var
            searchBind
                = {
                   columnIndexOutput : [
                     "Fie
                     ld
                     1",
                     "Fie
                     ld
                     2"
                   ],
                   restrictions : [
                     { columnName: "Field 1", restriction:["2..14"] }
                   ]
                };
            var parms = {
```

```
  interfaceName:
  "MyUniverseInterface",
  universeGetDef: true,
  mqlGetDef: true
};
try  {
  var result = maconomy::mqlUniverseQuery(universeName,
    searchBind, parms);
  ...
}
catch(e) {
  ...
}
```

popup

Prototype

```
function maconomy::popup(typeName, ordValue)
```

API Part

Popup Interface

License

No license required

Description

Creates and returns a pop-up value.

Parameters

Name	Type	Description
`typeName`	string	Name of the pop-up type to create a value of.
`ordValue`	int	The ordinal value of the popup value to create. The ordinal value must be greater than or equal to -1.

Return Value

This function returns the created pop-up value.

Context

All

Remarks

The function does not check if `typeName` is a valid type name. In addition, it is not checked if `ordValue` is out of range—that is, too large. This makes it possible to create an invalid pop-up

value that can lead to an error when this value is sent to the server in a subsequent subroutine call.

If the ordinal value is -1 or `null`, a null literal is created. This is equivalent to calling `popupNull`.

Example

```
var typeName = typeof(existingValue);
               var ordVal = ...;
               var newVal = maconomy::popup(typeName, ordVal);
```

popupNull

Prototype

```
function maconomy::popupNull (typeName)
```

API Part

Popup Interface

License

No license required

Description

Creates and returns a pop-up null value.

Parameters

Name	Type	Description
`typeName`	string	Name of the pop-up type to create a null value of.

Return Value

The function returns the created pop-up null value.

Context

All

Remarks

A pop-up null value corresponds to a blank pop-up entry in the Maconomy client.

This function does not check whether `typeName` is a valid type name. This makes it possible to create an invalid pop-up null value that can lead to an error when this value is sent to the server in a subsequent subroutine call.

Example

```
var typeName = typeof(existingValue);
               var newVal = maconomy::popupNull(typeName);
```

popupTitle

Prototype

```
function maconomy::popupTitle(popupValue)
function maconomy::popupTitle(typeName, ordValue)
function maconomy::popupTitle(dialogId, popupValue)
function maconomy::popupTitle(dialogId, typeName, ordValue)
```

API Part

Popup Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Returns the display title of the requested pop-up value.

The optional dialog identifier must be used for pop-ups of the type “LayoutNameType,” because the literals of this pop-up type depend on the currently opened dialog. The dialog identifier may always be present, so users who want to perform truly generic pop-up handling should always pass a valid dialog identifier.

Parameters

Name	Type	Description
`popupValue`	popup	The pop-up literal to return the display title of.
`typeName`	string	The name of the pop-up type to which the requested pop-up value belongs.
`ordValue`	int	The ordinal value of the pop-up value to get the display title of. The ordinal value must be greater than or equal to zero.
`dialogId`	string	A dialog ID referring to an open dialog. This ID must be passed to the function when using “LayoutNameType” pop-ups.

Return Value

The function returns the display title of the requested pop-up value. The type of the return value is `string`.

Context

All

Remarks

The prefix `maconomy::` is not necessary for this function. It is in addition a shorthand for the `maconomy::getPopupLiteralTitle()` function.

Example

```
var title = maconomy::popupTitle(existingValue);
```

popupTitles

Prototype

```
function maconomy::popupTitles(popupValue)
function maconomy::popupTitles(typeName)
```

API Part

Popup Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Returns an array that contains the display titles of all values in a pop-up type.

The optional dialog identifier must be used for pop-ups of the type “LayoutNameType,” because the literals of this pop-up type depend on the currently opened dialog. The dialog identifier may always be present, so users who want to perform truly generic pop-up handling should always pass a valid dialog identifier.

Parameters

Name	Type	Description
`popupValue`	popup	A pop-up literal in the type to get all display titles from.
`typeName`	string	The name of the pop-up type to get all display titles from.

Return Value

The return value is an array with elements of type `string`. The array contains the display titles of all values in the given pop-up type.

Context

All

Remarks

The prefix `maconomy::` is not necessary for this function. It is in addition a shorthand for the `maconomy::getPopupLiteralTitles()` function.

Example

```
var titles = maconomy::popupTitles(existingValue);
```

popupValidate

Prototype

```
function maconomy::popupValidate(popupTypeName)
```

API Part

Popup interface

License

M-Script Dialog API (add-on 63)

Description

Validates the name of a pop-up type.

Parameters

Name	Type	Description
`popupTypeName`	string	Analyze pop-up type name to be verified.

Return Value

A `validateReturn` object.

Context

All

preprocess

Prototype

```
procedure maconomy::preprocess(output, input)
```

API Part

Preprocessing Interface

Description

This function allows M-Script code sections to be dependent on add-ons as well as system parameters and system information. This means that you can, for example, show a message if a certain system parameter has been marked, or run a subroutine if a certain add-on has been installed. For more information, see “MDL and MPL Preprocessor” in the System Administrator Guide.

Parameters

Name	Type	Description
`output`	ostream	Output stream to which the preprocessed code section is written.
`input`	string|istream	String or input stream which is to be preprocessed.

Throws

If the preprocessing fails, a `messageError` exception is thrown.

Context

All

Example

```
var code = "...";
                maconomy::preprocess(io::stdout(), code);
```

printDelete

Prototype

```
procedure maconomy::printDelete(printHandle)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Deletes a server print.

Parameter

Name	Type	Description
`printHandle`	string	Identifies the server print. This handle is obtained from the `prints` property of the `dialogStatus` object.

Context

Standalone, server command-line, custom action

Remarks

Only prints that are produced by a dialog action can be deleted using this procedure.

When a dialog operation action produces a print, the print handle can be found as an array element in the `prints` property of the `dialogStatus` object.

All existing server prints are deleted when the login session is ended. This happens when `maconomy::logout` is called or when the M-Script session is deleted by calling `deletesession`. However, server prints should be deleted by calling `printDelete` when no longer needed to avoid taking up unnecessary space on the server.

Example

```
// Retrieve the print generated by the action. var
              printHandle =
              dialogData.status.prints[0];
```

```
...
// Delete the file on the
server.
maconomy::printDelete(printH
andle);
```

printGetPdf

Prototype

```
function maconomy::printGetPdf(printHandles, ostream [, args])
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Retrieves a print from the server as a PDF document and writes it to an output stream.

Parameters

Name	Type	Description
`printHandles`	string	Identifies the server print(s). The argument can be either a single print handle or an array of print handles. In the latter case all of the specified prints are concatenated into a single PDF document. If the print handles were produced by a combination of MPL 2 (or 1) and MPL 3 layouts, the concatenation fails. The handles are obtained from the `prints` property of the `dialogStatus` object.
`ostream`	stream	An open output stream. An output stream can be obtained by calling `file::open` or `io::ostring` (see the M-Script Language Reference for details).
`args`	object	Optional object containing additional settings. The `args` object can be used to control some aspects of the PDF generation. If the print(s) were produced with MPL 3, there are no properties that can be given. For MPL 2 (or 1), the properties that may be specified within the object are: ▪ `landscape` – Boolean — Set this property to `true` if the PDF document should be generated in landscape mode. The default must be specified in the MPL file. This property can only be used with the `papersize` property. ▪ `papersize` – string – Specify the paper size. Example values are "legal," "executive," "A4," and so on. The available paper sizes depend on the system setup. The default paper size must be specified in the MPL file.

Context

Standalone, server command-line, custom action

Remarks

Before you use this function, a print must have been queued on the server, for example, through a `dialogAction`. Given the print handle of such a print, `printGetPdf` converts it to a PDF file that is then written to the specified output stream. When concatenating several print handles to one PDF, the prints must all have been produced by the same MPL engine, meaning that MPL 2 (or 1) prints cannot be concatenated with MPL 3 prints.

Throws

`error::stdlib::maconomy` if a bad output file is specified, or if there is an unknown print handle, or if some server-side error occurs.

Example

```
// Create a print on the server.
                var data = maconomy::dialogAction(dialogId, "Print_Invoice");
                var p = data.status.prints[0];
                var args = { papersize: "legal", landscape: true };
                // Write a PDF version of the print to
                a file. var f =
                file::open("print.pdf", "wb");
                maconomy::printGetPdf(p, f, args);
                file::close(f);
                // Or print the PDF file such that the web browser
                // automatically starts the PDF reader
                //   setcontenttype("application/pdf");
                //   maconomy::printGetPdf(p, io::stdout(), args);
                // Delete the print from the
                server.
                maconomy::printDelete(p);
```

readvalue

Prototype

```
function maconomy::readvalue(input)
```

API Part

Preprocessing Interface

Description

This function tries to read an M-Script value from the specified input stream. However, M-Script preprocesses the code in `input` before passing it to the M-Script parser, using the `preprocess()` command.

Parameters

Name	Type	Description
input	istream	Stream that contains an M-Script expression.

Return Value

A `readValueReturn` object.

Throws

Exception of type `messageError`.

Context

All

Example

```
var x = maconomy::readvalue("{a:10, b:[20,30], c:\"John\"}");
```

revertToSelf

Prototype

```
procedure maconomy::revertToSelf()
```

API Part

Login Session Interface

License

No license required

Description

This procedure discontinues a user impersonation. See also `impersonateUser`.

Context

Standalone, server command-line.

Example

```
maconomy::login("Administrator", ...);
                maconomy::impersonateUser("OtherUser"); // Now we are
                "OtherUser"
                ...
                maconomy::revertToSelf(); // Now we are "Administrator"
                again maconomy::logout();
```

rglClose

Prototype

```
procedure maconomy::rglClose(rglId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Closes a previously opened RGL report.

Parameters

Name	Type	Description
`rglId`	string	Identifies the RGL dialog. This ID is obtained by calling `rglOpen`.

Context

Standalone, server command-line, custom action

Throws

If there is a failure, a `dialogError` exception is thrown.

Example

```
rglId = maconomy::rglOpen("...");
                ...
                maconomy::rglClose(r
                glId);
```

rglExecute

Prototype

```
function maconomy::rglExecute(rglId, record)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Updates field values in an RGL report dialog and executes the report.

Parameters

Name	Type	Description
`rglId`	string	Identifies the RGL dialog. This ID is obtained by calling `rglOpen`.

Name	Type	Description
`record`	object	The data for the entry. This property is an object of type `record`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

If there is a failure, a `dialogError` exception is thrown.

Example

```
// Change 'myField' to 42 and execute the report
                rglData = maconomy::rglExecute(rglId, {myField:{value:42}});
```

rglGetDef

Prototype

```
function maconomy::rglGetDef(rglId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Returns the static description of an RGL report dialog. This description includes the number and type of panes, field names, access rights, and so forth. The return value is identical to that of `dialogGetDef`, and for that reason it contains some information that is irrelevant to RGL reports.

Parameters

Name	Type	Description
`rglId`	string	Identifies the report dialog. This ID is obtained by calling `rglOpen`.

Return Value

The return value is an object of type `dialogDef`.

Context

Standalone, server command-line, custom action

Remarks

It is not necessary to call this function to access a report dialog, but it enables an M-Script application to not assume any knowledge of a specific report dialog before using it. This way, it is possible to write generic M-Script programs that can call any RGL report.

Example

```
// Get and print the definition of this report dialog. var
                rglDef = maconomy::rglGetDef(rglId);
                dumpvalue(dialogDef);

                print("\n");
```

rglInit

Prototype

```
function maconomy::rglInit(rglId)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Fetches default data for a specific report dialog.

Parameters

Name	Type	Description
`rglId`	string	Identifies the report dialog. This ID is obtained by calling `rglOpen`.

Return Value

This function returns an object of type `dialogReturn`.

Context

Standalone, server command-line, custom action

Throws

If there is a failure, a `dialogError` exception is thrown.

Example

```
// Get default report dialog data
                var rglData = maconomy::rglInit(rglId);
```

rglList

Prototype

```
function maconomy::rglList()
function maconomy::rglList(groupName)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Fetches a list of all RGL reports known to the server, or only those in a specific group.

This list is built from the `StandAloneList.txt` file on the server.

Parameters

Name	Type	Description
`group`	string	Name of the RGL report group. Must match one of the group names in the `StandAloneList.txt` file.

Return Value

This function returns an object of type `rglList`.

Context

Standalone, server command-line, custom action

Example

```
// Get and print report list var list =
                maconomy::rglList();
                dumpvalue(list);
                print("\n");
```

The output could be something like:

```
 [
  {
    title:"M-Script 1",
    name:"MScript1",
    group:"M-Script"
  },
  {
    title:"M-Script 2",
    name:"MScript2",
    group:"M-Script"
```

```
  },
  {
    title:"M-Script 3",
    name:"MScript3",
    group:"M-Script"
  }
]
```

rglOpen

Prototype

```
function maconomy::rglOpen(rglName)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Open a named RGL report.

Parameters

Name	Type	Description
`rglName`	string	The RGL report name. This refers to the internal name of a report in the `StandAloneList.txt` file.

Return Value

This function returns an object of type `rglId`.

Context

Standalone, server command-line, custom action

Example

```
// Open RGL report and check for errors
var rgl = maconomy::rglOpen("MScriptTest1");
if (!rgl.ok)
{
  print("Error: ^1\n", rgl.error);
  exit(0);
}
  // Get RGL dialog
identifier var rglId =
rgl.id;
```

rollback

Prototype

```
procedure maconomy::rollback()
```

API Part

Transaction Control Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Rolls back the changes to the database in the current transaction.

Context

Standalone, server command-line

Remarks

If automatic commit is disabled (which it is by default) it is necessary to commit all changes to the database before they take effect. If changes are regretted before committed, this function can be used to roll back the changes.

If automatic commit is enabled, this function has no effect.

To change the state of automatic commit call `maconomy::autoCommit`.

Example

```
// Open dialog 'Users' and get record for the user 'User'. var
               dialogId = maconomy::dialogOpen("Users");

               var dialogRet =

                   maconomy::dialogGet(d
                          ialogId,

                                          {NameOfUser : {value : "User"}});

               print("Getting user ok?: ^1\n", dialogRet.status.ok);

               // Delete the entry.

               dialogRet = maconomy::dialogDeleteUpper(dialogId);

               print("Deleting user ok? ^1\n", dialogRet.status.ok);

               // Regret and roll back
               operation.
               maconomy::rollback();
```

setCommonValue

Prototype

```
procedure maconomy::setCommonValue(name, value)
```

API Part

Value Storage Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Stores a named common value in the Maconomy database.

Parameters

Name	Type	Description
`name`	string	The name of the common value to store.
`value`	any	The M-Script value to store in the database.

Context

Standalone, server command-line, custom action

Remarks

This procedure stores a named common system-wide value in the Maconomy database. Call the procedure `setUserValue` to store a user specific value.

The parameter `value` can be any M-Script value including objects and arrays nested to an arbitrary level.

Example

```
var valueName = "customerPortalSettings"; var settings
              = { a : 42, b : [1,2] };
              maconomy::setCommonValue(valueName,
              settings);
```

setUserValue

Prototype

```
procedure maconomy::setUserValue(name, value)
```

API Part

Value Storage Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Stores a named user value in the Maconomy database.

Parameters

Name	Type	Description
`name`	string	The name of the user value to store.
`value`	any	The M-Script value to store in the database.

Context

Standalone, server command-line, custom action

Remarks

This procedure stores a named user-specific value in the Maconomy database. Call the procedure `setCommonValue` to store a common system-wide value.

The parameter `value` can be any M-Script value, including objects and arrays nested to an arbitrary level.

Example

```
var valueName = "customerPortalMaxRows"; var
                maxRows = 42;
                maconomy::setUserValue(valueName,
                maxRows);
```

setWaitForServer

Prototype

```
procedure maconomy::setWaitForServer(seconds)
```

Description

Overrides the global value for the WaitForServer timeout. The new timeout is used for all subsequent server operations:

- A timeout value that is greater than zero sets the timeout to the specified number of seconds.
- A timeout value of zero causes MScript to wait indefinitely for the server operations to complete; that is, the operation will never time out.
- A timeout value of null restores the original timeout specified by waitForServer in the .I file.

Context

Standalone

Example

```
maconomy::setWaitForServer(0); // disable the server timeout try {
                    maconomy::rglExecute(…); // execute a long-running report
```

```
} finally {
   // finally restore the original timeout:
   maconomy::setWaitForServer(null);
   }
```

setWarningReply

Prototype

```
function maconomy::setWarningReply(yes)
```

API Part

Dialog Interface

License

M-Script Dialog API (add-on 63)

Description

Sets the automatic reply to warnings encountered during dialog operations.

Parameters

Name	Type	Description
`yes`	bool	The answer to warnings that are encountered during dialog operations. If this parameter is `true` the Dialog Interface answers “Yes” to such warnings. Otherwise, the automatic reply is “No,” which causes the executing dialog operation to fail.

Return Value

This function returns the state before the call. If `true` is returned the automatic reply to warnings before the call was “Yes.” Otherwise, the automatic reply was “No” before the call.

Context

Standalone, server command-line

Remarks

Callbacks from the application are not supported from the M-Script Maconomy API. However, this function makes it possible to handle warnings encountered during dialog operations in a primitive way.

The default behavior for the Dialog Interface is to answer “Yes” to all warnings.

Note that answering “No” to a warning causes the executing dialog operation to fail and rolls back the current transaction.

Example

```
// Answer "No" to all warnings.
var oldReply =
maconomy::setWarningReply(false);
```

```
.
.
.
// Restore the previous state.
maconomy::setWarningReply(oldR
eply);
```

singleLogin

Prototype

```
function maconomy::singleLogin()
```

API Part

Login Session Interface

License

Add-on 79

Description

Attempts to perform a login using the user identity obtained from the underlying OS. The obtained user identity can either be a valid Maconomy account in itself, or be specified as the network user name for a Maconomy account.

Return Value

The return value is an object of type `loginReturn`.

Context

Standalone, server command-line

Throws

If the login fails, a `loginError` exception is thrown.

If `singleLogin` is not supported, a `messageError` exception is thrown.

Remarks

This function is only implemented for the Windows NT family. To pass a user identity from the client browser through the web server and on to M-Script, at least version 5.0 of Internet Explorer (IE) and version 5.0 of Internet Information Server (IIS) are required.

Example

```
var loginRes;
t
r
y
        loginRes =
    maconomy::singleLogin();
catch
error::stdlib::maconomy::login(e)
```

```
    ... // login failed on
server. catch
error::stdlib::maconomy(e)

     ... // "singleLogin" not
            supported.
```

sql

Prototype

```
function maconomy::sql(statement)
function maconomy::sql(statement, maxRecs, firstRec, ...)
```

API Part

SQL Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Executes an SQL select statement on the Maconomy server.

Parameters

Name	Type	Description
`statement`	string	Any SQL select statement to be executed on the Maconomy server. The statement can contain placeholders `^1, ^2, ...`. These placeholders are replaced by the arguments that follow `firstRec`. The placeholder `^1` is replaced by the first argument that follows `firstRec`, `^2` is replaced by the second argument that follows `firstRec`, and so forth.
`maxRecs`	int | null	The maximum number of records that the function call should return. If this parameter is omitted or `null` is passed, all records are returned.
`firstRec`	int | null	The number of the first record to return. The index of the first record is zero. If this parameter is omitted or `null` is passed. The function call returns records starting from the first available record.
`...`	simple	The placeholders `^1, ^2, ...` in `statement` are replaced by the corresponding remaining parameters. The placeholder `^1` is replaced by the first of these arguments, `^2` by the second, and so forth.

Return Value

The return value is an object of type `sqlReturn`.

Context

All

Remarks

If the call fails—for example, because of an error in the SQL statement—the result is a run-time error.

Not all database types are translated correctly to M-Script types. When you use Oracle as the underlying database, `bool` values and Maconomy pop-up values are returned as `int` values. When you use SQL Server, `amount` values are returned as `real` values as well. There is no work-around for this issue.

This function is almost identical in behavior to `sqlRestricted`. The difference is that the data returned by this function is not restricted in any way.

Example

```
// Get the name of the first 2 users in the system. var
               sqlResult = maconomy::sql('SQL', 2, null);

                 select nameofuser from userinformation order by nameofuser

               SQL
```

sqlDelete

Prototype

```
function maconomy::sqlDelete(relName, whereClause)
```

API Part

SQL Interface

License

M-Script Dialog API (add-on 63)

Description

Deletes records from the Maconomy database.

Parameters

Name	Type	Description
`relName`	string	The name of the relation to delete from.
`whereClause`	string	An SQL `where` clause that restricts the set of records to delete from the relation. If the `where` clause is the empty string, all records in the relation are deleted.

Return Value

The return value is an object of type `sqlModifyReturn`.

Context

Standalone, server command-line, custom action

Remarks

This function call generates an SQL `delete` statement that is executed on the Maconomy server. The generated SQL statement is:

```
delete from <relName> where <whereClause>;
```

Only a few relations that are used by the Maconomy Enterprise Portal can be modified using this function call, but it is possible for all customer-specific relations.

Example

```
// Delete item '1234'.
                var where  = "itemnumber = '1234'";
                var sqlReturn = maconomy::sqlDelete("Rel", where);
```

sqlInsert

Prototype

```
function maconomy::sqlInsert(relName, values)
```

API Part

SQL Interface

License

M-Script Dialog API (add-on 63)

Description

Inserts a record into the Maconomy database.

Parameters

Name	Type	Description
`relName`	string	The name of the relation to insert into.
`values`	object	Contains the field values to insert. The object properties must be named after the field names, and the property values must match the database types of the corresponding fields.

Return Value

The return value is an object of type `sqlModifyReturn`.

Context

Standalone, server command-line, custom action

Remarks

This function call generates an SQL `insert` statement that is executed on the Maconomy server. The object `values` is of the form:

```
{
  field1
  : val1,
  field2
  : val2,
  ...
  fieldn : valn
}
```

and the generated SQL statement is:

```
insert into <relName>
   (field1, field2, ..., fieldn)
   values (val1, val2, ..., valn);
```

If some fields are not assigned a value in the parameter `values`, those fields get the value `null` in the inserted record.

Only a few relations that are used by the Maconomy Enterprise Portal can be modified using this function call, but it is possible for all customer-specific relations.

Example

```
// Insert new item '1236'. var
               values = {
                             itemnumber :
                               "1236",
                             itemtext   : "Sofa, 2 persons, cotton"
                           };
              var modReturn = maconomy::sqlInsert("Rel",
                               values);
```

sqlNative

Prototype

```
function maconomy::sqlNative(statement)
function maconomy::sqlNative(statement, maxRecs, firstRec,
...)
```

API Part

SQL Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Executes a native SQL `select` statement on the database that underlies the Maconomy server. It is not possible to use any Maconomy-specific macros in the statement.

Parameters

Name	Type	Description
`statement`	string	Any SQL select statement to be executed on the Maconomy server. The statement can contain placeholders `^1, ^2, ....` These placeholders are replaced by the arguments that follow `firstRec`. The placeholder `^1` is replaced by the first argument that follows `firstRec`, `^2` is replaced by the second argument that follows `firstRec`, and so forth.
`maxRecs`	int | null	The maximum number of records that the function call should return. If this parameter is omitted or `null` is passed, all records are returned.
`firstRec`	int | null	The number of the first record to return. The index of the first record is zero. If this parameter is omitted or `null` is passed, this function call returns records starting from the first available record.
`...`	simple	The placeholder strings `^1`, `^2`, `...` in `statement` are replaced by the corresponding remaining parameters. The placeholder `^1` is replaced by the first of these arguments, `^2` by the second, and so forth.

Return Value

The return value is an object of type `sqlReturn`.

Context

All

Remarks

If the call fails—for example, because of an error in the SQL statement—the result is a run-time error.

Not all database types are translated correctly to M-Script types. When you use Oracle as the underlying database, `bool` values and Maconomy pop-up values are returned as `int` values. When you use SQL Server, `amount` values are returned as `real` values as well. There is no work-around for this issue.

This function is almost identical in behavior to `sql`. The difference is that all checks are disabled and that Maconomy macros are not available.

Example

```
// Get the name of the first 2 users in the system. var
                sqlResult = maconomy::sqlNative('SQL', 2,
                null);

                  select nameofuser from userinformation order by nameofuser
```

S
Q
L

sqlRestricted

Prototype

```
function maconomy::sqlRestricted(statement)
function maconomy::sqlRestricted(statement, maxRecs, firstRec, . . .)
```

API Part

SQL Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Executes an SQL select statement on the Maconomy server. Maconomy's access control system restricts which data is returned.

Parameters

Name	Type	Description
`statement`	string	Any SQL select statement to be executed on the Maconomy server. The statement can contain placeholders `^1, ^2, ...`. These placeholders are replaced by the arguments that follow `firstRec`. The placeholder `^1` is replaced by the first argument that follows `firstRec`, `^2` is replaced by the second argument that follows `firstRec`, and so forth.
`maxRecs`	int | null	The maximum number of records that the function call should return. If this parameter is omitted or `null` is passed, all records are returned.
`firstRec`	int | null	The number of the first record to return. The index of the first record is zero. If this parameter is omitted or `null` is passed, the function call returns records starting from the first available record.
`...`	simple	The placeholder strings `^1`, `^2`, `...` in `statement` are replaced by the corresponding remaining parameters. The placeholder `^1` is replaced by the first of these arguments, `^2` by the second, and so forth.

Return Value

The return value is an object of type `sqlReturn`.

Context

All

Remarks

If the call fails—for example, because of an error in the SQL statement—the result is a run-time error.

Not all database types are translated correctly to M-Script types. When you use Oracle as the underlying database, `bool` values and Maconomy pop-up values are returned as `int` values. When you use SQL Server, `amount` values are returned as `real` values as well. There is no work-around for this issue.

This function is almost identical in behavior to `sql`. The difference is that the data that is returned by this function is restricted by Maconomy's access control system.

This check is done via the access control views that are defined by Maconomy. These views reflect many of the important Maconomy relations with access requirements added to them. The actual check is simply done by prepending "AC" to the relation name, such as "JOBHEADER," which would become "ACJOBHEADER."

Not that to identify the relation names in the query string, an internal parsing is done of the string. The parser that is used knows most of the SQL language but is not complete. This means that some advanced SQL queries are unable to run using the `sqlRestricted` function.

Example

```
// Get the name of the first 2 users in the system.
                var sqlResult = maconomy::sqlRestricted('SQL', 2, null);
                  select nameofuser from userinformation order by nameofuser
                SQL
```

sqlUpdate

Prototype

```
function maconomy::sqlUpdate(relName, values, whereClause)
```

API Part

SQL Interface

License

M-Script Dialog API (add-on 63)

Description

Updates one or more existing records in the Maconomy database.

Parameters

Name	Type	Description
`relName`	string	The name of the relation in which to update records.
`values`	object	Contains the updated field values. The object properties must be named after the field names, and the property values must match the database types of the corresponding fields.

Name	Type	Description
`whereClause`	string	An SQL `where` clause that restricts the set of records to update in the relation. If the `where` clause is the empty string, all of the records in the relation are modified.

Return Value

The return value is an object of type `sqlModifyReturn`.

Context

Standalone, server command-line, custom action

Remarks

This function call generates an SQL `update` statement that is executed on the Maconomy server. The object `values` is of the form:

```
{
  field1 :
  val1,
  field2 :
  val2,

  ...

  fieldn : valn
}
```

and the generated SQL statement is:

```
  update
<relName>

   set field1 =
       val1,
       field2 =
       val2,

       .
       .
       .

       fieldn = valn;
```

Example

Only a few relations used by the Maconomy Enterprise Portal can be modified using this function call, but it is possible for all customer-specific relations.

```
var values = { itemtext : "Sofa, 2 persons, canvas" };

var where  = "itemnumber = '1234'";

var sqlResult = maconomy::sqlUpdate("Rel", values, where);
```

sqlValidate

Prototype

```
function maconomy::sqlValidate(statement)
```

API Part

SQL Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Validates an SQL statement on the Maconomy server.

Parameters

Name	Type	Description
`statement`	string	Any SQL `select` statement to be executed on the Maconomy server.

Return Value

The return value is an object that contains a Boolean field `ok` that indicates whether the SQL statement is valid. If `ok` is `false` the object also contains a string `message` that describes what prevented the SQL statement from being validated.

Context

All

Remarks

This function is almost identical in behavior to `sqlValidateRestricted`. The difference is that the SQL statement is validated without any access restrictions imposed.

Example

```
var res = maconomy::sqlValidate(sqlStatement);
               if (res.ok)
                 maconomy::sql(sqlSt
                       atement)
               else
                 print(res.m
                 essage);
```

sqlValidateRestricted

Prototype

```
function maconomy::sqlValidateRestricted(statement)
```

API Part

SQL Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Validates an SQL statement on the Maconomy server using Maconomy's access control system.

Parameters

Name	Type	Description
`statement`	string	Any SQL `select` statement to be executed on the Maconomy server.

Return Value

The return value is an object that contains a boolean field `ok` that indicates whether the SQL statement is valid. If `ok` is `false` the object also contains a string `message` that describes what prevented the SQL statement from being validated.

Context

All

Remarks

This function is almost identical in behavior to `sqlValidate`. The difference is that the SQL statement is validated with Maconomy's access restrictions.

Example

```
var res = maconomy::sqlValidateRestricted(sqlStatment);
               if (res.ok)
                 maconomy::sqlRestricted(sqlStatement)
               else
                 print(res.m
                 essage);
```

universeBind

Prototype

```
function maconomy::universeBind(universeHandler, searchBind)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used

Description

This function creates a query handler that is to be used with the `maconomy::mql*` set of functions for data retrieval.

Parameters

Name	Type	Description
`universeHandler`	string	A universe handler that is created by a call to the function `universeOpen`.
`searchBind`	object	Selection of fields and definition of restrictions. This property is an object of type `searchBind`.

Context

All

Throws

The exception `error::stdlib::maconomy::mql::invalidHandler` is thrown if the query handler could not be created. Information about an error is available in the exception.

Remarks

This call does not execute the query. Always remember to call the `maonomy::mqlClose` at some point after a successful call to `maconomy::universeBind`.

Example

```
try  {
                var universeHandler = maconomy::universeOpen("Employee");
                var searchBind = {
                      columnIndexOut
                         put: [
                         EmployeeNum
                         ber,
                         Name1
                      ],
                      restrictions : [
                        { columnName:EmployeeNumber, restriction:[1000..1010]
                        }
                      ]
                    };
                var mqlHandler =
                maconomy::universeBind(universeHandler,search
                Bind); var mqlResult  =
                maconomy::mqlGet(mqlHandler);
```

```
    ...
    maconomy::mqlClose(mqlHan
    dler);
}
catch(e) {
    ...
}
```

universeClose

Prototype

```
procedure maconomy::universeClose(universeHandler)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used

Description

This procedure releases resources that are used by a universe handler. Always remember to call `universeClose` when a universe handler is no longer needed.

Parameters

Name	Type	Description
`universeHandler`	string	A universe handler that is created by a call to the function `universeOpen`.

Context

All

Throws

The exception `error::stdlib::maconomy::universe::invalidHandler` is thrown if the universe handler could not be found. Information about an error is available in the exception.

Example

```
try {
    var universeHandler = maconomy::universeOpen(Employee);
    ...
    maconomy::universeClose(universeH
    andler);
}
catch(e) {
```

```
  ...
}
```

universeGetDef

Prototype

```
function maconomy::universeGetDef(universeHandler)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used

Description

This function gets information about types and titles for fields in the universe defined by the universe handler.

Parameters

Name	Type	Description
`universeHandler`	string	A universe handler that is created by a call to the function `universeOpen`.

Return Value

The return value is an object of type `universeDef`.

Context

All

Throws

The exception `error::stdlib::maconomy::universe::invalidHandler` is thrown if the universe handler could not be found. Information about an error is available in the exception.

Example

```
try  {
  var mqlQuery = 'MQLQUERY';
<MQL 1.3>
MSELECT Name1 FROM Employee
MQLQUERY
  var mqlHandler  = maconomy::mqlOpen(mqlQuery, mqlBind);
  var mqlDef      = maconomy::mqlGetDef(mqlHandler);
  ...
}
```

```
catch(e) {
  ...
}
```

universeOpen

Prototype

```
function maconomy::universeOpen(universename)
function maconomy::universeOpen(universename, interfaceName)
```

API Part

MQL Interface

License

M-Script Reporting (add-on 62)

Custom Universes (add-on 107) necessary if a custom universe is used

Description

This function creates a universe handler. A universe handler can be used for queries where field selection and restrictions are built at run time. A universe handler can also be used for getting field information about fields in a universe. For more information about MQL and universes, see the MQL Language Reference and the MUL Language Reference.

Parameters

Name	Type	Description
`universeName`	string	A universe identifier.
`interfaceName`	string | null	An interface identifier in the selected universe. If no interface identifier is specified, the default interface is used.

Return Value

The return value is a universe handler that are to be used with the other `maconomy::universe*` functions.

Context

All

Throws

The exception `error::stdlib::maconomy::universe::invalidHandler` is thrown if the universe is does not exist. Information about an error is available in the exception.

Remarks

Always remember to call the `maonomy::universeClose` at some point after a successful call to `maconomy::universeOpen`.

Example

```
// Defines a universe handler and ask for field information try   {
                  var universeHandler = maconomy::universeOpen(Employee);
                  var universeDef     =
                  Maconomy::universeGetDef(universeHandler);
                  ...
                  maconomy::universeClose(universeHan
                  dler);
                }
                catch(e) {
                  ...
                }
```

userValueExists

Prototype

```
function maconomy::userValueExists(name)
```

API Part

Value Storage Interface

License

M-Script Reporting (add-on 62)

M-Script Dialog API (add-on 63)

Description

Checks whether a named user value exists in the Maconomy database.

Parameters

Name	Type	Description
`name`	string	The name of the value whose existence should be checked.

Return Value

The return value is of type `bool`. The function returns `true` iff a user value with the specified name exists.

Context

All

Remarks

This function is provided for performance reasons because `getUserValue` can provide the same information. However, calling this function does not transfer the value (if it exists) from the server to the M-Script interpreter. If the value is a complex compound value, the overhead can be

significant. Consequently, this function should be used instead of `getUserValue` if only the question of value existence is of interest.

Call the function `commonValueExists` to check for existence of a common system-wide value.

Example

```
var valueName = "customerPortalUserPrefs";
                if (maconomy::userValueExists(valueName))
                {
                  ...
                }
```

Data Type Reference

This section provides a complete, alphabetically ordered reference description of all object data types used as parameters or return values by the subroutines described earlier.

Each data type is described in a standardized way in its own section. The first part of this section is a sample entry that illustrates how to read a data type reference entry. Read this entry first to get the most out of this chapter.

Sample Entry

Description

First a description of the data type is given. This description explains what the data type is used for.

Properties

All of the properties of the object are described. The description includes the name and type of each property along with a description of each property.

If a property is itself an object, then this type is described in its own reference entry in this chapter.

The type “simple” is used for a parameter that can be of any simple type—in other words, not an `object` or `array`. If a property can be of one of two types then the | symbol is used to separate the types.

A description of an object type could look like this:

Name	Type	Description
`ok`	bool	This property is true *iff* the operation succeeded.
`messages`	array	This is an array of messages returned from the server. Each element of the array is of type `message` described in another section.
`error`	object	This property only exists if `ok` is `false`. The object is of type `message` described in another section.

Example

If the type is an object, then an example with real data is given. In general, the example only expands the object one level. This means that the contents of nested arrays and objects cannot be seen from the example. However, some nested arrays and objects are so small or simple that it makes sense to expand them in the example.

In these cases the general rule is not followed.

An example of a data type could look like this:

```
{
  ok       : false
  messages : [...]
  error    : {...}
}
```

Note that `[...]` denotes a nested array and `{...}` denotes a nested object.

actionDef

Description

This property contains the description of a dialog action. The object is the element type of the `actions` property in the `dialogDef` type.

Properties

Name	Type	Description
`name`	string	The internal name of the action.
`title`	string	The internal name of the action.

Example

```
{
  name  : "Deliver_TimeSheet",
  title : "Submit Time Sheet"
}
```

ActionsEnabled

Description

This object describes which actions are enabled and which are not in the current state of a dialog. It is the type of the `actionsEnabled` property of the type `dialogData`.

Properties

Name	Type	Description
`<actionName>`	bool	The properties are all of type `bool` and are named after the internal names of the actions in the dialog.

Example

```
{
  Deliver_TimeSheet             : true,
  Submit_Time_Sheet_Temporarily : true,
  Release_TimeSheet             : false,
  Approve_TimeSheet             : false,
  Reopen_TimeSheet              : false,
  Copy_Previous_Period          : true
}
```

analyzeFieldDef

Description

The description of the field data in a column of the result of running an Analyzer report is an object. This object is found as the properties of the `columns` property of the `analyzeResult` type.

Properties

Name	Type	Description
`title`	string	The display name of the column. This will in many cases differ from the column name because the column name has been lower cased.
`index`	Int	The index of this column in the `columnindex` property of `analyzeResult`.
`type`	string	The index of this column in the `columnindex` property of `analyzeResult`.
`isKey`	bool	This property is `true` if the column is a sorting key and `false` if the column is a selected field.
`Sum`	real	The sum of all values returned in this column.This property is meaningful for numerical columns only.
`max`	real	The maximum value returned in this column.This property is meaningful for numerical columns only.

Example

```
{
  title : "Total Working Hours",
  index : 1,
  type  : "real",
  isKey : false,
  sum   : 3,749.00,
  max   : 1,885.50
}
```

analyzeResult

Description

The object is the type of the `result` property of the type `analyzeReturn`.

Properties

Name	Type	Description
`columnindex`	array	An array with elements of type `string`. Each `string` is the name of a column extracted when running the Analyzer report. The order of the entries is significant.
`columns`	object	An object with a description of each column in the result. The property names match the names found in `columnindex`. Each property in the object is an object of type `analyzeFieldDef`.
`rows`	array	The rows fetched from the Maconomy system. Each entry is an object of type `record`.

Example

```
{
  columnindex  :  ["employee no.", "total working hours"],
  columns      :  {
                  'employee no.'        : {...},
                  'total working hours' : {...}
                   }
  rows         :   [. . .]
}
```

analyzeReturn

Description

The object is the return value when executing an Analyzer report by calling `maconomy::analyze`.

Properties

Name	Type	Description
`result`	object	The data returned from the Analyzer. This includes a column description as well as the rows fetched from the Maconomy system. The object is of type `analyzeResult`.

Example

```
{
  result  :  {...}
}
```

connectionInfo

Description

The object is the return value when calling `maconomy::getConnectionInfo`.

Properties

Name	Type	Description
`ssl`	bool	This property is true *`iff`* the server connection is protected by SSL.

Example

```
{
  ssl  :  true
}
```

dialogData

Description

The object contains the current data in a dialog. The object is the type of the `dialogData` property of the type `dialogReturn`.

Properties

Name	Type	Description
`actionsEnabled`	object	An object containing the state (enabled/disabled) of all actions in the dialog. The property is an object of type `actionsEnabled`.
`layouts`	array	An array that contains the name (string) of all the available print layouts for the dialog. The names in this array correspond to the possible values of a “LayoutNameType” popup associated with the dialog.
`readOnly`	bool	This property is `true` *iff* the data in the dialog are read-only. (This happens if another user holds a high-level lock on the dialog entry.)
`usedBy`	object	Name of user locking the dialog which you attempted to open. Returned if `readOnly` (above) is `true` and if the user is known to the API.
`upperPane`	object	The data in the upper pane. This property is an object of type `dialogPaneData`.
`lowerPane`	object	The data in the lower pane. This property is an object of type `dialogPaneData`.

Example

```
{
  actionsEnabled : {...},
  layouts        : [ "standard" ],
  readOnly       : false,
  upperPane      : {...},
```

```
  lowerPane       : {...}
}
```

dialogDef

Description

The object is the static description of a dialog and is returned by `dialogGetDef`.

Properties

Name	Type	Description
`actions`	array	An array of actions in this dialog. Each element is of the type `actionDef`. Not available for RGL and print dialogs.
`printName`	string	The internal name of the associated print. This property is the empty string if the dialog has no associated print. Not available for RGL and print dialogs.
`printLayoutName`	string	The internal name of the layout of the associated print. This property is the empty string if the dialog has no associated print. Not available for RGL and print dialogs.
`type`	string	The type of the dialog. Possible values are: “card”, “card/table”, “MSL” (which means RGL report), and “parameter” (which means print dialog).
`name`	string	The internal name of the dialog. (`DialogTitle`).
`title`	string	The external name of the dialog. (`WindowTitle`).
`highLevelLock`	bool	This property is `true` *iff* the dialog use high- level locking. Not available for RGL and print dialogs.
`readAccess`	bool	This property is `true` *iff* the user has access to read existing records. Not available for RGL and print dialogs.
`newAccess`	bool	This property is `true` *iff* the user has access to create new records. Not available for RGL and print dialogs.
`updateAccess`	bool	This property is `true` *iff* the user has access to update existing records. Not available for RGL and print dialogs.
`deleteAccess`	bool	This property is `true` *iff* the user has access to delete existing records. Not available for RGL and print dialogs.
`upperPane`	object	A description of the upper pane of the dialog. The property is of type `dialogPaneDef`.
`lowerPane`	object	A description of the lower pane of the dialog. This property is only present for two-pane dialogs. The property is of type `dialogPaneDef`.

Example

```
{
  actions         : [...],
  printName       : "Print_TimeSheets",
  printLayoutName : "Standard",
  type            : "card/table",
  name            : "TimeSheets",
  title           : "Time Sheets",
  highLevelLock   : true,
  readAccess      : true,
  newAccess       : true,
  updateAccess    : true,
  deleteAccess    : true,
  upperPane       : {...},
  lowerPane       : {...}
}
```

dialogError

Description

The value of the `...::maconomy::dialog` exception family. Each exception has the type `error::stdlib::maconomy::dialog::`<kind> where <kind> is the value also found in the property `status.error.kind`.

Properties

Name	Type	Description
`message`	string	An error message.
`status`	object	A status object containing detailed error information. This object has the type of a `DialogStatus` object without the `ok` property.

Example

```
{
 type:[ "error", "stdlib",
  "maconomy", "dialog",
  "RecordNotFound"


 ],
 value:{
   status:{
```

```
      messages:[
      ],
      redrawAll:false,
          error:{
            kind:"RecordNotFound"
          }
      },
      message:"error in 'maconomy::dialogRead()'"
    },
     ...
}
```

dialogFieldDef

Description

The object is the static description of a field in a dialog pane. This is the type of each of the properties in the `columns` property of the type `dialogPaneDef`.

Properties

Name	Type	Description
index	int	The index of the field in the `columnindex` property of `dialogPaneDef`.
title	string	The display name of the field. (Since Maconomy does not support field labels, the display name equals the internal name.)
type	string	The type of the field. The property is the name of the M-Script type—for example, “string” or “int.”
kind	string	This property is “database” if the field is a database field and “variable” if the field is a variable field.
secret	bool	This property is `true` *iff* the field is a secret field—a field whose contents should not be displayed on a screen. This could, for example, be a password field.
mandatory	bool	This property is `true` *iff* the field is mandatory—a field that must have a non-blank value.
openNew	bool	This property is `true` *iff* the field is open in the New state—the NewUpper state (if this is an upper pane field) or the NewLower state (if this is a lower pane field). Not available for RGL and print dialogs.
openUpdates	bool	This property is `true` *iff* the field is open in the Exist state. Not

Name	Type	Description
		available for RGL and print dialogs.
`openInit`	bool	This property is true *iff* the field is open in the initial state of a print dialog. Not available for normal dialogs.
`favorites`	array	Array of favorite values for this field. The array contains one object of type `dialogFieldFavorite` for each of the favorites. This format makes it possible to transfer the favorite list directly to the GUI.

Example

```
{
  index      : 0,
  title      : "EmployeeNumber",
  type       : "string",
  kind       : "database",
  secret     : false,
  mandatory  : true,
  openNew    : true,
  openUpdate : false
}
```

dialogFieldFavorite

Description

The object is the static description of a favorite value for a field in a dialog pane. This is the type of each of the properties in the `favorites` array property of the type `dialogFieldDef`.

The format of dialog favorites is compatible with the input format for combo boxes in the GUI. This means that using the result of `dialogGetDef` for the component specification when opening a GUI window will cause all fields with favorites to be displayed as combo boxes.

Please refer to the M-Script GUI-II API Reference for more information.

Properties

Name	Type	Description
`title`	string	The title describing this favorite entry. This is the title that will be assigned to the combo box entry in the GUI.
`value`	simple	The favorite value for the field. This is the value that will be inserted in the field if the user selects the corresponding combo box entry in the GUI.
`values`	object	A case-insensitive object of values to assign to other fields in the pane when selecting this combo box entry in the GUI. The

Name	Type	Description
		property names are the names of the fields that should have values assigned if this favorite entry is selected. The property values are the values to transfer. The object always contains the property corresponding to the current field. The GUI does not recognize this property but silently ignores it.

Example

```
{
  title  : "JobNumber",
  value  : "424242",
  values :
  @{
    JobNumber      : "424242",
    ActivityNumber : "250",
    TaskName       : "General"
  }
}
```

dialogMessage

Description

This object is either a notification or an error message returned from a dialog operation. This is the type of each of the elements of the property `message` and the type of the property `error` of the type `dialogStatus`.

The property `text` is not used by all messages.

The properties `pane` and `field` are only used by some messages to indicate the field that caused the notification or error.

All possible notifications and error messages are described below. For each message, you can see which of the optional properties are used (if any) and why.

Properties

Name	Type	Description
kind	string	A string identifying the kind of message.
text	string	An optional property containing a message to display to the user.
pane	string	An optional property identifying a pane.Possible values are “upper” and “lower.”

Name	Type	Description
`field`	string	An optional property identifying a field by its name.
`rollback`	bool	In case of an error, this field indicates whether a rollback has been performed on the server.

Example

```
{
  kind  : "ApplicationError",
  text  : "Please enter an activity",
  pane  : "lower",
  field : "ActivityNumber"
}
```

Notifications

The Maconomy server can return the following notifications.

Kind	Text	Pane/Field	Description
`Notification`	yes	optional	The application sent a notification to the user.
`Warning`	yes	–	A warning was issued by the application. Please refer to the description of `setWarningReply`.

Errors

Kind	Text	Pane/Field	Description
`ApplicationError`	yes	optional	The application sent an error message to the user.
`FatalApplicationError`	yes	–	A fatal error occurred in the application.
`SystemError`	yes	–	An internal system error occurred.
`RecordNotFound`	–	–	A call to `dialogGet` failed. The record specified by `upperKey` does not exist.
`DataChanged`	–	–	Since the last operation performed on this dialog, data was changed by another user (or in another dialog referring to the same Maconomy dialog).
`DataDeleted`	–	–	Since the last operation performed on this dialog, data was deleted by another user (or in another dialog referring to the same

Kind	Text	Pane/Field	Description
			Maconomy dialog).
`KeyExists`	–	–	A call to `dialogPutUpper` or `dialogPutLower` failed. A record with the same key already exists.
`RecordInUse`	yes	–	A call to `dialogPutUpper` or `dialogPutLower` failed. A record with the same key already exists and is in use by another user (or in another dialog referring to the same Maconomy dialog). The text contains the user name of the other user.
`MissingMandatoryField`	–	yes	A mandatory field was not given a non-blank value.
`ActionDisabled`	–	–	An attempt was made to execute an action that is disabled.
`ClosedField`	–	yes	An attempt was made to modify the value of a closed field.
`AnsweredNoToWarning`	–	–	A check warning was encountered and the Dialog Interface answered “No”. Please refer to the description of `setWarningReply`.
`MissingInputFile`	–	–	A dialog operation requested an input file but no file was (or not enough files were) enqueued. Please refer to the description of `fileEnqueue`.
`ExceedingInputFiles`	–	–	More input files were enqueued than requested by the dialog operation. Please refer to the description of `fileEnqueue`.

dialogPaneAccess

Description

This object describes which dialog operations can be performed in the current dialog state. This is the type of the `access` property of the type `dialogPaneData`.

Properties

Name	Type	Description
`newAccess`	bool	This property is `true` *iff* a new operation can be performed in this pane in the current state. Prior to version 6.0 the name was “new.”

Name	Type	Description
`updateAccess`	bool	This property is `true` *iff* an update operation can be performed in this pane in the current state. Prior to version 6.0 the name was “update.”
`deleteAccess`	bool	This property is `true` *iff* a delete operation can be performed in this pane in the current state. Prior to version 6.0 the name was “delete.”
`findAccess`	bool	This property is `true` *iff* finding is enabled for this pane in the current state. (Finding is not supported by the API). Prior to version 6.0 the name was “find.”

Example

```
{
  newAccess    : true,
  updateAccess : false,
  deleteAccess : true,
  findAccess   : true
}
```

dialogPaneData

Description

This object contains the current data in a dialog pane. This is the type of the properties `upperPane` and `lowerPane` of the type `dialogData`.

Properties

Name	Type	Description
`access`	object	Describes which operations can be performed on the current pane in the current state. The property is an object of type `dialogPaneAccess`.
`rows`	array	This array contains the records in the pane. Card panes contain exactly one row while table panes can contain any number of rows, including zero. The array elements are of type `record`.

Example

```
{
  access : {...},
  rows   : [...]
}
```

dialogPaneDef

Description

This object contains the static description of a dialog pane. It is the type of the properties `upperPane` and `lowerPane` in the type `dialogDef`.

Properties

Name	Type	Description
`type`	string	The type of the pane. Valid values are “card,” “table,” “MSL” (meaning RGL), “parameter” (meaning print).
`columnIndex`	array	An array with elements of type `string`. Each `string` is the name of a field in the pane. The order of the entries is significant.
`columns`	object	An object with a description of each field in the pane. The property names match the names found in `columnIndex`. Each property in the object is an object of type `dialogFieldDef`.
`relationName`	string | null	The name of the relation corresponding to this pane. If no relation corresponds to this pane, the property has the value `null`. Not available for RGL and print dialogs.
`keyDef`	array	An array of strings containing the names of the key fields in this pane. Not available for RGL and print dialogs.
`newSpelling`	string	The external name for a new entry in this pane. (In some languages “new” can depend on the gender of the word describing the entries in the pane.) Not available for RGL and print dialogs.
`name`	string	The internal name of the pane.
`title`	string	The external name of the pane.
`searches`	object	A `dialogSearchDef` object.

Example

```
{
  type         : "card",
  columnIndex  : [...],
  columns      : {...},
  relationName : "TimeSheetHeader",
     keyDef       : [...],
     newSpelling  : "New Time Sheet",
```

```
    name          : "TimeSheetHeader",
    title         : "Time Sheet"
}
```

dialogReturn

Description

This object contains data retrieved from the Maconomy database for a specific dialog. It is the return value of all Dialog Interface functions corresponding to dialog operations other than open, delete, and close.

Properties

Name	Type	Description
`status`	object	A status field informing about success or failure of the operation. This property is an object of type `dialogStatus`.
`dialogData`	object	The data contained in the dialog after the operation. If the operation failed, this prop- erty does not exist and the `status` property will indicate the error. This property is an object of type `dialogData`.

Example

```
{
  status      : [...],
  dialogDate  : {...},
}
```

dialogSearchDef

Description

This object describes the Maconomy Foreign Key Searches defined for a dialog. The format of this object corresponds exactly to that of a GUI-II search specification, and may be used to implement automatic Maconomy searches in the GUI.

Properties

The `dialogSearchDef` object contains one property for each of the search specifications. The name of this property is not used for anything. Each property is an object with the following properties:

Name	Type	Description
`title`	string	Search title.
`fields`	array	Array of field names corresponding to the fields for which this search is defined.

Example

```
searches
{
  s0
    {
      title:”Find Timesheet”,
      fields:  [  “Employeenumber”,  “Periodstart”  ]
    }
   S1:  ...
}
```

dialogStatus

Description

This object informs of success or failure of a Dialog Interface call. This is the type of the `status` property in the type `dialogReturn`.

Properties

Name	Type	Description
`ok`	bool	This property is `true` *iff* the dialog operation succeeded.
`messages`	array	An array containing messages returned from the server. Each array element is an object of type `dialogMessage`.
`redrawAll`	bool	This property is `true` if a table operation has affected other table rows than the current one.
`error`	object	This property exists *iff* `ok` is `false` and is an object of type `dialogMessage`.
`files`	array	This property exists *iff* one or more output files are produced by the dialog operation. The array elements are file handles of type `string`. These can be used in calls to `fileDelete`, `fileEnqueue`, and `fileGetData`.
`prints`	array	This property exist *iff* one or more prints are produced by the dialog operation. The array elements are print handles of type `string`. These can be used in calls to `printGetPdf` and `printDelete`.
`rowCount`	int	This property exists when calling `dialogDeleteLower` and returns the number of rows actually deleted. When deleting rows via a partial key, the number of rows deleted may be greater than one. If no rows match the specified key, the value is zero.

Example

```
{
  ok       : false,
  messages : [...],
  redrawAll: false,
  error    : [...],
  files    : [...],
  prints   : [...],
  rowCount : [...],
}
```

fieldValue

Description

The value of a field in a `record`.

The `fieldValue` is an object with one property named `value`, which contains the actual data. From M-Script version 9.1, it is possible to use the data directly instead of wrapping it in a `fieldValue` object.

Properties

Name	Type	Description
`value`	simple	The value of the field.

Example

```
{
  value  :  1,885.50
}
```

hllLock::invalidKey

Description

The value of the `...::maconomy::hllLock::invalidKey` exception. This exception is thrown by `hllLock` when the key does not have values for all key fields in the specified external relation.

Properties

Name	Type	Description
`message`	string	An error message.
`keyField`	string	Name of the missing key field.

Example

```
{
```

```
  value:  {
    message  : "maconomy:hllLock: Invalid key. Missing field
     'Componentid'",
    keyField :  "Componentid"
  }
}
```

hllLock::invalidLockId

Description

The value of the `...::maconomy::hllLock::invalidLockId` exception. This exception is thrown by `hllRenewLock` or `hllUnlock` if the `lockId` argument is an invalid `lockId` object. This will not occur if the return value from `hllLock` is passed as an argument to the function or procedure.

Properties

Name	Type	Description
`message`	string	An error message.

Example

```
{
  value:  {
    message  : "invalid lockId"
  }
}
```

hllLock::invalidRelation

Description

The value of the `...::maconomy::hllLock::invalidRelation` exception. This exception is thrown by `hllLock` when the specified external relation name does not exist.

Properties

Name	Type	Description
`message`	string	An error message.
`relationName`	string	Name of the invalid relation.

Example

```
{
  value:  {
    message  : "maconomy:hllLock:  unknown relation specified
     'bogusrelationname'",
```

```
    relationName :  "bogusrelationname"
  }
}
```

hllLock::locked

Description

The value of the `....::maconomy::hllLock::locked` exception. This exception is thrown by `hllLock` when the specified record is already locked.

Properties

Name	Type	Description
`message`	string	An error message.
`otherUser`	string	Name of the user currently holding the lock.

Example

```
{
  value:  {
    message  : "maconomy:hllLock:  entry already locked by
     'Administrator'",
    otherUser : "Administrator"
  }
}
```

hllLock::lockLost

Description

The value of the `....::maconomy::hllLock::lockLost` exception. This exception is thrown by `hllRenewLock` when the lock is lost and cannot be obtained because it is locked by another user.

Properties

Name	Type	Description
`message`	string	An error message.

Example

```
{
  value:  {
    message  : "maconomy:hllRenewLock:  lock lost (lock held by
     'Administrator'",
    otherUser : "Administrator"
  }
```

```
}
```

lockId

Description

The object is the return value when calling `maconomy::hllLock`.

The `lockId` is an object with one property named `lockId`, which contains the actual lock ID.

Properties

Name	Type	Description
`lockId`	string	Unique identifier of current record.

Example

```
{
  lockId  :  2100
}
```

licenseError

Description

The value of the `...::maconomy::license` exception.

Properties

Name	Type	Description
`message`	string	An error message.
`license`	object	Object containing the license requirements as described below.

The license object contains these properties.

Name	Type	Description
`addOns`	array	Array of arrays of add-on numbers. The outer array level combines the inner array level with “OR” requirements. The inner array level combines the numbers with “AND” requirements (disjunctive normal form).
`customerNumbers`	array	Array of required customer numbers.

Example

```
{
  type:[
  "error",
```

```
  "stdlib",
  "maconomy",
  "license"
  ],
  value:{
    message:"No license to run this script. Customer number
  requirements: none. Add-on requirements: (230 and 231) or (240)",
  },
  license:{
      customerNumbers: [],
      addOns: [ [230,231], [240] ]

     }
  }
}
```

loginReturn

Description

The return value from calling `maconomy::login` or `maconomy::singleLogin` is an object.

Properties

Name	Type	Description
`ok`	bool	This property is `true` *iff* the login attempt succeeded.
`userName`	string	The Maconomy user name used in the login.
`message`	string	This property contains a message if the login attempt failed. Otherwise this property is the empty string.
`errorCode`	int	An error code indicating the result of the login attempt: 0 = No error. 1 = Invalid user. 2 = User not activated. 3 = User expired. 4 = Invalid password. 5 = Password expired. 6 = Access blocked. 7 = Too many users.

Example

```
{
  ok        :  false,
  message   :  "Password is not correct",
  errorCode :  4
}
```

loginError

Description

The value of the `...::maconomy::login` exception.

Properties

Name	Type	Description
`userName`	string	Name of user for which the login was attempted.
`message`	string	An error message.
`errorCode`	int	An error code indicating the result of the login attempt.

Example

```
{
  type:[
  "error",
  "stdlib",
  "maconomy",
  "login"
  ],
  value:{
    message:"This user does not exist",
    errorCode:1
  }
  ...
}
```

mqlBind

Description

An object used by the MQL Interface. The object contains actual values for query parameters and page size information.

Properties

Name	Type	Description
`pageSize`	integer | null	Number of rows per page when page oriented data retrieval is used. If no page size is given, a default value of 50 is used.
`parameters`	@object | null	A case insensitive object containing properties matching the parameters specified in an MQL query command. If a parameter is not specified in the object, the default value specified in the query is used. Each of these properties is of type `fieldValue`.

Example

```
{
  pageSize:10,
  parameters : @{
    parmEmployeeNumberFrom : {value : "1000"},
    parmEmployeeNumberTo   : {value : "1010"}
  }
}
```

mqlDef

Description

This object contains static information about a query. The object is used by the MQL Interface and describes the fields selected in the query defined by the used query handler. The object is constructed without executing the query.

Properties

Name	Type	Description
`pageSize`	array	An array with elements of type `string`. Each `string` is the name of a column extracted when executing the query. The order of the entries is significant.
`parameters`	@object	An object with a description of each column in the result. The property names match the names found in `columnindex`. If two columns exist with the same name, a suffix is added to the property name, making the property name unique. Each property in the object is an object of type `mqlFieldDef`.

Example

```
{
  columnIndex:[
  "EmployeeNumber",
```

```
  "Name1"
  ],
  columns:@{
    EmployeeNumber:{
    index:0,
    type:"string",
    title:"Employee No."
  },
  Name1:{
    index:1,
    type:"string",
    title:"Name 1"
   }
  }
}
```

mqlFieldDef

Description

The description of the field data in a column defined by a query handler. The value is an object. This object is found as the properties of the `columns` property of the `mqlDef` type.

Properties

Name	Type	Description
`index`	int	The index of this column in the `columnIndex` property of `mqlDef`.
`title`	string	The display title of the column.
`type`	string	The type of the values in this column. The property is the name of the M-Script type—e.g. “string” or “int.”

Example

```
{
  index:1,
  type:"string",
  title:"Employee No."
}
```

mqlResult

Description

The return value of a call to one of the `maconomy::get*` functions. The value is an object.

Properties

Name	Type	Description
`more`	bool	Flag indicating if more rows exist. Used for page oriented data retrieval.
`page`	integer	Current page number. Used for page oriented data retrieval.
`mqlData`	object	The rows fetched from the Maconomy system. The object is of type `mqlRows`.

Example

```
{
  more:false,
  page:1,
  mqlData:{...}
}
```

mqlRows

Description

An object containing rows. Used by the MQL Interface.

Properties

Name	Type	Description
`rows`	array	The rows fetched from the Maconomy system. Each entry is an object of type `record`.

Example

```
{
  rows:[
    @{
      EmployeeNumber:{ value:"1000" },
      Name1:          { value:"James Hanson"}
    },
    @{
      EmployeeNumber:{ value:"1000" },
```

```
      Name1:          { value:"Bill Jackson"}
    }
  ]
}
```

readValueReturn

Description

The return value of the `readvalue` function, which preprocesses the input stream before passing it to the M-Script parser.

Properties

Name	Type	Description
`ok`	bool	This property is `true` *iff* the preprocessed command is valid. In this case, the `message` property does not exist.
`message`	string	An optional message describing why the command failed preprocessing. This property exists only when `ok` is `false`.
`value`	any	The value of the successfully preprocessed command. If `ok` is `false`, this property is `null`.

Example

See `readvalue` or `file::readvalue`.

record

Description

An object that contains data from the Maconomy system. A record contains the data corresponding to one row fetched using the MQL Interface, SQL Interface, the Analyzer Interface, or the Dialog Interface.

A key used to identify a dialog entry is also of this type. In this case, the record need only contain values for the fields defined as key fields.

A record passed to `dialogUpdateUpper`, `dialogUpdateLower`, `dialogPutUpper`, `dialogPutLower`, and `rglExecute` is also of this type. In this case the record need only contain the modified fields.

For each field in the record this object contains a property whose name is the name of the field. Each of these properties are of type `fieldValue`—but from version 9.1 of M-Script, it is possible to supply the data directly instead of wrapping it in a `fieldValue` object.

As of version 6.0 of M-Script, the records returned from `sql` and `analyze` reports are case insensitive objects. In addition to this, records returned from all other dialog functions are case insensitive objects when used with a Maconomy 7.0 application (or newer).

Example

```
{
```

```
  itemnumber : { value : "1234" },
  itemtext1  : { value : "Sofa, 2 persons, leather" },
  name       : "John" // Shorthand
}
```

rglID

Description

An object that contains status information and an RGL identifier returned from `rglOpen`.

Properties

Name	Type	Description
`ok`	bool	Status of the dialog open—a false value means the operation failed.
`message`	string	Error message—is `null` if `ok` is `true`.
`id`	string	The report dialog identifier.

rglList

Description

An array that contains a list of RGL reports. Each entry in the array is an object with the following properties:

Properties

Name	Type	Description
`title`	string	The external name of the report. This is for end-user presentation.
`name`	string	The internal name of the report.
`group`	string	The name of the group to which this report belongs.

Example

```
{
  {
    title:"M-Script 1",
    name:"MScript1",
  },
  {
    title:"M-Script 2",
```

```
    name:"MScript2",
    group:"M-Script"
  },
  {
    title:"M-Script 3",
    name:"MScript3",
    group:"M-Script"
  }
}
```

searchBind

Description

The object contains static information about a foreign key search associated with a search handler.

Properties

Name	Type	Description
`title`	string	Title of the search window.
`columnIndex`	array	An array with elements of type `string`. Each `string` is the name of a column. The order of the entries is significant. The array is a list of all the columns in the associated search window.
`columnIndexDefaultOutput`	array	An array with elements of type `string`. Each `string` is the name of a column. The order of the entries is significant. The array indicates the default columns open for user restrictions in the associated search window.
`columnIndexKeys`	array	An array with elements of type `string`. Each `string` is the name of a column. The order of the entries is significant. The array indicates the key columns open in the associated search window.
`columnIndexTarget`	array	An array of objects indicating target for key fields in dialog. The entries are objects of the type `searchTarget`.
`columns`	@object	An object with a description of each column in the search window. Each property in the object is an object of type `mqlFieldDef`.

Example

```
{
```

```
  columnIndexOutput: [
    "EmployeeNumber",
    "Name1"
  ],
  restrictions : [
    { columnName: "EmployeeNumber", restriction:["1000..1010"] }
  ],
    orders:  [
    { columnName: "EmployeeNumber", order: "ascending" }
    ]
}
```

searchDef

Description

An object that selects fields and defines restrictions. The object is used by the MQL Interface.

Properties

Name	Type	Description
`columnIndexOutput`	array	An array with elements of type `string`. Each `string` is the name of a column to be selected in the query. The order of the entries is significant.
`restrictions`	array|null	An array with elements of objects of type `searchRestriction`. The list of restrictions defines the restrictions to be applied to the query. The restrictions are on concoctive normal form, that is the “and” operation is implicitly applied between the restrictions.
`orders`	array|null	An array with elements of type `searchOrder`. The order of elements is significant. The list defines the order applied to the query.
`mqlBind`	object|null	Bind data for the query, for example, the definition of page size for page oriented data retrieval. This property is an object of type `mqlBind`.

Example

```
{
  title:"Find Company",
    columnIndex:[
    "CompanyNumber",
    "Name1",
```

```
  "Name2",
  "Name3",
  ...
 ],
 columnIndexDefaultOutput:[
 "CompanyNumber",
 "Name1",
 "Name2",
 "Name3"
 ],
 columnIndexDefaultRestriction:[
 "CompanyNumber",
 "Name1"
],
columnIndexKeys:[
"CompanyNumber"
],
columnIndexTarget:[
{ columnName:"CompanyNumber", targetName:"CompanyNumber" }
],
columns:{
CompanyNumber:{
index:0,
type:"string",
title:"Company No."
},
Name1:{
index:1,
type:"string",
title:"Name 1"
},
Name2:{
index:2,
type:"string",
title:"Name 2"
},
```

```
  Name3:{
  index:3,
  type:"string",
  title:"Name 3"
  },
  ...
 }
}
```

searchOrder

Description

The object selects the sorting order. The object is used by the MQL Interface.

Properties

Name	Type	Description
`columnName`	string	The identifier of a column.
`order`	string	Order selection for the column. Legal values are `"ascending"` and “`descending`”.

Example

```
{
  columnName:"EmployeeNumber",
  order:"ascending"
}
```

searchRestriction

Description

The object defines a restriction. The object is used by the MQL Interface.

Properties

Name	Type	Description
`columnName`	string	The identifier of a column.
`restriction`	array	An array with literals matching the type of the column selected by the `columnName` property. For numbers, the literal is of type `string` where the Maconomy way of restriction specification can be used (for example, “`1000..1020`” or “`1000..`”). The “or” operation is implicitly used between the elements in the array.

Example

```
{
  columnName:"EmployeeNumber",
  restriction:["1000..1010", "1020.."]
}
```

searchSpec

Description

Object for definition of a foreign key search handler. Contains selection of a column in a dialog pane.

Properties

Name	Type	Description
`dialogName`	string	The internal name of the dialog.
`pane`	string	The pane identity. Possible values are “upper” and “lower.”
`columnName`	string	The identifier of a column in the pane of the selected dialog.
`currentRecord`	object	An object where the property names specify column names. Each entry is an object of type `record`.

Example

```
{
  dialogName    : "Employees",
  pane          : "upper",
  columnName    : "CompanyNumber",
  currentRecord : @{ EmployeeNumber : {value:"11"} }
}
```

searchTarget

Description

The object selects a target key column in a dialog. The object is used by the Dialog Interface.

Properties

Name	Type	Description
`columnName`	string	The identifier of a column.
`targetName`	string	The identifier of the target key column in the dialog.

Example

```
{
  columnName:"EmployeeNumber",
  targetName:"SeniorEmployeeNumber"
}
```

serverInfo

Description

An object containing information about the Maconomy server to which the interpreter is connected. Returned by the function `getServerInfo`.

Properties

Name	Type	Description
`server`	object	Information about the server version.
`application`	object	Information about the Maconomy application.
`installation`	object	Information about the Maconomy installation.
`shortname`	string	Shortname for the database.
`language`	string	Language code for the database.
`database`	string	Name of the database server.

The `server` object contains the following fields:

Name	Type	Description
`raw`	string	A string containing the raw version code of the server.
`major`	int	The major version number of the server.
`minor`	int	The minor version number of the server.

The `application` object contains the following fields:

Name	Type	Description
`raw`	string	A string containing the raw version code of the Maconomy application.
`country`	string	The country code for the Maconomy application.
`major`	int	The major version number of the Maconomy application.
`minor`	int	The minor version number of the Maconomy application.

Name	Type	Description
patch	int	The patch level of the Maconomy application.
subpatch	int	The patch sublevel of the Maconomy application.

The `installation` object contains the following fields:

Name	Type	Description
companyName	string	The company name.
. . .	string	The name of the original licensed company (for multi-company installations).
registrationNo	string	The registration number.

Example

```
{
  server:{
    raw:"36.00.0.0",
    major:36,
    minor:0
},
  application:{
    raw:"DK_8_0",
    country:"DK",
    major:8,
    minor:0,
    patch:0,
    subpatch:
  },
  installation:{
    companyName:Maconomy A/S,
    originalCompanyName:Maconomy Group,
    registrationNo:723211915
  },
  shortname:"dk80",
  language:"DK",
  database:"oracle"
}
```

sqlFieldDef

Description

The description of the field data in a column of the result of executing an SQL statement. The value is an object. This object is found as the properties of the `columns` property of the `sqlResult` type.

Properties

Name	Type	Description
`title`	string	The display name of the column. This will in many cases differ from the column name because the column name has been lower cased.
`index`	int	The index of this column in the `columnIndex` property of `sqlResult`.
`type`	string	The type of the values in this column. The property is the name of the M-Script type; for example, “string” or “int.”

Example

```
{
  title : "NAMEOFUSER",
  index : 0,
  type  : "string "
}
```

sqlModifyReturn

Description

The object is the return value when calling `sqlInsert`, `sqlUpdate`, or `sqlDelete`.

Properties

Name	Type	Description
`ok`	bool	This property is `true` *iff* the modification succeeded.
`rows`	int	The number of rows modified.

Example

```
{
  ok   : true,
  rows : 1
}
```

sqlResult

Description

The `result` property from the return value of a call to `sql` or `sqlRestricted`. The value is an object.

Properties

Name	Type	Description
`columnindex`	array	An array with elements of type `string`. Each string is the name of a column extracted when executing an SQL statement. The order of the entries is significant.
`columns`	object	An object with a description of each column in the result. The property names match the names found in `columnindex`. Each property in the object is an object of type `sqlFieldDef`.
`rows`	array	The rows fetched from the Maconomy system. Each entry is an object of type `record`.

Example

```
{
  columnindex  : [...],
  columns      : [...],
  rows         : [...],
}
```

sqlReturn

Description

An object that contains data from a call to `sql` or `sqlRestricted`.

Properties

Name	Type	Description
`result`	object	The data returned when executing an SQL statement. This includes a column description as well as the rows fetched from the Maconomy system. The object is of type `sqlResult`.

Example

```
{
  result : {...}
```

```
}
```

ssoProperties

Description

Object returning the status of the Single Sign On functionality by `getSSOProperties()`.

Properties

Name	Type	Description
`ok`	bool	Indicates whether the status query went OK.
`realm`	string	The name of the realm (domain) under which the Maconomy service has been registered, taken from the Kerberos setup file.
`realms`	string	The list of known realms and associated Key Distribution Centers, taken from the Kerberos setup file if a list of realms has been specified there; if not, this property is empty.
`kdc`	string	The network name or IP of the Kerberos Key Distribution Center, taken from the Kerberos setup file.
`serviceName`	string	The service name: the name of the service Maconomy has been registered as at the authenticator, taken from the Kerberos setup file.

Example

```
{
    ok: true
    realm: "example.com",
    realms: "example.com=kdc.exampleexample2=kdc2.example.com",
  kdc: "kdc.example.com”,
    serviceName: "maconomy/test.example.com"
}
    realm: "example.com",
```

systemParameterReturn

Description

The return value of the `getSystemParameter` function, which tests whether a given system parameter is set or not.

Properties

Name	Type	Description

Name	Type	Description
`ok`	bool	This property is `true` *iff* the query went OK.
`value`	bool	The true/false value of the system parameter. If `ok` is `false`, this property is `null`.
`error`	string	Error message if the system parameter does not exist. This property exists only when `ok` is `false`.

Example

```
{
  ok: true,
  value: true
}
```

universeDef

Description

An object containing static information about a universe. The object is used by the MQL Interface and describes the fields available for selection from a universe defined by a universe handler.

Properties

Name	Type	Description
`name`	string	The identifier of the universe used for internal reference.
`title`	string	The title of the universe.
`description`	string	A very short description of what the universe can be used for.
`summary`	string	The summary is a short version of the help text.
`helpText`	string	The help text is for an in-depth description of what the universe can be used for, and how it works.
`interface`	string	The identifier of the used universe interface.
`index`	array	An array with elements of type `string`. Each string is the name of a field or a group. The order of the entries is significant.
`groups`	@object	An object with a description of each group in the universe. The property names match the names found in index. Each property in the object is an object of type `universeGroupDef`.

Name	Type	Description
`columns`	@object	An object with a description of each field in the universe. The property names match the names found in index. If two columns exist with the same name, a suffix is added to the property name, making the property name unique. Each property in the object is an object of type `universeFieldDef`.

Example

```
{
  name:"jobUniverse",
  title:"Job Universe",
  description:"",
  summary:"",
  helpText:"",
  interfaceName:"default", index:[
  index:[
    "Employee",
    "Activity",
    "NumberOfWeekSUM",
    "PlanOfWeekSUM"
  ],
  groups:@{
    Employee:{
    title:"Employee",
    index:[
      "Employee.EmployeeNumber",
      "Employee.EmployeeName"
    ]
  },
  Activity:{
    title:"Activity",
    index:[
      "Activity.ActivityNumber",
      "Activity.ActivityText"
    ]
  },
},
columns:@{
```

```
NumberOfWeekSUM:{
  type:"real",
  title:"Registration Time",
  description:"",
  groupValue:true
},
PlanOfWeekSUM:{
  type:"real",
  title:"Planned Time",
  description:"",
  groupValue:true
},
Employee.EmployeeNumber:{
  type:"string",
  title:"Employee No.",
  description:"",
  groupValue:false
},
Employee.EmployeeName:{
  type:"string",
  title:"Name 1",
  description:"",
  groupValue:false
},
Activity.ActivityNumber:{
  type:"string",
  title:"Activity No.",
  description:"",
  groupValue:false
},
Activity.ActivityText:{
  type:"string",
  title:"Activity",
  description:"",
  groupValue:false
```

```
      }
    }
  }
```

universeFieldDef

Description

A description of the field data in a universe defined by a universe handler. The value is an object. This object is found as the properties of the `columns` property of the `universeDef` type.

Properties

Name	Type	Description
`title`	string	The display title of the field.
`description`	string	A very short description of what the field can be used for.
`groupValue`	bool	Flag indicating if the field is a group value field. A group value field is defined using one of the row group functions available in MQL. The available row group functions are `SUM`, `MIN`, `MAX`, `AVG` and `COUNT`.
`type`	string	The type of the values in this field. The property is the name of the M-Script type; for example, “string” or “int.”

Example

```
{
  title:"Registration Time",
  description:"Weekly registration time",
  groupValue:true,
  type:"real"
}
```

universeGroupDef

Description

A description of the group of fields in a universe defined by a universe handler. The value is an object. This object is found as the properties of the `groups` property of the `universeDef` type.

Properties

Name	Type	Description
`title`	string	The display title of the group.

Name	Type	Description
`index`	array	An array with elements of type `string`. Each `string` is the name of a field or a group. The order of the entries is significant. Each name matches a name found in the properties groups or columns of the object of type `universeDef`.

Example

```
{
  title:"Registration Time",
  description:"Weekly registration time",
  groupValue:true,
  type:"real"
}
```

userInfo

Description

An object containing various information about the Maconomy user currently logged in. Returned by the function `getUserInfo`.

Properties

Name	Type	Description
`userName`	string	The account name of the user.

Example

```
{
  userName: "Administrator"
}
```

validateReturn

Description

The return value of functions that validate the spelling of entities such as dialogs, Analyzer reports, and pop-ups.

Properties

Name	Type	Description
`ok`	bool	This property is `true` *iff* the entity name was spelled correctly.

Name	Type	Description
`message`	string	An optional message describing why the entity name could not be validated. This property exists only when `ok` is `false`.

Example

```
{
  message:"analyzer document 'xxx' not found on server",
  ok:false
}
```

valueDeleteReturn

Description

The return value when trying to delete an M-Script value stored in the database. The value is an object.

Properties

Name	Type	Description
`ok`	bool	This property is `true` *iff* one or more values were deleted in the database. More than one property can be deleted when calling `deleteAllUserValues`.

Example

```
{
   ok : true
}
```

valueGetReturn

Description

The return value when trying to retrieve an M-Script value stored in the database. The value is an object.

Properties

Name	Type	Description
`ok`	bool	This property is `true` *iff* a value of the requested name exists in the database.
`Value`	any	The retrieved M-Script value. This value can be of any type and complexity. If the property `ok` is `false` then this property does not exist.

Example

```
{
   ok    : true,
   value : {
             a : 42,
             b : [1,2]
            }
}
```

Part 3 – M-Script GUI-II Reference

The Maconomy M-Script Language is designed for generating dynamic web pages based on data from a Maconomy system. This sections describes how graphical user interface (GUI) components as seen in the Portal are created using M-Script.

Two other M-Script sections describe other aspects of the M-Script language:

- The M-Script Language Reference describes the basic M-Script language constructs.
- The M-Script Maconomy API Reference describes how M-Script interfaces with the Maconomy application.

Prerequisites

To understand and use the M-Script GUI-II API you must have a good understanding of the M-Script programming language. Consequently, no basic M-Script language constructs are explained in this document. It is assumed that you already know the M-Script language well.

The M-Script Language Reference provides a thorough introduction to the M-Script language.

What is the M-Script GUI-II API?

The acronyms GUI and API stand for “graphical user interface” and “application programmer interface,” respectively. This means that the M-Script GUI-II API is an interface that allows an M-Script programmer to easily set up forms and other GUI components on any supported web device without any knowledge of the language used by the web device, such as HTML.

The number II indicates that this is the second revision of the GUI API. The previous version has been deprecated—the previous API calls still exist for backward compatibility, but Deltek strongly requests that programmers use only the new API.

The API consists of a collection of functions and procedures that can be called from M-Script. All of these subroutines reside in the M-Script module gui, which means that all of the subroutine names must be prefixed with “gui::.”

The following is a simple example of an M-Script that uses the GUI API.

```
#version 15 newsession();
var specification = ...; // Details left out.
var layout        = ...; // Details left out.
var dialogData    = ...; // Details left out.
var userData      = ...; // Details left out.
gui::open("Example Page",
       "script.ms",
       specification, layout, dialogData,
       userData);
```

This script creates a session and displays a GUI component on the current web device. It does not have any routines for handling input, however.

Since the name “M-Script GUI-II API” is long, the remainder of this document uses the term “GUI API.”

Target Independence

One of the main features of the GUI API is that it is independent of the web device that is used to display GUI components to end users. The web device—the *target*—is automatically detected,

and code for the detected target is generated. This code could, for instance, be slightly different dialects of HTML for Netscape's and Microsoft's web browsers.

This means that an M-Script programmer should not worry about which dialect of HTML a specific web browser supports. The GUI components to display are described in an abstract way, leaving such details to the GUI API code.

The supported targets are:

- Netscape Navigator version 6 or higher (Windows and Macintosh), although best viewed on Mozilla 1 or higher or Netscape Navigator 7 or higher.
- Microsoft Internet Explorer version 5.5 or higher (Windows only).

Reading this Section

In this section the term "iff" is used to mean "if and only if." The sentence "The return value is true iff the value exists" means that the return value is true if the value exists and false if it does not exist. The term "simple M-Script value" means any M-Script value of type bool, int, real, amount, date, time, or string.

Throughout this document examples are given to illustrate the use of the GUI API. To simplify these examples, not every possible error is checked when calling a GUI subroutine. This makes it easier to read the examples. However, you should be aware that when using the GUI API for creating real applications, extensive error checking is necessary to produce robust M-Script applications.

Also, the example scripts are not complete scripts. To reduce the examples to a reasonable size, details about creating sessions, logging in, and so on, are left out unless understanding the example requires these details to be present.

An ellipsis (. . .) is used in many places to denote that some code that is not relevant to the example has been left out, as the following example shows

```
// Function for getting user's max. rows selection. function
readUserMaxRowsSelection()
{
    ...
}
```

and to denote left out object or array complexity as the following example shows.

```
{
   a : 42;
   b : [...], // Array details left out.
   c : {...}, // Object details left out.
   ...        // More properties left out.
}
```

Finally, an ellipsis is used in subroutine prototypes to denote a subroutine that accepts an arbitrary number of arguments, as the following example shows.

```
// 'sampleFunction' accepts 2 or more arguments.
function sampleFunction(arg1, arg2, ...)
```

Object Specifications

Sometimes a complete M-Script object is specified with all of its properties and sub-objects like the following.

My Object Specification

```
1  <propertyName> : object
2    kind : string
3    type : string
```

My Object Description

1. Description of <propertyName>.
2. Description of Kind property.
3. Description of type property.

The first part gives an overview of the object, and the second part gives an in-depth description of the properties. The line numbers can be used to connect the two.

The object description contains one line for each property in the object. Each line specifies the name of the property, followed by the type. If the property name is enclosed in brackets, the name is a user-specified property name. Typically this is used for storing a set of elements with different names—for instance, input fields that are defined in an object with one property for each field.

Indentation is used to identify which object a specific property belongs to when using embedded objects.

Organization of this Section

The rest of this section is divided into the following parts:

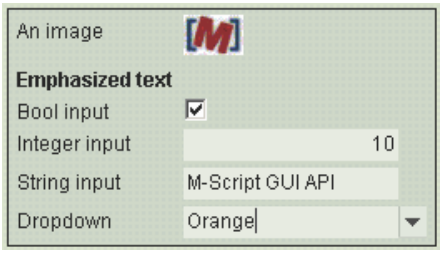
- The sections Overview through Getting Device Information provide a tutorial that guides you step-by-step through the various parts of the M-Script GUI API.
- The sections Component Specification through Composer Package Routines contain a complete listing of all of the data types, events, and subroutines in the M-Script GUI API. This reference description is also useful during the reading of the first set of sections, where some details have been omitted.

Overview

The GUI API can be used to create three different kinds of visible components: card, table, and composer components. Card components are for single records; table components show multiple records in a table, and the composer component can be used to join multiple card or table components.

Card Components

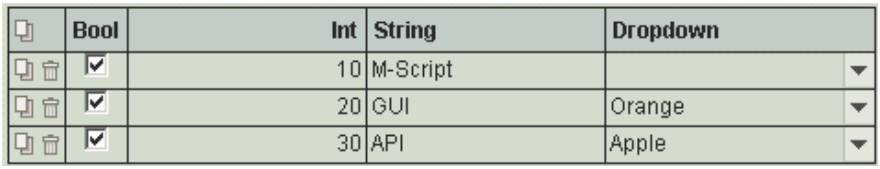
These are components that hold input fields for exactly one record—for instance an “Item Information Card” from Maconomy. A simple card component could look like this the following figure.



This example shows some static elements like various kinds of text and an image, with input fields for Boolean, integer, string, and enumerated data.

Table Components

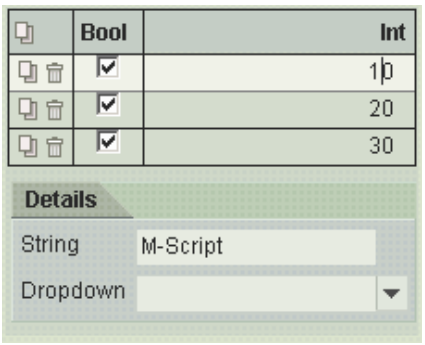
These components hold input fields for multiple records—for instance, all of the items in a shopping basket or the lower pane of a Maconomy dialog. A simple table component could look like the following figure.



This example shows a table with three rows, each containing a Boolean, integer, string, and enumerated field.

Details Islands

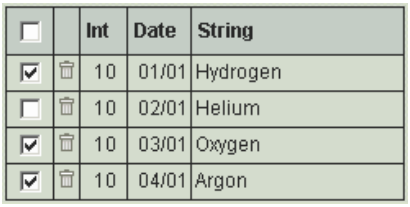
Table components can also have details islands associated with them. These island elements show extra fields for the current row and can be used for tables that have so many columns that they cannot be shown as normal table columns. Details islands are shown below the table itself, as seen in the following figure.



Row Actions

The “insertRow” and “deleteRow” actions can be moved around, split, or completely left out if needed. You can also add “selectRow” checkboxes that allow a user to mark a set of rows for some specific action.

This example shows an added a row selector, the removal of the ability to insert a row, and moving the delete row icon to the left.



List Components

List components are simply read-only tables. They can be used to improve performance in tables that only present data to users.

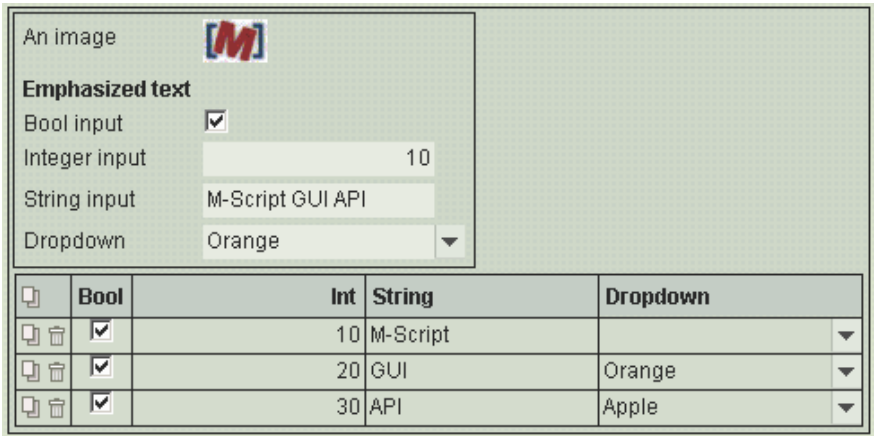
Keyboard Shortcuts

The following table describes the keyboard shortcuts that are provided.

Shift	Ctrl	Key	Description
		Up/Down	Navigation in tables. Selection of drop-down values.
	√	Up/Down	Navigate up/down from drop-down lists.
		Enter	Submit data. New text line in text box.
	√	Enter	Submit data—also in text box.
(√)	√	G	Open search window.
		Tab	Navigation in cards, and horizontal navigation in tables.
√		Tab	Backward navigation in cards, and backward horizontal navigation in tables.

Composer Components

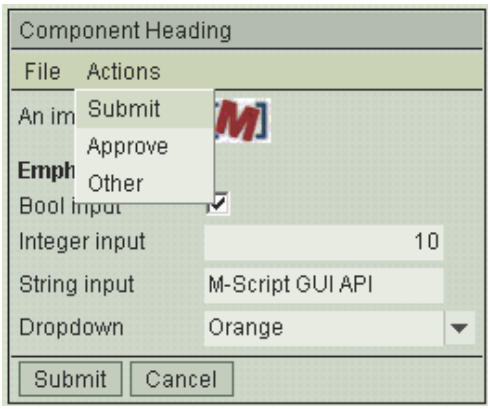
A composer can be used to recursively join multiple components to obtain card/table dialogs as used in Maconomy—or even more complex combinations with multiple table and card components. A simple card/table dialog created with a composer component could look like the following figure.



This example has simply combined the two previous components into a single card/table component.

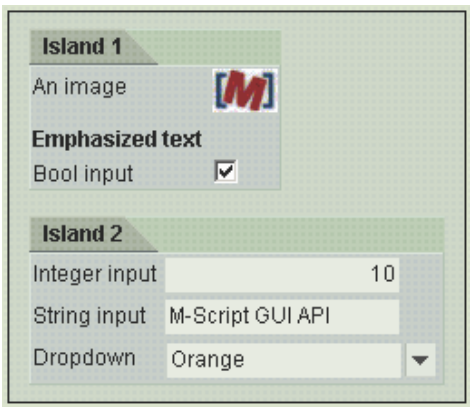
Headings, Menus, and Buttons

All components can be equipped with various headings, menus, and buttons as the following figure shows.



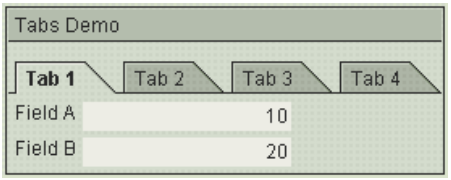
Data Islands

You can also format card layouts using islands as in Maconomy's normal client.



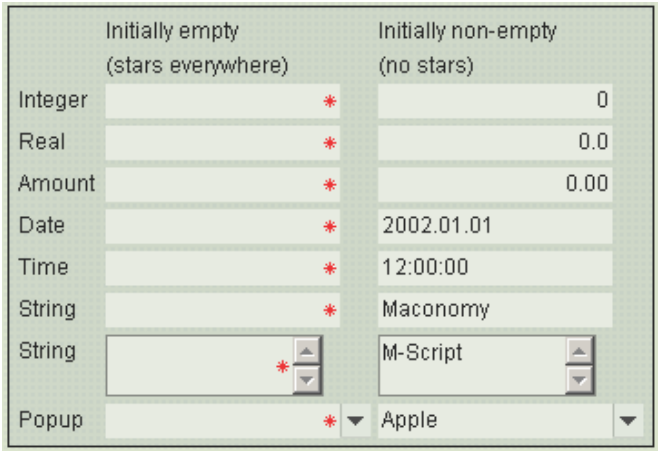
Tabs

The GUI framework supports the use of tabs to visualize different components.

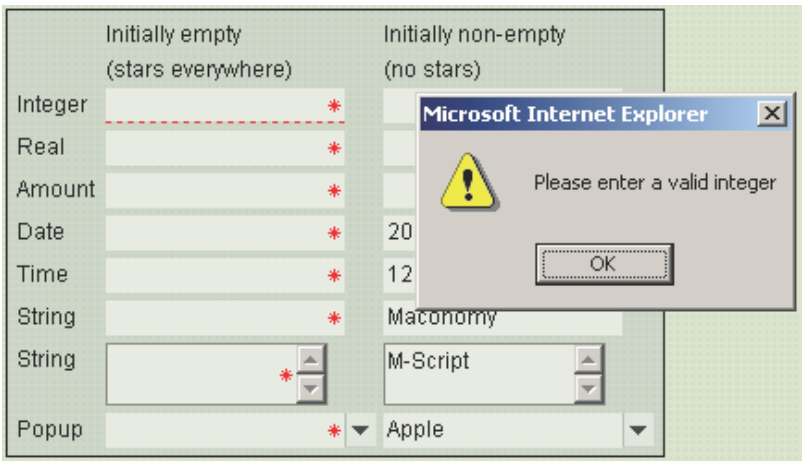


Input Validation

The GUI framework performs automatic validation of the input data, and it does not allow a user to enter any data that has not been validated. The validation scheme handles mandatory validation and type validation. Mandatory validation enables a programmer to define some input fields as mandatory, which means that data *must* be entered in those fields. Mandatory fields are marked with a red star as the following figure shows.



If a user tries to submit empty data in a mandatory field, he or she receives an alert stating the problem.

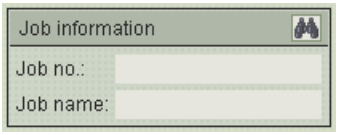


Following this, the invalid fields are marked with red underlines.

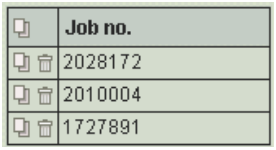
Type validation works in exactly the same way—if, for instance, a user enters something other than a date in a date field, an alert pops up stating the problem.

Search Specifications

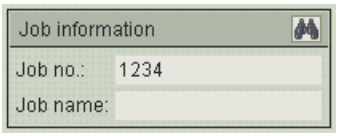
The GUI supports the specification of a search functionality that can help users to find the correct data for a field—for instance, a Maconomy job number that a user must enter. To illustrate this requires a simple job information card like the following.



Here the user needs to enter a job number. But the programmer has made it possible to search for a job, as can be seen by the highlighted search icon in the top-right corner. When the focus is in the **Job no.** field, the user can press Ctrl+G or click the icon, which displays the following window.



In this window, the user can select a row and then press Enter. The value of that row is then transferred to the original window.



Opening Separate Windows

As shown previously, you can open new child windows in various ways, for instance by defining a search. However, there are also other ways to do this.

One way is to call gui::newWindow(...) which takes a URL, opens a new browser window, and points it to the specified URL.

The second way is to add an “Open” button to the component. Such a button opens a separate window and sends a subWindowInitialized event to the event handler of that window.

When the parent window receives the subWindowInitialized event, its window data is passed to windows that are opened with searches and “Open” buttons. Both may use gui::notifyParent(...) to communicate with the parent window. The search windows may also use gui::searchTransfer(...) to transfer data back to the parent window—as if it had been entered by the user.

Programming Model

The programming model that is used in the GUI API is an event-based model. This means that after a window has been created, it communicates with the associated M-Script program via events. These events can identify user input, button activation, and much more.

The events are handled by an event handler, which must be written by an M-Script programmer.

Architecture

The GUI-II framework takes web programming one step further than normal HTML forms programming by using dynamic HTML and JavaScript to an extent where only data flows between the web browser and the web server. There is no massive data overhead involved in the transactions, and no HTML is ever transmitted during normal use. This means that you get a fast and efficient data entry interface without the familiar screen flicker while new HTML is loaded.

This is achieved with a complete window handler written in JavaScript that takes care of user input, data validation, and transmission of data back and forth between the browser and the server.

The exact implementation is illustrated the following figure. It consists of two frames—a visible frame, where the actual HTML is shown, and a hidden frame that is used for communication.

The creation of a GUI-II window is a two-step procedure that works as the following figure illustrates.

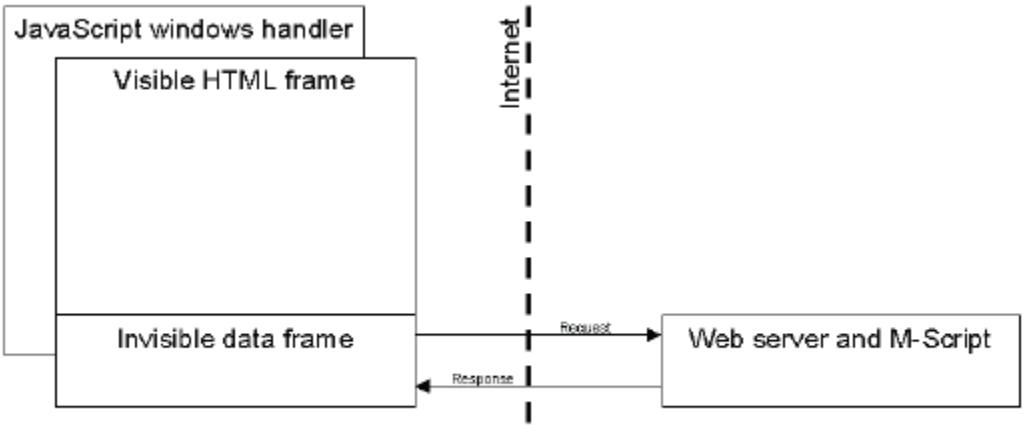


Illustration of GUI-II architecture

When a call to gui::open(...) is executed, it results in an HTML page that contains a link to the window handler JavaScript code, all of the static HTML, and a hidden frame as shown in the preceding figure.

An onLoad JavaScript handler then changes the location of the hidden data frame to a URL that requests the initial data from M-Script. This is then returned, formatted as JavaScript data, and the window handler interprets the data to fill in the missing card data, as well as table rows, using dynamic HTML.

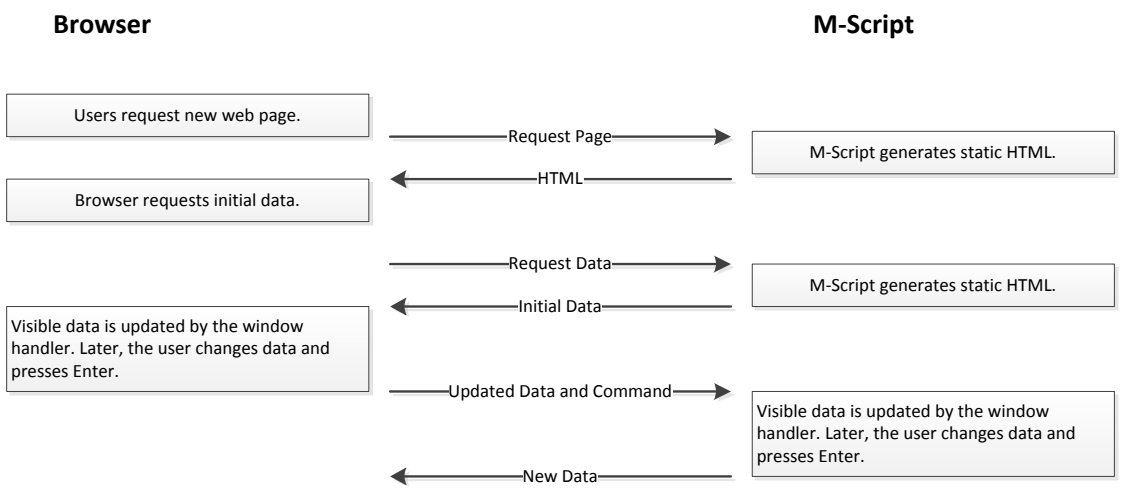
After these steps, the system is ready to handle user input and communicate with the web server. In this mode the system tracks a user's input of data and navigation through the various components and records, and whenever a user presses Enter or changes focus from one record to another, the window handler submits the new data to the M-Script interpreter.

When new data is submitted to the M-Script interpreter it first validates the input according to the rules defined by the programmer. If the input is valid, M-Script updates its internal tables and notifies the related M-Script program of the changes.

The M-Script program may then, based on the new input, choose to update fields, insert or remove table rows, change access permissions, and much more. The result is a set of commands encoded in JavaScript that is returned to the web browser.

At last the window handler that is embedded in the web browser interprets the commands returned from M-Script, updates the screen accordingly using dynamic HTML, and then waits for new user input.

The entire data flow is shown in the following figure.



Installation

In addition to the M-Script interpreter, the GUI framework needs its associated JavaScript window handler, a style sheet file, and a few images that are used for the various icons. The name of each of these files can be specified, with a few other GUI-II settings, in M-Script's initialization file as shown in the following example.

```
gui_windowHandlerURL      = /GUI/dialogHandlerCode.js
gui_styleSheetURL         = /GUI/gui2.css
gui_searchImageURL        = /GUI/search.png
gui_mandatoryDropDownText = *
gui_mandatoryImageURL     = /GUI/mandatory.png
gui_deleteImageURL        = /GUI/delete.png
gui_insertImageURL        = /GUI/new.png
gui_menuArrowImageURL     = /GUI/menuArrow.png
gui_deleteImageText       = Delete line
gui_insertImageText       = Insert line
gui_intErrorText          = Please enter a valid integer
gui_realErrorText         = Please enter a valid real
gui_amountErrorText       = Please enter a valid amount
gui_dateErrorText         = Please enter a valid date
gui_timeErrorText         = Please enter a valid time
gui_stringErrorText       = Please enter some text
gui_inputLengthErrorText  = Input field cannot hold ^(actualLength)
   characters. Maximum input length is
   ^(maxLength).
gui_dropDownErrorText     = Please select a value
gui_dictionaryPrefix      = T
gui_specialVars           = cid;layouttreeid;callingcid
gui_searchWindowPackage   = custom::searchWindow(2)
gui_forcedCloseFrameName  = forcedClose
gui_maxWindows            = 5
```

All of the image names refer to the image basename only. A state name that describes the state of the image is concatenated to the image basename. This can be "_pushed" or "_active," where "_pushed" is added when a user clicks the image, and "_active" is added when a user hovers the mouse over the image.

Most of the settings are required to use the GUI functions. An exception of type error::stdlib::gui is thrown if gui::open() is called when some of the required settings are missing from the initialization file.

Using Themes

You can also specify a single theme for the look-and-feel of the GUI. This works by specifying a single URL where all of the GUI files can be found. The GUI framework then assumes that all of the files have some fixed names that are concatenated to the theme URL.

The theme URL setting is themeURL. The use of this setting prohibits the use of all of the other settings that are derived from the theme (see "Theme Defaults" for a list).

List of GUI Settings

The settings are shown in the following table. Required settings are flagged by a check mark.

Setting	Description	Req.
gui_windowHandlerURL	URL that points to the window handler JavaScript code.	√
gui_styleSheetURL	URL that points to the GUI-II style sheet.	√
gui_searchImageURL	URL that points to the search icon image. Possible states are “_pushed” and “_active.”	√
gui_mandatoryDropDownText	The text that is used to indicate an empty mandatory drop-down value (typically a ‘*’).	√
gui_mandatoryImageURL	URL that points to the mandatory icon image. No states added.	√
gui_deleteImageURL	URL that points to the delete-row icon image. Possible state is “_pushed.”	√
gui_insertImageURL	URL that points to insert-row icon image. Possible state is “_pushed.”	√
gui_menuArrowImageURL	URL that points to menu arrow icon image. No states added.	√
gui_selectImageURL	URL that points to arrow icon image used for popups and combo boxes. No states added.	
gui_deleteImageText	Tool tip text for delete-row image.	√
gui_insertImageText	Tool tip text for insert-row image.	√
gui_intErrorText	Error message for invalid or missing integer values.	√
gui_realErrorText	Error message for invalid or missing real values.	√
gui_amountErrorText	Error message for invalid or missing amount values.	√
gui_dateErrorText	Error message for invalid or missing date values.	√
gui_timeErrorText	Error message for invalid or missing time values.	√
gui_stringErrorText	Error message for invalid or missing string values.	√

Setting	Description	Req.
gui_inputLengthErrorText	Error message for strings that are too long (the size exceeds the inputLength setting). The error message is formatted using parameters in the same style as the print function. The predefined format parameters are named actualLength and maxLength. actualLength is the current number of characters the user has entered. maxLength is the size set by the inputLength specification.	√
gui_dropDownErrorText	Error message for missing drop-down values.	√
gui_loadingWindowText	When opening a search window from the Portal, the <text> in this parameter is shown as a “Please wait” message. The text is localized like other text strings in Maconomy. The default text is “Loading window, please wait...”	√
gui_dictionaryPrefix	A single letter that designates which dictionary to use for translating all of the error messages specified previously. Entering a “T” for this setting, for instance, causes all of the error messages to be translated using the dictionary that is normally used for translating M-Script strings prefixed with the `#T` modifier. See also “String Localization” in the M-Script Language Reference.	
gui_specialvars	A semicolon-separated list of query variable names that are considered “special” by the GUI. If any of these variables is set to something when the GUI opens, the original value is passed on unchanged to all following script executions. Example: `Gui_specialvars = cid;treeid;other`	
gui_searchWindowPackage	The name of the package that is used for handling dialog search.	

Setting	Description	Req.
gui_forcedCloseFrameName	The name of the frame for handling forced closing of GUI windows. The setting should be set to the name of an existing frame whose contents may be overwritten by M-Script. If a GUI window is forcibly closed, using this setting ensures that the window's handleClose() event handler is called. (A window is forcibly closed if the browser's Close or Refresh button is clicked, the browser's URL is changed, or if the contents of the frame that displays the GUI window is replaced in any other way.)	
gui_maxWindows	The maximum number of windows that may exist at a given child level. If, for example, the number is set to 5, a maximum of 5 GUI windows can exist together with 5 search windows for each of these windows. If the limit is exceeded, M-Script deletes the window data for the oldest window from the session without calling the handleClose() event handler. The default value is 5, but this setting is ignored if gui_forcedCloseFrameName is set.	
gui_optimizations	1 or 0. Controls whether a number of performance optimizations have been enabled in GUI layer. Can be turned off while debugging.	
gui_inputFieldSizeFactor_Int gui_inputFieldSizeFactor_Amount gui_inputFieldSizeFactor_Real gui_inputFieldSizeFactor_Date gui_inputFieldSizeFactor_Time	These settings are used for adjusting the width of various GUI elements. The width of input fields of the specified type is multiplied by the specified factor. The default value for the input field size factor for Amount, Real, and Date types is "1.1," whereas the others have a default factor of "1.0."	

Theme Defaults

The following list shows the default values for those GUI settings that are derived from the theme URL. The value is simply concatenated to the theme URL.

Setting	Theme Default Value
gui_styleSheetURL	CSS/gui2.css
gui_searchImageURL	Pics/search.png

Setting	Theme Default Value
gui_mandatoryImageURL	Pics/mandatory.png
gui_deleteImageURL	Pics/delete.png
gui_insertImageURL	Pics/new.png
gui_menuArrowImageURL	Pics/menuArrow.png
gui_selectImageURL	Pics/selectArrow.png

Creating GUI Components

GUI components are created from three different specifications that are represented by different M-Script objects.

The first specification is the component specification, which defines the component kind (card, table, or composer), the name of the data source, buttons, and fields. This is the core specification of the component.

The next specification is the layout specification, which defines where the various component elements are placed in the window and adds static information to the component (such as hard-coded text and images). The layout is not required to include all of the elements that are specified in the component, unless they are specified as required.

The last specification is the initial component data, which contains all of the records that should be shown initially, as well as the initial access permissions.

These three specifications are merged by the GUI procedure gui::open(...), and from this information the GUI produces the necessary output.

The Component Specification

The component specification is an object that defines the core elements of a component.

The following is an example of an object that could be passed as the componentSpec parameter to gui::open(...).

```
{
    myCard:
    {
        kind:    "card",
        data:    "myCardData",
        buttons: { ... },
        fields:  { ... }
    }
}
```

The first property defines the component kind, the second is a reference to the data source, and the button and field specifications follow.

The full specification of the component specification can be found in “Component Specification.”

The Layout Specification

The layout specification is an object that describes which components to show and where to place them.

The following is an example of an object that could be passed as the layout parameter to gui::open(...).

```
var layout =
{
    name :          "myCard",
    header:        [...],
    menu: [...],
```

```
    buttonsTop: [...],
    pane: [...],
    buttonsBottom: [...]
}
```

The first property name refers to a component specification from which the field specifications are retrieved. The layout of the main component areas follows—the header, the menu, the top buttons, the central area called the pane, and the bottom buttons.

The Data Specification

The data specification is an object that defines the initial data that is used by the components. It contains one property for each component. Each property must be named according to the data name in the component specifications.

Thus a data object for the previous examples could look like the following example.

```
{
    myCardData :
    {
        rows:
        [{
            field_b : true,
            field_i : 10,
            field_s : "M-Script GUI API",
            field_p : 0
        }]
    }
}
```

Each data property must contain a rows array that holds the actual data. If the component is a card component, the array only holds one data record. In the case of a table, it contains any number of records (including none).

Reading GUI Data

The core GUI function for retrieving information from a GUI window is gui::get(). Any program that wants to use the GUI framework must call this function at some point to get some information about the GUI state and incoming events.

The return value from this function is either null, if the program is executed before the GUI is initialized, or an object such as the following, if the program is started by the GUI.

```
{
    action:{ kind:"enter" },
    focus:{
        component: "myCard",
        field:     "field_i"
    },
    data:{
        myCardData:{
            rows: [...],
            buttons: {...},
            fields:  {...},
```

```
            enabled: true
        }
    },
}
```

The data consists of two parts—information about the current data in the GUI (myCardData) and information about the current user action and why the program was invoked.

What the preceding example shows is that a user pressed Enter, because the action kind is “enter.” The example also shows that the user did so with the cursor placed in the **field_i** field of the myCard component as defined by the focus property. After that the data in the component follows. This consists of the actual record (the rows property), the button permissions, field permissions, and enabling information of the whole component.

Based on this information, the programmer can now choose to update some of the fields (a calculator could, for instance, find the sum of two fields and update a third with that value) or change the access permission by closing some of the fields, and so on.

For detailed information about the return value from gui::get(), see gui::get in Subroutines.

Programming with the GUI

Now that the examples have shown how GUI windows are created and how GUI actions are presented to the M-Script program, you can start building a simple M-Script program that uses the GUI. For this purpose this example uses a simple calculator. With this calculator you can add and subtract values.

First of all you need a component specification like the following:

```
var specification =
{
   myCard :
   {
      kind : "card",
      data : "myCardData",
      buttons:
      {
         add: { kind: "submit" },
         sub: { kind: "submit" }
      },
      fields :
      {
         a  : { type: "real" },
         b  : { type: "real" },
         r  : { type: "real" }
      }
   }
};
```

This specification defines a card component with two buttons and three input fields. The buttons are standard “submit” buttons, which means that the current record is submitted to M-Script when a user clicks the button. In this case there is only one component. Had there been more components, each of them would be defined in separate properties of the specification.

You also need a layout like the following example:

```
var layout =
{
   name : "myCard",
   header: [ [ "A simple calculator" ] ],
   buttonsTop:
   [
      { name: "add", title: "Add" },
      { name: "sub", title: "Subtract" }
   ],
   pane :
   [
      [ "First operand",  { kind : "input", name : "a" } ],
      [ "Second operand", { kind : "input", name : "b" } ], [
      "Result",         { kind : "input", name : "r" } ]
```

```
    ]
};
```

The layout refers to the previously defined component specification named myCard—the name must correspond to one of the properties (components) of the component specification. It then adds some text to the header and places the two buttons at the top. At last it defines some static text and places the inputs in the component.

Both the header and the pane are interpreted as two-dimensional tables when the component is rendered, where each inner array represents one row of the table. Thus in the previous case there is a header with only one cell and a pane with three rows, each of which has two cells—a label and an input field.

Next you need some initial data like the following example:

```
var data =
{
    myCardData:
    {
        rows:
        [
            {
                a: 0.0, // Here we use the short form (no value property)
                b: 0.0,
                r: 0.0
            }
        ],

        fields:
        {
            r: { disabled: true }
        }
    }
};
```

This defines the myCardData used by the component. It has one record, and it also defines the **r** field as disabled, which means that a user cannot enter data into it.

It may seem a bit strange to place access permissions in the data object instead of the layout, but the reason is that the programmer can change the permissions later on, which makes it dynamic and thereby part of the data. Later versions of the GUI API might allow initial permissions to be set in the layout also.

Now you have all of the necessary information for displaying the component. You do this with a call to gui::open(...), as follows:

```
gui::open("Calculator",
        "calculator.ms",
        specification,
        layout,
        data,
        {} );
```

The first two parameters are the name of the window and the name of the script that contains the event handler (this can be the same as the script that defines the window).

The next step is to create the event handler. This uses gui::get() to get the next GUI event and react on it.

```
var g;
if ((g=gui::get()) != null)  // Get GUI information
{
   var data = g.data.myCardData.rows[0];  // Extract field data
   if (g.action.kind == "button")  // Check buttons
   {
          // Calculate result
      if (g.action.button == "add")
         data.r.value = data.a.value + data.b.value;
      else
      if (g.action.button == "sub")
         data.r.value = data.a.value - data.b.value;

      gui::update("myCard", data);  // Do the actual GUI update
   }
}
else
   ...
```

If the event handler is contained in the same script as the window specification, you must ensure that the gui::open(...) call is not executed while handling events. If it were called then, you would open the window again, which was not the intention. For this reason, you separate the initialization code from the event handling code with an “if”statement.

The event handler checks for buttons clicks, calculates new data, and updates the display with the GUI procedure gui::update(...).

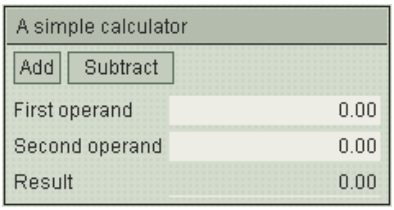
That is about all that is needed for the calculator. The complete code follows.

```
#version 15


if (!hassession())
   newsession();


// Event handler
var g;
if ((g=gui::get()) != null)  // Get GUI information
{
   var data = g.data.myCardData.rows[0];  // Extract field data

      // Calculate new values
   if (g.action.kind == "button")  // Check buttons
   {
      if (g.action.button == "add")
         data.r.value = data.a.value + data.b.value;
      else
      if (g.action.button == "sub")
         data.r.value = data.a.value - data.b.value;
```

```
        gui::update("myCard", data);  // Do the actual GUI update
    }
}
else
{
        // Component specification
    var specification =
    {
        myCard :
        {

            kind : "card",
            data : "myCardData",
            buttons:
            {
                add: { kind: "submit" },
                sub: { kind: "submit" }
            },
            fields :
            {
                a  : { type: "real" },
                b  : { type: "real" },
                r  : { type: "real" }
            }
        }
    };

        // Layout specification
    var layout =
    {
        name: "myCard",
        header: [ [ "A simple calculator" ] ],
        buttonsTop:
        [
            { name: "add", title: "Add" },
            { name: "sub", title: "Subtract" }
        ], pane:
        [
            [ "First operand",  { kind : "input", name : "a" } ],
            [ "Second operand", { kind : "input", name : "b" } ],
            [ "Result",         { kind : "input", name : "r" } ]
        ]
    };

        // Initial data and field access
    var data =
```

```
    {
        myCardData :
        {
            rows:
            [
                {
                    a: 0.0,
                    b: 0.0,
                    r: 0.0
                }
            ],

            fields:
            {
                r: { disabled: true }
            }
        }
    };

        // Start the GUI

    gui::open("Calculator",
            "calculator.ms",
            specification, layout, data, {} );
}
```

The calculator should look like the following figure.



The Event Handler

The most complex part of any GUI program is the event handler. This handler is executed every time that a user does something that the program should be informed of. This could, for example, be key presses, adding or removing table rows, or changing focus from one record to another.

The event handler calls gui::get() to get the event information and then checks which kind of event it has received. The event information is stored in the “action” property of the data that is returned from gui::get(), and depending on the action (event) kind, this object also contains some more information that is specific to the current action.

A typical event handler would look something like the following example.

```
var g;
if ((g=gui::get()) != null)
{
    if (g.focus.dataChanged)
```

```
{
   ... if necessary then update internal state based on the current focus
   information and field data.
   (for instance using the Maconomy API) ...
}

switch (g.action.kind)
{
case (event is a button)
   ... check which button it is and react on it ...
break;

case (event is focus change)
   ... use gui::setFocus() to change focus
      if a focus change is allowed ...
break;

   ... and so on ...
}

if (current data is valid)

   gui::acceptData(); // Call this in order to allow the
   // user to change component focus.
}
```

Multiple Components

When multiple components are put together with a composer component, there are a few things that should be taken extra care of.

Acknowledging Input

First of all, an M-Script programmer needs to acknowledge all operations before a user is allowed to change focus from one component to another. Consider as an example a Time Sheet component where some input combinations are illegal (Job and Activity, for instance). In this case, the data might seem valid from the GUI's point of view, but invalid when used in the business application.

The normal operation of the GUI is to allow a user to change focus from one record to another (two different table rows or different components) whenever the data is valid. But this focus change should be disallowed when the data is invalid with respect to the business application. Thus for this reason the M-Script programmer must tell the GUI window handler that the data that was entered is valid, before the window handler allows focus changes.

The same problem applies if a row in a table is created, and the user should be forced to fill in some of the fields before changing focus to another row.

The function used to acknowledge user input is called gui::acceptData().

Focus Information

It is also important to pay attention to the focus information. It is, for instance, possible to click a button in one component, while the focus is in another; it is also possible to request the deletion of a row that is different from the current row.

Composer Packages

Although the framework as described so far works very well and is sufficient for many applications, it does have two major drawbacks. It is impossible to add extra functionality to existing applications without having direct access to the source, and the code tends to be rather monolithic, with all of the functionality centered around the gui::get() function.

To overcome these problems, a system has been created that allows a programmer to add components to an existing layout, without the original application knowing anything about them. These are called components for composer packages, because the components are defined in M-Script packages and must be included in a composer component that exists in the original application. Thus you must modify some of the existing code, but only in the layout that is typically made available to end users already.

Composer Package Interface

A composer package consists of a number of functions that the GUI framework calls to either get information about the components, or to dispatch events to the components. Some of the functions are mandatory, while others are optional. The composer package can supply these functions in one of two ways:

1. If a public function named getComposerInterface() is defined in the package, this package must return an object with the functions as its members, for example:

```
Public function getComposerInterface()
{
    return
        {
             ... // Add all functions for this composer package here
        }
}
```

2. If no public function named getComposerInterface() is defined in the package, the package itself is searched for public functions.

These two design methods are mutually excluding, that is, if getComposerInterface() is defined, all functions must be in the returned object, because the package is not searched for other public functions.

Specification and Setup

A composer package is included in a GUI layout by specifying the package path in the handler property for a layout specification. You can also specify an initializer value here, which is later passed to the composer package.

Consider the following simple example:

```
   // Define a composer in which we can put our packages
var specification =

{
```

```
    myComposer:
    {
        kind: "composer"
    }
};

    // Put the packages in the layout
    // (the layout object is assumed to be editable by the user)
var layout =
{
    name: "myComposer",
    header: [[ "Composer package demo" ]],
    pane:
    [
            // Add to rows to the pane – one with a card component and
            // one with a table element
        [ { kind: "component", handler: ".::myCard(1)", init: ... } ],
        [ { kind: "component", handler: ".::myTable(1)" } ]
    ]
};

    // Define standard empty data object for composer
var data =
{
    myComposer: {}
};

    // At last we open our composer
gui::open("Composer package demo", "demo.ms",
            specification, layout, data, {});
```

The only new feature in the preceding example is the properties handler and init in the layout. This is the only thing that must be modified in the original component (in this case, a rather dull component with one composer and no functionality) to add a composer package to it.

For a package to be an actual composer package, it must define a certain set of functions. The GUI framework calls these functions at certain times to retrieve the component layout, specification, and data, and to allow the component to react on the various events.

When the GUI reads the preceding layout, it detects the new handlers, loads the specified packages, obtains the composer package interface, and then the composer package specification. This function is expected to return a composer package specification as described in Composer Package Specification.

This information is then merged into the existing specifications by the GUI framework, and the result is a complete specification of all of the included components.

Composer Package Specification

The composer package specification object that is returned by getSpecification() describes the components, layout, and data that are supplied by a package. This specification is an object with the following properties.

Name	Type	Description
component	object	A component specification, as described in “Component Specification.”
layout	object	A layout specification, as described in “Layout Specification.”
data	object	Initial data, as described in “Data Specification.”
composerData	value	Data that is private to this composer package. This value is included in the event data for this package. (See “Event Description.” If no composer-specific data exists, you can omit this property.

The following example continues the preceding example code. First it shows the composer package specification for the card component:

```
package myCard(1);


      // This package defines a card with two fields – "A" and "B"
public function getSpecification(userData)
{
   var myComponent =
   {
      myCard:
      {
         kind: "card",
         fields:
         {
            a: "int",
            b: "string"
         }
      }
   };

   var myLayout =
   {
      name: "myCard",
      borders: false,
      pane:
      [
         [ "Field A:", { name:"a" } ],
         [ "Field B:", { name:"b" } ]
      ]
```

```
    };

    var myData =
    {
        myCard:
        {
            rows:
            [
                { a:10, b:"Copenhagen" }
            ]
        }
    };

    var myComposerData = { ... };

    return {
        component: myComponent,
        layout: myLayout,
        data: myData,
        composerData: myComposerData
    };
}
```

The alternative composer design using getComposerInterface() would look like the following.

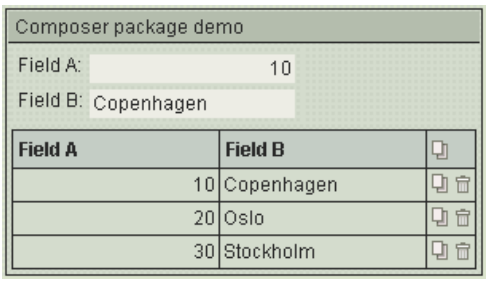
```
...
public function getComposerInterface()
{
    return
    {
        getSpecification: function(userData)
        {
            // See implementation of getSpecification()
            // in the previous example
        },
        ... // Other functions for this composer package goes here
    };
}
```

The table package, which looks much like the card package, is:

```
package myTable(1);


    // Define a table component with two fields - "A" and "B"
public function getSpecification(userData)
{
    var myComponent =
    {
        myTable:
```

```
        {
            kind: "table",
            fields:
            {
                a: "int",
                b: "string"
            }
        }
    };

    var myLayout =
    {
        name: "myTable",
        borders: false,
        columns:

         [
            { name: "a", heading: "Field A" },
            { name: "b", heading: "Field B" }
        ]
    };

    var myData =
    {
        myTable:
        {
            rows:
            [
                { a:10, b:"Copenhagen" },
                { a:20, b:"Oslo" },
                { a:30, b:"Stockholm" }
            ]
        }
    };

    return { component: myComponent, layout: myLayout, data: myData };
}
```

These composer package specifications will result in output that looks roughly like the following figure.



Top-Level Composer Packages

You can specify a composer package directly in the gui::open call and thus avoid the title, spec, layout, and data parameters to gui::open completely (see “gui::get”). These must instead, be supplied by the composer package.

A top-level composer package must supply the getTitle() function in its interface, which returns the title of the window. The getTitle() function itself is supplied the user value from gui::open() as a parameter, for example:

```
// In main script
#version 15
gui::open("composer(1)", null);



// In package "composer(1)" - file "composer.1.ms"
#version 15
public function getTitle(userValue)
{

    return "My title";
}
```

Event Handling

Events to components defined by a composer package are dispatched directly to the event handlers in the package, in addition to returning the event data from gui::get(). The event handlers in a composer package are all named handleXXX where xxx is the name of the event. Thus, for instance, the handler for the initialized event must be named handleInitialized().

Event handlers must be placed in the composer package interface that is returned by getComposerInterface(), or—if this function is not defined—as public functions in the composer package itself.

The following is an example of a public event handler function for the enter event in the card package.

```
public function handleEnter(e)
{
       // Extract data for easier access
    var data = e.data.myCard.rows[0];

       // Modify the data
    data.a.value *= 2;
```

```
    data.b.value = "[" + data.b.value + "]";

       // Update GUI with new values
    gui::update("myCard", data);

       // Everything is okay, so make sure the user may change focus
    gui::acceptData();

       // Return true to indicate that –
       // yes – we did change something
    return true;
}
```

The following are worth noting:

- The GUI framework ensures that only the component in which “Enter” is pressed gets the event. Thus, because the preceding code is placed in the card package, the programmer knows that the user must have pressed Enter in the card component.
- All event handlers are passed a parameter that contains exactly the same properties as would have been returned from a gui::get() call. This includes read-only access to the other component’s data.
- The event handler can call any gui:: function and update any component that it wants to update.
- Many event handlers are functions that must return true or false, depending on whether or not they modified anything in the GUI. This is used by the internal event dispatcher to decide which event handlers to call afterwards (see “Package Dependencies”).
- The event handlers are called by the GUI framework when the main program calls gui::get(). There is no check to avoid multiple calls to gui::get() and thereby multiple calls to the event handlers.
- If relative package names are used to specify a composer package, the package names must be relative to the main script.

Package Dependencies

With the composer package features described so far, you can add components to an existing application. They may read data from other components, and they may update other components, but you cannot handle the situation where a composer package must be called when something outside the package is modified. This could, for instance, be the situation where you want to add a summary component to an existing table component—in this case, the summary component would not be called unless a user placed the cursor inside it or pressed a button inside it, or took similar specific action that would execute the component.

For this reason you can specify composer packages’ dependencies on other components, such that if a package P depends on a component C, it is guaranteed that the package P is executed when something in component C is updated.

Dependencies can be many things; a component may be required to get new data from the Maconomy dialog API when a certain other component is updated, or the dependent component may simply read the GUI data directly from the other component’s data. In the first case, the components are loosely coupled, meaning that they work completely independently of each other. In the second case, the components are said to be strongly coupled, meaning that at least one of them cannot work without the other, because it reads the other’s GUI data directly.

Loosely coupled components can be joined completely independently of each other in a layout that is specified by a higher-level component. This gives more flexibility when composing separate components. In this case, the higher-level component should use gui::addDependency() (see “gui::addDependency”) to add dependencies between components.

Example

```
var layout =
{
   ... contains composer packages "myCard(1)" and "myTable(1)" ...
};

   // Make sure myCard is notified when myTable is updated
gui::addDependency("myCard(1)", "myTable(1)");

gui::open(...);
```

Strongly coupled components may choose to let the packages themselves specify the dependencies (this can also be done in loosely coupled components, but it is not recommended). This is done by adding an optional function named getDependency() in the package interface. This function is passed the user data from gui::open() as the first parameter and must return an array of strings that identify the names of the components that the package depends on.

To continue the previous example, you can specify that the card component depends on the table component by adding the following function to the card package, instead of using gui::addDependency():

```
public function getDependency(userData)
{
   return [ "myTable" ];
}
```

The package that depends on other components must implement an event handler named handleDependencyUpdate().

In more complex applications, where there are many packages and components that depend on each other, the GUI framework calculates an optimal event sequence that guarantees that events like initialized are sent to the handlers in an optimal sequence. This sequence ensures that if package A depends on a component in package B, package B receives the initialized event before package A.

Dependency Cycles

Applications where two (or more) components are mutually dependent should not be implemented in different packages. Mutually dependent components must be placed in the same composer package.

The GUI framework does handle cycles in the dependencies, but it finds some fixed finite order to call the handlers in, and does not go into an infinite loop where two packages receive dependencyUpdated events infinitely.

Getting Device Information

Calling the function gui::getDeviceInfo() returns information about the target for which the GUI is currently generating code. The return value is an object that could look like the following example.

```
{
   browser :
   {
      name : "msie",
      version : [6,0]
   },
   os :
   {
      name : "windows",
      type : "2000"
   },
   target :
   {
      name : "html"
   }
}
```

In this case, the detected target is the browser “Microsoft Internet Explorer” version 6.0 for Windows. The target language is HTML.

The properties of the return object are described in the following table.

Name	Type	Description
browser	object	This object property contains two properties: ▪ The property name is a string that describes which browser is being used. Possible values are “msie” (Microsoft Internet Explorer), “netscape” (Netscape Navigator), “opera” (Opera), and the empty string if the browser is unknown. ▪ The property version is an array that describes the browser version. The elements are the various components of the version; for example version 6.0 becomes the array [6,0]. This property is the empty array if the version is unknown.
os	object	This object property contains two properties: ▪ The property name is a string that describes which operating system the browser is running on. Possible values are “windows” (Microsoft Windows), “mac” (MacOS), “sun” (Sun Solaris or SunOS), “linux” (Linux), and the empty string if the operating system is unknown. ▪ The other property is type, which describes the subtype of the operating system. Possible values are “XP” (Windows XP), “2000” (Windows 2000), “NT” (Windows NT), “ME” (Windows ME), “98” (Windows 98), “95” (Windows 95), “X” (Mac OS X), and “classic” (other Mac). There are no subtypes for Unix browsers.

Name	Type	Description
target	object	This object property has one property, name, which is a string that describes which target language is being used. Possible values are “html” for HTML or the empty string if the target language could not be detected. In the latter case the GUI is not able to generate code, and calls to gui::open()fail.

Component Specification

This chapter provides details about the specification of a component. Such a specification consists of a component kind, button specifications, field specifications, and search specifications, each of which is described in the following sections.

As stated previously, a GUI specification may contain multiple component specifications. For this reason, the top-level GUI specification is an object that has one property for each component in the window.

The properties of a component specification are identical for card, table, and composer components.

The properties of a component specification are described in the following table.

Name	Type	Description
kind	string	Defines the component kind: “card,” “table,” “list,” or “composer.”
data	string	Defines the name of the data source for the component. The name is used to find the correct data property of the initial data object. This property is optional, with the component name as the default value.
buttons	object	Defines the available buttons for this component. Optional. See “Button Specifications” for more information.
fields	object	Defines the available input fields. Optional. See “Field Specifications” for more information. Not available for composer components.
searches	object	Defines the available searches. Optional. See “Search Specifications” for more information. Not available for composer components.
rowActions	object	Defines the set of available row actions (insert, delete, and select row). See “Row Actions” for more information.

Button Specifications

All buttons are defined in a single button specification object, which contains one property for each of the buttons. The names of these properties are the names of the buttons and should be used in the layout specification when referring to a button.

Each button specification is an object that has the following properties.

Name	Type	Description
kind	string	Defines the button kind: “submit,” “undo,” “reset,” “cancel,” “search,” or “open.”
required	bool	Normally buttons are optional, but if this property is set to true, the button is required and must be included in the layout. The required and optional properties should not be used together.
toolTip	string	A short descriptive help message that pops up when a user lets the mouse hover above the button. Can be overridden in the layout.
optional	bool	Normally buttons are optional, but if this property is set to false, the button is required and must be included in the layout. The required and optional properties should not be used together.
script	string	Defines the name of an M-Script program that should be executed when an “open” button is clicked. This property is only valid for “open” buttons.
handler	package	Specifies a package (a loaded package that is returned from the function loadpackage) that should be used as the event handler package when an “open” button is clicked. This property is only valid for “open” buttons.
parameters	object	Specifies a set of parameter values to be passed on to the search handler package. This parameters object is passed as the parameters property on the action object of the event object that is passed to the event handler handleSubWindowInitialized in the package. This property is only valid for “open” buttons that have a search handler specified (not for scripts).
xsize	int	The width (in pixels) of the new window opened by an “open” button. This property is only valid for “open” buttons.
ysize	int	The height (in pixels) of the new window opened by an “open” button. This property is only valid for “open” buttons.

The shorthand form of a button specification is a text string that represents the kind field.

The following is an example of a set of button specifications that covers all button kinds.

```
buttons:
{
   help:         "submit",               // shorthand
   instructions: { kind:     "submit" },
   update:       { kind:     "submit",
                   required: true },
   undo:         { kind:     "undo" },
```

```
    reset:          { kind:     "reset" },
    cancel:         { kind:     "cancel", toolTip: "Cancel operation" },
    find:           { kind:     "open",
                      script:   "find.ms",
                      xsize:    200,
                      ysize:    100 }
}
```

Submit Buttons

“Submit” buttons are the “standard” buttons in the GUI framework. When a submit button is clicked, the GUI sends a button event to the M-Script event handler with the current data record (the focus.dataChanged property indicates whether or not the data has been changed since the last event).

Undo Buttons

“Undo” buttons do nothing but undo the typing done by the user. No events are sent to the event handler (it is a client-side operation only).

Reset Buttons

“Reset” buttons do the same as undo buttons, but they also notify the event handler with a normal button event. The input is validated before calling the event handler.

Cancel Buttons

“Cancel” buttons do not undo any input. A click on a cancel button results in a normal button event being sent to the event handler. No input validation is done. This means that the input data that is available for the event handler does not show the same as the GUI displays. This is very important to know, because a click on a cancel button forces the displayed data to be out of sync with the data that is available for M-Script.

You should always do a complete data update from M-Script, or open a new dialog, or something else to force the data into sync again after receiving a “cancel” button event.

Search Buttons

“Search” buttons have the same functionality as the search icon (the binoculars) in the top-right corner of a GUI component. These buttons can be used to place a search button somewhere closer to where it is needed in the layout. However, it is not rendered as a pair of binoculars, but in the same way as all of the other buttons.

Open Buttons

“Open” buttons use JavaScript to open a browser window with the specified x and y sizes. Following this, a subWindowInitialized event is sent to the event handler in the script or package that is specified for the button. See “fieldChanged Event” for more information.

The new window may then use gui::open(...), if a script is specified, to create a GUI window, or simply output whatever data it wants. If a package is specified, this package must implement a handleSubWindowInitialized procedure for the subWindowInitialized event.

Since the new window is a child window of the original, it has some restricted access to its parent. First of all the child window has its parent’s window data available in the object that is returned from gui::get(...) when receiving the subWindowInitialized event. In this situation, the event data looks as follows:

```
{
   action:{ kind:"subWindowInitialized" },
   parent:{
      focus:{
         component:"card",
         field:"b"
      },
      data:{... parent dialog data ...},
      userValue:{... parent user data ...}
   }
}
```

Thus, through the event data a newly opened child window may read its parent's current focus and dialog data (see "Common Event Data" for more information). However, it may, in addition, interact with the parent using gui::notifyParent(...), which sends a childNotification event to the parent with any kind of streamable M-Script data.

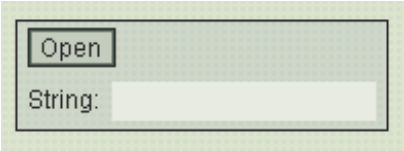
Search Handler Packages

The following is an example of the use of a search handler package. First you need a main script that opens a GUI window with an "open" button.

```
#version 15

if (!hassession()) // Ensure we have a session newsession();

var g;
if ((g=gui::get()) != null)
{
   ... // do whatever is required
}
else
{
      // Create specification
   var spec =
   {
      card:
      {
         kind: "card",
         fields:
         {
            s: "string"
         },
         buttons:
         {
            o: // Add an "open" button which opens a local package and
                  // and passes {a:10, b:20} as parameters
               {
                  kind: "open",
                  handler: loadpackage(".::handler(1)"),
```

```
                parameters:
                {

                a: 10,
                b: 20
            }
        }
    }
};
    // Create layout
var layout =
{
    name: "card",
    buttonsTop:
    [
        { name: "o", title: "Open" }
    ],
    pane:
    [
        [ "String: ", { name:"s" } ]
    ]
};

    // Create data (empty string field)
var data =
{
    card:
    {
        rows: [ { s:"" } ]
    }
};
gui::open("Open button test", "", spec, layout, data, null);
}
```

The preceding GUI specification and layout result in a window like the following.



If the button is clicked, the GUI opens a window and loads the assigned package, after which it calls the subWindowInitialized event handler. The handler looks as follows and does nothing but dump the parameters that are passed to it.

```
#version 15

package handler(1);
```

```
public procedure handleSubWindowInitialized(e)
{
    dumpvalue(e.action.parameters);
}
```

Field Specifications

All fields are defined in a single field specification object that contains one property for each of the input fields. The names of these properties are considered to be the names of the fields and should be used in the layout specification when referring to a field.

A field may either be an input field (the basic data entry field), or one of the more unusual kinds like an image or an inline button. Image fields depend on the data to supply the URL of the image—which means that you do not specify any image URL in either the component or the layout specification, as opposed to static images that are defined in the layout without referring to a field specification.

Inline button fields enable you to add a button anywhere in a component. The data for a button field specifies the button title, which makes it possible to change the button title on the fly.

The raw field kind gives full low-level control of the rendering device by simply outputting the associated data to the device.

Each field specification is an object with the following properties.

Name	Type	Description
kind	string	The field kind. This can be “input,” which is the normal user input field, “raw,” “image,” or “submit.” The kind property is optional with “input” as the default. See “Input Fields” through “Submit Fields” for more information on the different field kinds.
type	string	The type (bool, int, and so on) of an input field. Enumerated types can be specified with an array of the literals. Maconomy pop-ups can be specified as maconomy::popupTypeName, for example, maconomy::currencyType. This is valid for input fields only.
mandatory	bool	Set this to true if an input field is mandatory, meaning that a value must be entered/selected by the user. This is valid for input fields only. Optional with false as the default.
favorites	array	List of favorite values for the field. Adding such a list converts the input field to a combo box where a user can enter text as well as select text from a drop-down list.
dynamicFavorites	bool	Indicates that the field has dynamically generated favorite values.

Name	Type	Description
secret	bool	Set this to true if an input field is secret. This hides the input so that others cannot see what is entered. This is valid for input fields only. Optional with false as the default.
toolTip	string	A short descriptive help message that pops up when a user lets the mouse hover above the field. Can be overridden in the layout. Does not work for input fields where the pop-up functionality has been used to show the current content of the input (usable when the input data overflows the input box.
inputLength	int	Specifies the maximum number of characters allowed in an input field. The default value is 255 for normal input fields and unlimited for multiline text fields. This value can be made smaller in the layout.[1]
multiline	bool	Setting this to true makes the input field a multiline field with unlimited input buffer size. This value can be overridden by the layout.
allowNull	bool	This option makes it possible to return null values for empty int, real, amount, and string values. If it is not set, these return 0 (int), 0.0 (real), 0.00 (amount), or "" (string) unless the field is mandatory, in which case a value must be entered. Valid for input fields only. Optional with false as the default.
server	string	Optional Maconomy server handle that should be used to find the literals of a Maconomy pop-up type. The handle must be requested from maconomy::getServerHandle(…). Valid for input fields only. Optional with the default server as the default value.
required	bool	Normally fields are optional, but if this property is set to true, the field is required and must be included in the layout. The required and optional properties should not be used together. Optional with false as the default.
optional	bool	Normally fields are optional, but if this property is set to false the field is required and must be included in the layout. The required and optional properties should not be used together. Optional with true as the default.

[1] On Internet Explorer, this check is performed while the user enters the data, and so it is impossible to enter too many characters. On Netscape, this cannot be done, so the check is not performed before the data is submitted to M-Script.

Name	Type	Description
events	object	Specification of optional events that are required for the input field (for instance, receiving an event when a drop-down is changed). See “Field-Specific Events” for more information.
autoFetch	bool	Set this to true if you want to be able to update the favorite list depending on what the user enters in the field, for example, when a user enters a string in the field, the event field ChangedViaAutofetch is fired, and you can filter a list depending on the user data and update the content of the dynamic favorite. (This is similar to an AJAX search) Can only be set for dynamic Favorites fields.
minCharsBerforeAutoFetch	bool	A user must enter a minimum of characters before auto-fetch occurs. Required when autoFetch is enabled. Can only be set for dynamicFavorites fields.

The shorthand form of a field specification is a text string that defines the type of the field (the field is assumed to be an input field).

The list of allowed combinations is provided in the following table.

Field Kind	Property											
	type	mandatory	favorites	secret	toolTip	inputLength	multiline	allowNull	server	required	optional	events
input	*	°	°	°		°	°	°	°	°	°	°
raw										°	°	°
image					°					°	°	°
submit					°					°	°	

(* = required, ° = optional)

Input Fields

The most important property of an input field is its type property. The valid field types are most of M-Script's simple types as well as some special notations for enumerated values and Maconomy popup types.

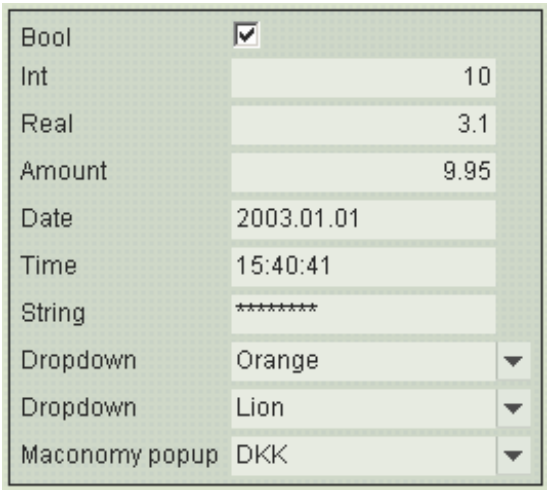
The full list of field types is described in the following table.

Type Identifier	Description
bool	Boolean value. Is rendered as a check box.
int	Integer value. Is rendered as a right-aligned text input element.
real	Real value. Is rendered as a right-aligned text input element.
amount	Amount value. Is rendered as a right-aligned text input element.
date	Date value. Is rendered as a left-aligned text input element.
time	Time value. Is rendered as a left-aligned text input element.
String	String value. Is rendered as a left-aligned text input element
[…]	Enumerated value. Is rendered as a drop-down element See "Enumerated Values" for more information.
maconomy::typeName	Maconomy pop-up value. Is rendered as a drop-down element. See "Maconomy Pop-Up Values" for more information.

The following is an example that covers all of the preceding types with their rendered output.

```
var specification =
{
   myCard :
```

```
    {
        kind : "card",
        data : "myCardData",
        fields :
        {
            b  : "bool",                // Shorthand
            i  : { type: "int",
                 mandatory: true
                 },
            r  : { type: "real" },
            a  : { type: "amount" },
            d  : { type: "date" },
            t  : { type: "time" },
            s  : { type: "string",
                 secret: true
                 },
            p1 : { type: [ "Orange", "Apple", "Banana" ] },
            p2 : { type: [ { title: "Lion", value: "L" },
            { title: "Gnu",  value: "G" } ] },
            p3 : { type: "maconomy::currencyType" }
        }

    }
};
```



Enumerated Values

Enumerated types can be defined in various ways. First of all the type can be entered as an array of literals, where each literal is either a simple text string or an object if more control of the values is needed. Each literal has a title, a value, and an index associated with it. These properties can be set individually, but if only a simple text string is entered as a literal they are assigned sequential numbers beginning from zero.

Consider the following type as an example.

```
[ "Orange", "Apple", "Banana" ]
```

The resulting title, value, and index values are as follows.

Title	Value	Index
empty	Null	-1
Orange	0	0
Apple	1	1
Banana	2	2

To specify the value directly, you must specify each literal as an object that has the following properties.

Name	Type	Description
title	string	The title of the literal.
value	simple	The value associated with the literal. This can be any (streaming) M-Script value.

The following is an example of an enumerated type specification that defines the titles and values separately.

```
 [
    { title: "Orange", value: "o" },
    { title: "Apple",  value: "a" },
    { title: "Banana", value: "b" }
]
```

In this case the values have been changed to “o,” “a,” and “b,” instead of a sequential number.

The default value for the blank (empty) literal is null. This can, however, be changed to any other (streamable) M-Script value, for instance, an integer or an object that has various properties. The blank value is changed by adding a property named blankValue to the field specification. This property must hold the exact blank value. The following is an example.

```
fields:
{
    field1:
    {
        type: [ "a", "b", "c" ],
        blankValue: 42
    },
    field2:
    {
        type: [ "a", "b", "c" ],
        blankValue: { isNull: true, other: 42 }
    }
}
```

A shorthand for defining an enumerated field is to write an array of literals instead of an object that has the properties kind, and type, for example:

```
fields:
{
   enum1:
   {
      kind: "input",
      type: ["Orange","Apple","Banana"]
   },

   enum2: ["Orange","Apple","Banana"]
}
```

Maconomy Pop-Up Values

Maconomy pop-ups are much like enumerated values, except that only the Maconomy pop-up type name must be specified. Thus, for instance, to add an input field that refers to Maconomy's "currencyType," simply state the type name as maconomy::currencyType. The GUI then takes care of fetching all of the literals from the Maconomy server and assigns titles and values (a sequential number starting from zero).

If the pop-up literal must be fetched from a special server, the server handle can be stated as part of the field specification.

Examples:

```
fields:
{
   popupA: { type: "maconomy::currencyType" },
   popupB: { type: "maconomy::currencyType",
             server: maconomy::getServerHandle(...) },
}
```

Dynamic Drop-Downs

Enumerated values that have a static set of literals may not always be enough. In a time sheet application for Maconomy, for instance, it may be convenient to have a task list that changes literals depending on the selected job. If this is combined with the fieldChanged event (see "fieldChanged Event") you can create an application where the task list is updated in the same moment that a user selects a new job number.

Dynamic drop-downs do not have any literals defined in the field specification. The literals are instead defined as part of the data, because it is something that can be changed dynamically.

In the field specification you just define the field as type dynamicDropDown.

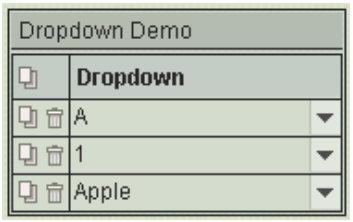
```
myTable:
{
   kind: "table",
   fields:
   {
      dd: { type: "dynamicDropDown" }
   },
   ...
```

```
}
```

In the data part you add the literals as an extra property of the field data named literals.

```
var data =
{
   myTable:
   {
      rows:
      [
         { dd: { literals:[ "A", "B", "C" ], value:0 } },
         { dd: { literals:[ "1", "2", "3" ], value:0 } },
         { dd: { literals:[ "Apple", "Orange", "Banana" ], value:0 } }
      ]
   }
};
```

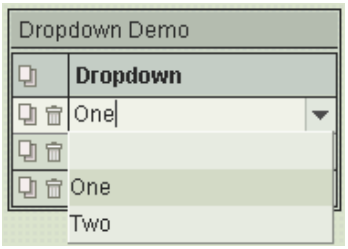
The following figure shows an example of the output.



The literals may also be updated using gui::update() as shown in the following example.

```
var g;
if ((g=gui::get()) != null)
{
   if (g.action.kind == "enter")
   {
         // Update first row in table
      gui::update("myTable",
                  {dd: {literals: ["One", "Two"], value: 0 }},

                  0);
   }
   gui::acceptData();
}
```

The output is as follows.



The literals may of course also be updated using gui::updatePartial() (see “gui::updatePartial”).

Combo Boxes and Favorite Values

Normal text input fields such as strings, integers, and dates can be supplied with a list of favorite values, such as the last ten job numbers used or the most frequently used activities in the company. An input field that has such a list is called a combo box, because it combines a normal input field and a drop-down list.

A favorite list is specified as an array of objects each of which has a title and a value property. The title is what a user sees when selecting from the list, whereas the value is what is actually inserted into the input field. An example could be a combination of job names and job numbers; the user sees the name, but the number is entered in the field.

Example:

```
fields:
{
    i:
    {
        type: "int",
        favorites:
        [
            { title: "Empty", value: ""  },
            { title: "Zero",  value: "0" },
            { title: "One",   value: "1" }
        ]
    }
}
```

Dynamic Favorites Values

Favorite lists can also be generated dynamically. To do this, the field must be marked as having dynamic favorites by setting the Boolean property dynamicFavorites to true in the field specification. The actual favorites must then be specified in the initial data or set later on using gui::update or similar.

Example:

```
var data =
{
    myCard:

    {
        rows:
        [
            { s: { value:"<empty>",
                   favorites:     // Favourites in initial data
                   [
                       { title: "This is not your favorite", value: "41" },
                       { title: "But this is", value: "42" }
                    ]
                }
            }
        ]
```

```
    }
};
```

The favorite list may then be changed using gui::update.

```
gui::updatePartial(
    "card",
    { s: { value: "Task A",
           favorites: [ { title: "Task A1", value: "101" },
                        { title: "Task A2", value: "102" }
                      ]
         }
    });
```

Raw Fields

Raw fields give a programmer full control of the rendering device by outputting the associated data directly to it.

Raw fields support only the “required” and “optional” properties of a field specification.

Typical uses for raw fields include HTML anchors and (dynamic) read-only text. Note that if links are produced in a raw field, these must contain a target=_parent specification; otherwise, the new page opens inside the GUI-II frameset, which is undetectable for the user, but if used many times it results in an unnecessarily large set of embedded frames.

Image Fields

Image fields are used to place dynamic images in a component. The URL of the image is supplied as data to the component.

Image fields support only the “required” and “optional” properties of a field specification.

Submit Fields

Submit fields are inline buttons that have submit actions (there is no support for inline undo, reset, cancel, or open buttons). The title of the submit button is supplied as data to the component.

Activating an inline button results in a normal button event being sent to the event handler with the name of the button. This means that you should be careful not to have normal buttons that have the same name as inline buttons.

Button fields support only the “required” and “optional” properties of a field specification.

A typical use of an inline button could be in a table where each row has an inline button that a user can activate to specify a certain action on a certain row. This may be more intuitive than first placing the cursor on the required row and then clicking one of the normal buttons above or below the table.

Field-Specific Events

Normally the GUI works in a record-oriented way, where events are sent when a user changes focus between records, submits a whole record, or adds/deletes a whole table row (record). However, in some cases it can be useful to receive events when the user clicks in certain fields or when users change the values of certain fields.

To enable these events you add the property events to the field specification. This property must be an object that can contain the properties fieldChanged, fieldClicked, headingClicked, and

grandTotalClicked. Each of these can be true or false, depending on whether or not the event should be enabled or disabled (the default is disabled).

Example:

```
fields:
{
    myPopup:
    {
        type: [ "Orange", "Apple", "Banana" ],
        events:
        {
            fieldChanged: true
        }
    },
    myRaw:
    {
        kind: "raw",
        events:
        {
            fieldClicked: true
        }
    },
    myImage:
    {
        kind: "image",
        events:
        {
            fieldClicked: true
        }
    }
}
```

The fieldChanged event can be associated with the text input fields (string, int, date, and so on) and pop-up fields—either static, Maconomy pop-ups, or dynamic pop-ups.

The fieldClicked event can be associated with text input fields, raw, and image fields. Associating the event with a text input field makes the input read-only.

See “fieldChangedViaAutofetch Event” and “fieldClicked Event” for information about the actual events.

Search Specifications

The searches are, in the same way as the other GUI specifications, defined in a single object, which contains one property for each of the search specifications. The names of these properties are considered to be the names of the search specifications, although the name is not used.

A single search specification is an object consisting of the following fields.

Name	Type	Description
kind	string	The search kind. You can have low-level searches, where the programmer must take care of everything that is related to the search, and dialog-specific searches where the search is handled as a Maconomy Foreign Key Search. The possible values of the kind property are “simple” and “dialog.” The default value is “simple.””
title	string	The title of the search. Shown as tool-tip information when hovering the mouse over the search icon.
script	string	The name of the M-Script program that is started when the search is initialized.
fields	array	Array of field names (strings).
xsize	int	The width of the search window (in pixels).
ysize	int	The height of the search window (in pixels).
rowsPerPage	int	The number of rows to show on each page of the search result. This field is optional, with a default value of 15 rows.
componentKind	string	The kind of component that is used to display the search result. The possible values are table and list. This field is optional, with the default value of table.
dialogName	string	The name of the dialog that a foreign key search relates to.
pane	string	The name of the dialog pane that a foreign key search relates to. This can be either “upper” or “lower.”
searchInfoPackage	package	A package that is loaded with the loadpackage function. This package can be used to do “last-minute” modifications to the searches that are handled by the standard search handler. See “Search Handler Packages” for more information.

Name	Type	Description
searchWindowPackages	string	The name of an M-Script package that handles a dialog search. If this package is not specified, the package that is specified by the initialization file option gui_searchWindowPackage is used. If this option is not set, the default value is mscript::gui::maconomy::search::searchWindow(2), which is a standard package that is installed with M-Script that takes care of standard Maconomy Foreign Key Searches. The exact interface for this package is described in the source code of the default package.

An example of a few different searches could look like the following.

```
searches:
{
   jobSearch:
   {
      title:  "Job search",
      script: "jobSearch.ms",
      fields: ["jobNo", "jobName"]
   },
   customerSearch:
   {
      title:  "Customer search",
      script: "customerSearch.ms",
      fields: ["customerNo", "customerName"],
      xsize:  250,
      ysize:  400
   },
   taskSearch:
   {
      kind: "dialog",
      dialogName: "Timesheet", title: "Task search",
      fields: [ "taskNo" ]
   }
}
```

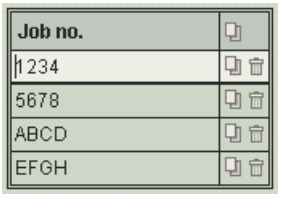
The M-Script program that handles the actual search could, for instance, be something like the following, which presents a user with four hard-coded job numbers.

```
#version 15


/****************************************** Search window specification
******************************************/
procedure openSearchWindow()
{
   var specification =
   {
```

```
        myTable:
        {
            kind: "table",
            data: "myTable",
            fields:
            {
                jobNo: { type: "string" }
            }
        }
    };

    var layout =
    {
        name: "myTable", columns:
        [
            { kind : "input", heading : "Job no.", name : "jobNo" }
        ]
    };

        // Hard-coded job numbers - might as well come from
        // an SQL request.
    var data =
    {
        myTable:
        {
            rows:
            [
                { jobNo: "1234" },
                { jobNo: "5678" },
                { jobNo: "ABCD" },
                { jobNo: "EFGH" }
            ]
        }
    };

    gui::open("Job Search Window",
            "ex006search.ms",
            specification, layout, data, {} );
}

/*****************************************
Event handler
*****************************************/
var g;
if ((g=gui::get()) != null)
{
    switch (g.action.kind)
```

```
    {
        // Create GUI-II search window when this
        // script is opened as a search script.
        case "searchInitialized":
            openSearchWindow();
        break;

        case "initialized":
            gui::setFocus("myTable", "jobNo", 0);
        break;

        case "enter":
            // Get the value (job no.) of the current row
            var value = g.data.myTable.rows[g.focus.rowIndex].jobNo;

            // and transfer it to the parent window. gui::searchTransfer({jobNo :
            value}); gui::close();
            break;

        }

    gui::acceptData();
}
```

This little search program would present the user with this window.



Common Search Specifications

A problem with the preceding search setup is that the programmer must specify a script name and window size, and so on, in the search object for every field specification. To avoid this, it is possible to write a common search specification. This is an object named searchSetup that must be placed in the component specification. The object may contain the same properties as the specific search specifications with the exception of the fields properties.

Example:

```
var specification =
{
    kind: ...,
    fields: ..., searches:
    ..., searchSetup:
    {
            // These settings are common for all searches
        script: "commonSearch.ms",
        xsize: 250,
```

```
        ysize: 400
    }
};
```

The common search settings are overwritten by settings in the specific search specifications.

Maconomy Dialog-Based Ctrl+G Searches

The common search specification is best used to supply Maconomy Foreign Key Searches for all fields in a dialog. To do this, two things must be done: First, the searches property of the dialog pane description that is returned by maconomy::dialogGetDef should be used as the searches property of the component. Next, the searchSetup facility should be used to set the search type to be “dialog” for all searches that are returned by maconomy::dialogGetDef. Example:

```
var dialogSpec = maconomy::dialogGetDef(dialogID);
var searches   = dialogSpec.lowerPane.searches;
var fields     = ... // Get fields from dialogSpec.lowerPane.columns;

var specification =
{
    kind:     "table",

    fields:   fields,
    searches: searches,
    searchSetup:
    {
        kind:       "dialog",
        dialogName: "Timesheet",
        pane:       "lower"
    }
};
```

MQL Ctrl+G Searches

The following properties must be set in a search specification to use MQL-based searching (assuming that the standard search handler package is used).

Name	Type	Description
kind	string	Must be set to “mql.”
title	string	The title of the search, used for help text and window titles.
fields	array	Array of field names from which the search can be initiated.
mqlQuery	string	The actual MQL query.

Name	Type	Description
parameterSpec	object	The field specification for all of the parameter fields involved in the MQL query. Future M-Script versions may not need this, since the information exists in the MQL query already.
restrictionFields	array	Array of field names. Lists all of the fields that should be selectable in the search restriction.
parameterValues	object	Default values for restriction fields. A parameter value is displayed and can be edited if the user selects the corresponding restriction field; otherwise, the parameter value is used in the query as stated in this object.
outputFields	array	Array of field names. Lists all of the fields that are shown in the output columns.
transferFields	array	Array of mappings from field names in the MQL query to field names in the parent GUI window.

The following is an example that searches through sales orders.

```
var mqlQuery = 'MQL';
   mselect ordernumber, name1
   from    orderheader
   where   ordernumber in pOrdernumber and pInternal = 10
   using parameters
         pOrdernumber: string,
         pInternal:    integer
MQL

var searchSpec =
{
   kind: "mql",               // This is an MQL search
   title: "Test MQL search", // Show this in window title

   fields: [ "order" ],       // Enable search from field "order"

   mqlQuery: mqlQuery,        // The MQL query
      parameterSpec:           // Parameter specification for all fields
      @{                       // - must be case insensitive!
         pOrdernumber:           // Field named "pOrdernumber"
         {
            title: "Ordernumber", // - with title "Ordernumber"
            type: "string"        // - and of type "string"
         },
         pInternal:
         {
            title: "Internal",
```

```
            type: "int"
        }
    },
    restrictionFields:        // Allow user to restrict on these fields
    [
        "pOrdernumber"          // - for instance "pOrdernumber"
    ],
    parameterValues:          // Set these default values
    {
        pOrdernumber: { value: "200001" }, // - modifiable by user
        pInternal: 10                      // - not modifiable
    },
    outputFields:             // Show these fields in the output columns
    [
        { name: "ordernumber", size: 100 },  // Specify width in pixels
        "name1"
    ],
    transferFields:           // When user selects result transfer ...
    [
        { src: "Name1", dst: "Name" } // MQL "Name1" to GUI "OrderName"
    ]
};
```

Parameter Specification

The parameter specification is an object (must be case insensitive) where each property corresponds to a field definition. Each field definition must state field title and type (any of the normal Maconomy types).

The parameter specification must cover all of the fields that are selected in the MQL query and all MQL parameter fields.

Restriction Fields

The restriction fields array is a simple array of the field names that should be part of the restriction.

Parameter Values

The parameter values object is used to specify optional default values for the MQL parameter fields. Each property in the object represents a field, and the value itself must be specified with either the usual single-value object or just the actual value.

Parameter values are either used as is to restrict the query, or they may be overridden by the user if selected in the restriction part.

Output Fields

This array specifies which fields to show in the output columns. It is an array of field names—or objects with a name property. The object notation allows the use of a size property that specifies the width of the output column (in pixels).

Transfer Fields

This is an array of mappings from MQL field names to GUI field names (in the original window from which the search was initiated). It is used to specify which MQL field values to transfer into the parent window.

The mappings are specified with objects that have both a src and a dst property—that represent the source MQL field name (src) and the destination GUI field name (dst).

MQL Ctrl+F Searches

A Maconomy Ctrl+F search can be simulated with an “open” button that opens a search window (as a script or as a package) and passes a suitable set of parameters to it. The standard search window handler package (mscript::gui::maconomy::search::searchWindow(2)) can do this under the right circumstances. The following are required.

The “open” button must specify mscript::gui::maconomy::search::searchWindow(2) as the handler package.

The “open” button must specify the properties shown in the following table in the parameters object.

The event handler of the window that uses the search package must be able to handle childNotification events from the search package. The search package sets the event property of the action object in the event object to “transfer” to signal that a user has selected a row in the search window. The data that is transferred from the search window is a copy of the selected row.

The “open” button parameter object must contain the following properties.

Name	Type	Description
mqlQuery	string	The actual MQL query.
title	string	Window title.
parameterSpec	object	The field specification for all of the parameter fields involved in the MQL query. Future M-Script versions may not need this, because the information already exists in the MQL query.
restrictionFields	array	Array of field names. Lists all of the fields that should be selectable in the search restriction.
parameterValues	object	Default values for restriction fields. A parameter value is displayed and can be edited if a user selects the corresponding restriction field; otherwise, the parameter value is used in the query as stated in this object.
outputFields	array	Array of field names. Lists all of the fields that are shown in the output columns.

In addition, the parameter object can contain any of the following properties.

Name	Type	Description
rowsPerPage	int	The number of rows to show on each page of the search result. This field is optional, with a default value of 15 rows.

Name	Type	Description
componentKind	string	The kind of component that is used to display the search result. The possible values are table and list. This field is optional, with the default value of table.

The properties are used in the same way as for Ctrl+F searches.

The following is an example.

```
#version 15

if (!hassession())
   newsession();

if (!maconomy::isLoggedIn())
   maconomy::login("Administrator", "123456");



var g;
if ((g=gui::get()) != null)
{
       // Search package notifies its parent with a "transfer" event
   if (    g.action.kind == "childNotification"
       &&  g.action.event == "transfer")
   {
          // Search package passes data from user's selected row
       var orderNo = g.action.value.data.ordernumber.value;

       gui::alert( sprint("User selected order: ^1", orderNo) );
       gui::update("card", {s: orderNo});
   }
}
else
{
       // An MQL query for orders
   var mqlQuery = 'MQL';
       mselect ordernumber, name1
       from    orderheader
       where   ordernumber in pOrdernumber and pInternal = 10
       using parameters
          pOrdernumber: string,
          pUnused:      integer,
          pInternal:    integer
   MQL

          // Parameters for the search window
```

```
    var searchParameters =
    {
        mqlQuery: mqlQuery,
        title: "Order search",
        parameterSpec:
        @{

        pOrdernumber:
        {
            title: "Order number",
            type: "string"
        },
        pUnused:
        {
            title: "Unused",
            type: "int"
        }, pInternal:
        {
            title: "Internal",
            type: "int"
        }
    },
    restrictionFields:
    [
        "pOrdernumber"
    ],
    parameterValues:
    {
        pOrdernumber: { value: "200001" },
        pInternal: 10
    },
    outputFields:
    [
        { name: "ordernumber", size: 100 },
        { name: "name1" }
    ]
};

var package = "mscript::gui::maconomy::search::searchWindow(2)";

        // GUI specification for our window
    var spec =
    {
        card:
        {
            kind: "card",
            fields:
            {
```

```
                s: "string"
            },
            buttons:
            {
                o:
                {
                    kind: "open",
                    handler: loadpackage(package),
                    parameters: searchParameters
                }
        }
    }
};

        // GUI layout for our window
var layout =
{
    name: "card",
    buttonsTop:
    [
        { name: "o", title: "Open" }
    ],
    pane:
    [
        [ "String: ", { name:"s" } ]
    ]
};

var data =
{
    card:
    {
        rows: [ { s:"" } ]
    }
};

gui::open("Open button test", "", spec, layout, data, null);
}
```

searchInfoPackage

When the standard search handler package is used for dialog-based searches (not MQL searches) you can specify a package that can do last-minute corrections to the search specifications based on the current input data. To do this, the property searchInfoPackage must be set to a loaded package in the search specification.

```
var specification =
{
    kind: ...,
```

```
    fields: ...,
    searches: ...,
    searchSetup:
    {
        searchInfoPackage = loadpackage("mySearchInfo(1)");
    }
};
```

The search info package *must* contain a public function named getMaconomyDialogSearchInfo, which accepts two parameters (search name and parent window information). This function may then return an object with new values for the properties outputFields and restrictionFields.

The object that is returned must look like the following.

```
{
    outputFields:
    {
        fields:
        [
            { name: "field name", size: pixel width },
            ...

        ]

        mergeFields: true/false
    },
    restrictionFields:
    {
        fields: array of field names,
        mergeFields: true/false
    }
}
```

Each of the outputFields and restrictionFields contains an object that has two properties—the fields property contains an array of field names, and the mergeFields property defines whether the new fields are merged with the existing ones or overwrite them completely (the default is to merge).

The following is a complete example of a search modifier.

```
public function getMaconomyDialogSearchInfo(searchName, parentInfo)
{
        // Parent window has focus in "Customernumber", so specialize for
        // a customer search
    if (parentInfo.focus.field == "Customernumber")
        return
        {
            outputFields:
            {
                fields:
                [
                        { name: "Chargecode1", size: 80 },
```

```
                    { name: "Chargecode2", size: 40 }
            ]

            // No "mergeFields" here, so merge with existing outputFields
        },
        restrictionFields:
        {
            fields: ["TheCustomerNumber", "Name1"],

                // Overwrite existing fields
            mergeFields:false
        }
    };
else
        // Parent window has focus in "Itemnumber", so specialize for
        // an item search
    If (parentInfo.focus.field == " Itemnumber ")
        return
        {
            restrictionFields:
            {
                fields: ["ItemNumber", "ItemGroup"],
                //Merge these fields into existing restrictionFields
                mergeFields: true
            }
        };
        else
            return {}; //All other cases don't add anything
    }
```

Row Actions

The row-specific actions “insert row,” “delete row,” and “select row” can be moved around in the layout and even left out completely if required. However, to use the actions in the layout, they must be specified in the component specification first.

The default setup is to enable both “insert row” and “delete row,” and disable “select row.” Actually, the “insert row” and “delete row” are combined to the “resize” action. Combining them results in a nicer layout, but does not change any functionality.

The row actions are enabled/disabled in an object called rowActions in the component specification. This object contains one Boolean property for each action—setting it to true enables the action and vice-versa.

Example:

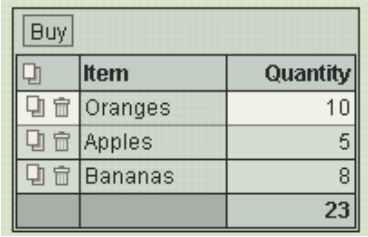
```
var specification =
{
    myTable:
    {
        kind: "table", fields:
        ..., rowActions:
```

```
        {
            selectRow: true,  // Enable row selector
            insertRow: false, // Disable row insert
            deleteRow: true   // Enable row delete
        }
    }
};
```

See “Row Actions” for the layout specification.

Grand Total Rows

The GUI also supports automatic, as well as manual, calculation of grand totals in a table. With this feature you can sum up hours that are spent on something, the number of items in a sales order, and so on. All that is needed is to specify that one or more table rows should have automatic grand totals associated with them.



To get the grand total as illustrated in the preceding figure, you must add a grandTotals object to the specification.

```
var specification =
{
    table:
    {
        kind:"table",
        fields: ..., grandTotals:
        {
            fields:
            {
                quantity: "sum"
            }
        },
        buttons: ...
    }
};
```

As this example shows, the grand totals definition is an object that has one property for each field for which grand totals are required. The “sum” string in the preceding example is actually a shorthand for a more complex object structure:

Property	Type	Description
kind	string	Specification of grand total kind. Four possibilities exist: “none” for no totals, “sub” for automatic totals, “static” for a fixed value, and “custom” for a value that can be changed by the programmer.
title	string	This is the exact value of the total in case of a “static” grand total. This value is ignored for every other kind of total.
trafficLighting	string	Optional traffic lighting setup for the totals. See “Traffic Lighting.”

The automatic “sum” totals can only be calculated for integers, reals, and amounts.

The “custom” totals can be updated with a call to gui::updateGrandTotals .

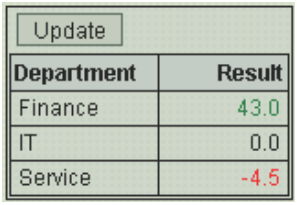
Grand totals are only updated after a call to gui::update, gui::deleteRow, gui::insertRow, or gui::setRows. This is important to note, because some of these functions must be called manually after a user has changed some data. The grand totals are, for instance, not updated automatically just because a user enters something in an input field.

Grand totals can be enabled for both tables and read-only lists.

Traffic Lighting

Traffic lighting is a technique for highlighting data depending on what value it has at the moment when it is rendered. It is often used in dashboards screens and similar summary pages, where you want to highlight specific data such as projects that are overdue.

The following is an example where a field is colored red for negative numbers and green for positive numbers.



Traffic Lighting Specification

Traffic lighting is specified in the GUI field specification part as an object named trafficLighting with the following properties.

Name	Type	Description
algorithm	string	Name of the algorithm that calculates the color. The algorithm refers to the name of an M-Script function that must return a CSS class name, which is applied to the field.

Name	Type	Description
parameters	array of strings	Array of field names, the values of which are to be passed as parameters to the traffic lighting algorithm.

The preceding red/green example was constructed with the following GUI specification.

```
var specification =
{
    table:
    {
        kind: "list",
        fields:
        {
            ...,
            result:
            {
                type: "real",
                trafficLighting:
                {
                    algorithm: "redGreen"
                }
            }
        },
        trafficLighting:
        {
            handler: loadpackage(".::trafficLight(1)")
        },
        buttons: ...
    }
};
```

The traffic lighting specification consists of two parts—a field-specific one that specifies the algorithm, and a component-specific one that specifies the package in which traffic lighting algorithms can be found. The global specification must be placed in an object named trafficLighting side-by-side with the fields and buttons specifications.

Traffic Lighting for Grand Totals

Traffic lighting can also be specified for grand totals in a table. All that is required is a traffic lighting specification in the grand totals specification. The format of the trafficLighting object is identical to that of the normal fields.

```
grandTotals:
{
    fields:
    {
        total:
        {
            kind: "sum",
            trafficLighting: { algorithm: "redGreen" }
```

```
        },
        ...
    }
}
```

To help calculating traffic lighting for grand totals you can add a trafficLightingGlobals object to the data for the component. This object can contain values that assist the traffic lighting calculation.

```
var data =
{
    table:
    {
        rows: ...,
        trafficLightingGlobals:
        {
            employeeNumber: "1010"
        }
    },
    composer: {}
};
```

The traffic lighting globals are available in the info parameter that is passed to the algorithm function. They are also available (unmodified) in the data object in the same way as the rows are. However, the data is read-only just like the rows, so the gui::updateTrafficLightingGlobals function should be used to modify the values.

Traffic Lighting Calculation

The actual calculation of the traffic lighting value (a CSS class name) is done by functions in the traffic lighting package mentioned in the component-specific traffic lighting setup. Such a package must supply public functions named exactly as the algorithms that are mentioned in the traffic lighting specification. The previously mentioned red/green function would, for instance, look like the following.

```
#version 14

package trafficLight(1);

public function redGreen(info, p0)

{
    if (p0 < 0)
        return "neg";

    if (p0 > 0)
        return "pos";

    return null;
}
```

The parameters that are passed to an algorithm function are defined in the traffic lighting specification, but it always includes the info and p0 parameters. Additional parameters can be passed with values from other fields in the record.

The info parameter is an object that has the following properties.

Name	Type	Description
fieldName	string	Name of the field for which the traffic lighting is calculated. May be useful if the algorithm is used for multiple fields
trafficLightingGlobals	object	The traffic lighting global values from the traffic lighting specification.

The p0 parameter contains the value of the field for which traffic lighting is calculated.

The return value must be the name of a CSS class or null for no traffic lighting.

Cascading Style Sheet Setup

The GUI framework is installed with three sets of styles to be used for traffic lighting. The styles are more or less only supplied for a proof of concept and simple testing, since most applications require their own styles. The styles are warm0–warm4, cold0–cold4, neg, and pos. The warm and cold ranges color the field background in a red/yellow palette and a blue palette. The pos and neg styles color the text green and red.

New styles are slightly difficult to add due to the complexity of the style sheet for the GUI. The problem is that the style must be stated for cards, odd/even rows, lists, and grand totals. In addition to this, the style name must be preceded with “tl-“ to ensure that it is recognized as a traffic lighting style.

For a style named “bold” you would need to write the following style rule to make the text stand out as bold.

```
input.tl-bold,
table.grid tr.even input.tl-bold,
table.grid tr.odd input.tl-bold,
table.grid tr.selected input.tl-bold,
table.list td.tl-bold,
tr.grandTotals td.tl-bold
{
    font-weight: bold;
}
```

The styles may be added to a separate style sheet using a stylesheetURL property in the layout as mentioned in “Common Layout Features.”

Extra Events

In addition to the usual “user changed focus” event that is received only when a user also changes some data, it is also possible to receive a similar event when a user changes focus even if no data has been modified. However, this does have a performance impact, because the server-side script must be invoked every time that the user changes focus between either rows or different components. For this reason, the event must be enabled by the programmer.

To enable the gotFocus event, the new property named extraEvents exists in the component specification. This property is an object that can contain the Boolean property gotFocus.

Example (enables gotFocus event when changing between the rows in a table):

```
var spec =
{
```

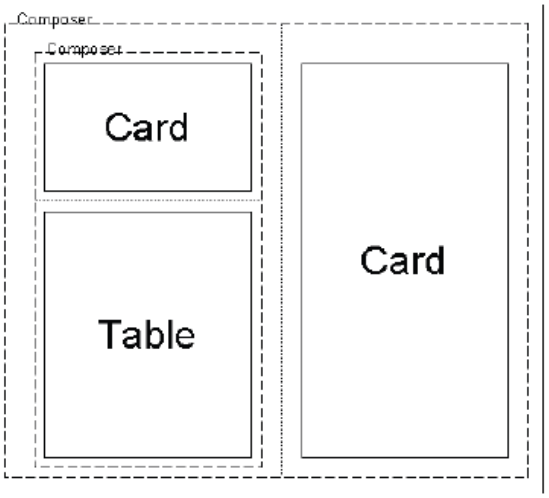
```
    myTable:
    {
        kind: "table",
        fields: …,
        extraEvents:
        {
            gotFocus: true
        }
    },
    …
};
```

See “gotFocus Event” for more information.

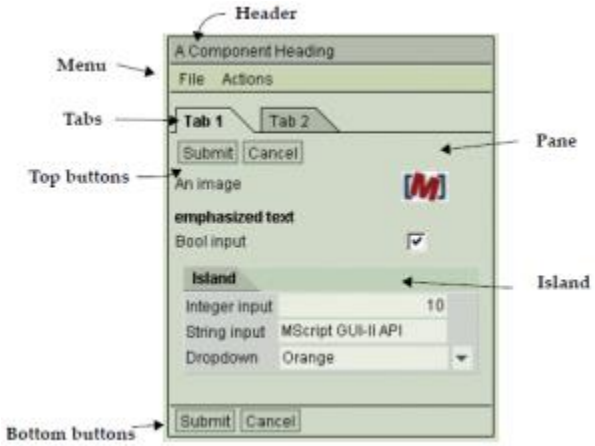
Layout Specification

The layout specification is an object that has a recursive structure; the basic components are the card and table layouts. These cannot contain other components and are considered the basic building blocks. The “glue” that is used to put these components together is the composer component, which encapsulates card and table components, as well as other composers, in a two-dimensional grid layout.

To illustrate this with an example, you could have a system that has two card components and a table formatted like the following figure.



All three kinds of components have quite a few layout features in common. These include the possibility to add a header, a menu, some buttons, tabs, and islands as shown in the following figure.



Common Layout Features

The following properties are common for card, table, and composer components.

Name	Type	Description
name	string	The name of the component that this layout represents.

Name	Type	Description
header	array	Optional header specification. See “Header.”
menu	array	Optional menu specification. See “Menu.”
buttonsTop	array	Optional button specification of the top button row. See “Buttons.”
buttonsBottom	array	Optional button specification of the bottom button row. See “Buttons.”
buttonsTabs	array	Optional specification of tabs. See “Tabs.”
borders	bool	Optional identification of whether a border should be drawn or not. The default value is true.
stylesheetURL	string	Optional URL to a style sheet for this specific component. Typically used in components with HTML in raw fields or for traffic lighting styles.
style	string	Optional specification. Use “compact” for tables that have a smaller font, icons, and line spacing.

Example layout specification of a card component:

```
var layout =
{
       // Common features
   name: "myCard",
   header: [...],
   menu: [...],
   buttonsTabs: [...],
   buttonsTop: [...],
   buttonsBottom: [...],

         // Card specific
   pane: ...
};
```

Header

The header layout is specified in a two-dimensional grid/table with the possibility to insert various elements in each of the cells. The GUI only supports static text in the header, but future versions may also allow dynamic data in the header.

A header element is an object that has the following properties.

Name	Type	Description
kind	string	The element kind. The value must be “text.”

Name	Type	Description
title	string	The actual text of the element.
emphasized	bool	The formatting of the text, emphasized or not. This field is optional, with false as the default.
span	int	Optional row span value. This can be used to make header text flow across multiple columns.

The shorthand form of an element specification is a text string which is printed as it is.

Header example:

```
var layout =
{
    ...,
    header:
    [
        [ "Here is some Header information" ],
        [ { kind: "text",
            emphasized: true,
            title: "Emphasized header" } ]
    ],
    ...
};
```

Menu

Menus and buttons are actually much like each other—they both represent the buttons that are specified in the component specification, they may both refer to the same buttons (even multiple times), and anything that can be defined for a button can be defined for a menu, too. The difference between menus and buttons lies only in how they are rendered—their functionality is identical.

The menu of a component is specified as an array of top-level menu items. These represent the anchor points of the menu items, and each of them contains the actual pull-down menu data—again stored as an array of menu items.

Each top-level item is represented as an object with the following properties.

Name	Type	Description
title	string	The title of this menu entry.
content	array	The actual content of the menu (see the following table).

The content property of the menu items can contain the following properties.

Name	Type	Description
title	string	The title of this menu entry.

Name	Type	Description
content	array	The optional sub-content of the menu. This is a reference to an array of the same kind as the one that content was found in.
button	string	The name of the button that this menu entry refers to. This property is optional, but must be used together with the component property.
component	string	The component of the button that this menu entry refers to. This property is optional, but must be used with the button property.

Example:

```
var layout =
{...,
    [
       { title: "File",
          content:
          [
             { title: "Open",  component: "myCard", button: "m1" },
             { title: "Close", component: "myCard", button: "m2" }
          ]
       },
       { title: "Actions",
          content:
          [
             { title: "Submit",  component: "myCard", button: "m1" },
             { title: "Approve", component: "myCard", button: "m2" },
             { title: "Other",
                content:
                [
                   { title: "Change",
                     component: "myCard", button: "m2" },
                   { title: "Reopen",
                     component: "myCard", button: "m2" }
                ]
             }
          ]
       },
       ...,
    }
];
```

Buttons

Buttons can be placed either at the top or the bottom of the component, but no matter how they are used, the layout specification of a button remains the same. Both the buttonsTop and buttonsBottom hold an array of the required button references where each reference is an object that has the following properties.

Name	Type	Description
title	string	The title of the button.
name	string	The name of the button (refers to the property name that holds the button specification).
size	int	The width of the button (in pixels).
toolTip	string	A short descriptive help message that pops up when a user lets the mouse hover above the field. Overrides any tool tip in the specification.

Example:

```
var layout =
{
    ...,
    buttonsTop:
    [
        { name: "b1", title: "Approve" },
        { name: "b2", title: "Reject", size: 100 }
    ],
    buttonsBottom:
    [
        { name: "b3", title: "Ok" },
        { name: "b4", title: "Cancel" }
    ],
    ...
};
```

Tabs

Tabs are in fact nothing but buttons in disguise—they behave like buttons, must be defined as submit buttons, and may be enabled and disabled in the same way as buttons. The only difference is their appearance. This has a few implications; first of all, a programmer has the responsibility for disabling and enabling the right tabs at the right time; second, the programmer also has the responsibility of reopening the page with a layout that depends on the current tab.

The Maconomy Portal Development Group has created some special packages that make the handling of tabs much easier. Please contact Deltek for further information.

Tabs are included in a layout by adding a property named buttonsTabs.

Example:

```
var layout =
{
    ...,
    buttonsTabs: [
        { name: "b1", title: "Main" },    // A tab named "Main"
        { name: "b2", title: "Details" } // A tab named "Details"
    ],
```

```
    ...
};
```

Card Layout

The card layout has one extra property—the pane property—in addition to the common layout properties. This property holds a two-dimensional M-Script array that represents the layout grid of the pane. Each of the array entries stores a layout element, which may be some simple text, a field reference, an image, and much more. The layout elements are objects that have various properties, depending on the element kind.

At the top level the pane property looks as follows.

```
var layout =
{
    ...,
    pane:
    [
        [ {...}, {...},...],
        [ {...}, {...},...],
        ...
    ],
    ...
};
```

Each of the layout elements can contain the following properties.

Name	Type	Description
kind	string	Defines the element kind. Valid values are “text,” “input,” “raw,” “button,” “image,” “island, and “spacer.” Optional, with the default value “input.”
title	string	Defines the title for text elements.
emphasized	bool	Defines whether a text element is rendered as bold or not.
name	string	Name of the input field used in an “input” element.
size	int	The size of an input field. The metric that is used depends on the rendering device. With HTML it is the visible number of characters.
mandatory	bool	Overrides the mandatory field in the specification. You cannot make mandatory fields non-mandatory in this way.
readOnly	bool	Setting this to true makes the input field read-only, independent of the current enabling specified in the GUI data. The enabling can still be changed programmatically, but without any effect.
open	bool	The opposite of readOnly.

Name	Type	Description
toolTip	string	A short descriptive help message that pops up when a user lets the mouse hover above the field. Overrides any tool tip in the specification.
inputLength	int	Specifies the maximum number of characters allowed in an input field. The default value is 255 for normal input fields and unlimited for text fields that are defined as multiline fields in the field specification. The input length can only be made smaller in the layout.[2]
multiline	bool	Specifies an input field as being a multiline input field. The cols and rows properties must also be specified in conjunction with this. Specifying a field as multiline in the layout does not modify the inputLength property.
cols	int	The width (number of characters) of a multiline input field.
rows	int	The height (number of rows) of a multiline input field.
span	int	The number of columns that an element may span.
source	string	The URL of a static image
align	string	Horizontal alignment of the element. Possible values are “left,” “center,” and “right.”
valign	string	Vertical alignment of the element. Possible values are “top,” “middle,” “bottom.”
pane	array	The pane of an island element.
height	int	The height of a spacer element (in pixels).
width	int	The width of a spacer element (in pixels).

[2] On Internet Explorer, this check is performed while a user enters the data, and so it is impossible to enter too many characters. On Netscape, this cannot be done, so the check is not performed before the data is submitted to M-Script.

The preceding properties can be combined in various ways as shown in the following table.

Kind	Property																	
	title	emphasized	name	size	mandatory	readOnly	open	toolTip	multiline	cols	rows	span	source	pane	align	valign	height	width
text	*	°						°				°			°	°		
input			*	°	°	°	°		°	°	°	°			°	°		
raw			*					°				°			°	°		
button			*	°		°	°	°				°			°	°		
image			°					°				°	°		°	°		
island	*											°		*	°	°		
spacer												°			°	°	°	°

(* = required, ° = optional)

The shorthand notation is a text string instead of an object. This inserts the text as if it were a text element.

Examples:

```
{ kind: "input",          // Normal input field
   name: "field_1a" },

{ name: "field_1b" },     // Normal input field - shorthand

{ kind: "input",          // Multiline input field
   name: "field_2",
   multiline: true,
   cols: 60,
   rows: 8 },

{ kind: "raw",            // Raw input field
   name: "field_3" },

{ kind: "button",         // Inline button
   name: "field_4" },

{ kind: "image",          // Static image (static URL)
   source: "http:..." },

{ kind: "image",          // Dynamic image (URL defined by data)
```

```
   name: "field_5" }
```

A complete card layout specification could like the following example.

```
var layout =
{
   name : "myCard",
   header: [ [ "A Component Heading" ] ],
   menu:
   [
      { title: "File",
        content:
        [
           { title: "Open",  component: "myCard", button: "m1" },
           { title: "Close", component: "myCard", button: "m2" }
        ]
      }
   ],
   buttonsTop:
   [
      { name: "b1", title: "Submit" },
      { name: "b2", title: "Cancel" }
   ],
   pane:
   [
      [ "An image",
        { kind:   "image",
          source: "../Maconomy_M.gif" }
        ],
        [ { kind:       "text",
            title:      "emphasized text.",
            emphasized: true }
        ],
        [ "Bool input",
           { kind: "input",
             name: "field_b" }
        ],
        [ "Integer input",
           { kind: "input",
             name: "field_i" }
        ],

        [ "String input",
           { kind: "input",
             name: "field_s" }
        ],
        [ "Dropdown",
           { kind: "input",
```

```
                name: "field_p" }
            ]
        ],
        buttonsBottom:
        [
            { name: "b1", title: "Submit" },
            { name: "b2", title: "Cancel" }
        ]
    };
```

Islands

Islands can be used to make groups of closely related information. For a customer card, this could be address information in one island and billing information in another island.

Islands are inserted as an element of kind island that has the properties title and pane set. The title property is a string, which is shown in the title bar. The pane property contains a two-dimensional array, just like a card component, which can contain most of the elements that can be shown in a card pane (except an island).

Islands are a layout feature only, and do not require any special setup or configuration.

Example:

```
var layout =
    {
        ...,
        pane:
        [
            [
                {
                  kind: "island",
                  title: "Island 1",
                  pane:
                  [
                    [ "Field A" , { name: "a" } ],
                    [ "Field B" , { name: "b" } ]
                  ]
                }
            ],
            [
                {
                   kind: "island",
                   title: "Island 2",
                   pane:
                   [
                       [ "Field D" , { name: "d" } ],
                       [ "Field E" , { name: "e" } ]
                   ]
                }

            ]
```

```
        ]
    };
```

The following is the output.

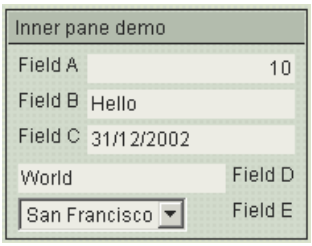
Inner Panes

You can also specify inner panes or sub-panes in a card pane. Much like an island, this allows a user to group information, but without any special visualization. Inner panes are simply inserted as a two-dimensional array, instead of an element of a certain kind, as is the case with all of the other card elements.

Example:

```
var layout =
{
    ...,
    pane:
    [
        [
            [ // Inner pane begin
                [ "Field A" , { name: "a" } ],
                [ "Field B" , { name: "b" } ],
                [ "Field C" , { name: "c" } ]
            ] // Inner pane end
        ],
        [
            [ // Inner pane begin
                [ { name: "d" }, "Field D"
                ], [ { name: "e" }, "Field E" ]
            ] // Inner pane end
        ]
    ]
};
```

The following is the output.



Spacers

Space between the elements in a card can be added with a simple “spacer” element. Such an element has two optional properties—width and height—that specify the spacing in pixels.

Example:

```
var layout =
    {
        ...,
        pane:
        [
            [
                {
                    kind: "island",
                    title: "Island 1",
                    span: 2, // Span in order to cover the spacer element
                    pane:
                    [
                        [ "Field A" , { name: "a" } ],
                        [ "Field B" , { name: "b" } ],
                        [ "Field C" , { name: "c" } ]
                     ]
                }
        ],
        [
            { kind: "spacer", height: 50 }
        ],
        [
            { kind: "spacer", width: 30 },
            {
                kind: "island",
                title: "Island 2",
                pane:
                [
                    [ { name: "d" }, "Field D" ],
                    [ { name: "e" }, "Field E" ]
                ]
            }
        ]
    ]
};
```

The following is the output (the small boxes illustrate the spacer elements).

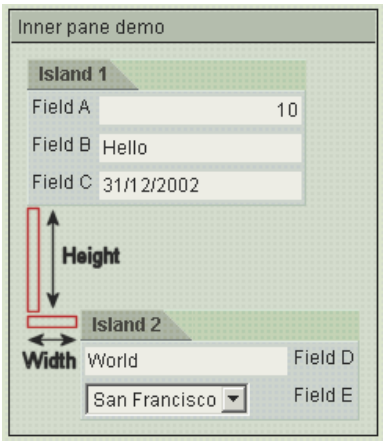


Table Layout

The table layout is made up of two parts—the column specifications and an optional *details island* specification. The column specification is stored in the `columns` property as an array of objects that has the following properties.

Name	Type	Description
kind	string	Defines the element kind. Valid values are “input,” ”raw,” “button,” and “image.” Text elements are not allowed.
heading	string	Column heading.
name	string	Name of the input field used in an “input” element.
mandatory	bool	Overrides the mandatory field in the specification. It is not possible to make mandatory fields non-mandatory in this way.
readOnly	bool	Setting this to true makes the input field read-only, independent of the current enabling specified in the GUI data. The enabling can still be changed programmatically but without any effect.
open	bool	The opposite of readOnly.
size	int	The column width (in pixels).
multiline	bool	Specifies an input field as being a multiline input field. The cols and rows properties must also be specified in conjunction with this.
cols	int	The width (number of characters) of a multiline input field.
rows	int	The height (number of rows) of a multiline input field.

The allowed combinations of the preceding properties are not as complex as for a card element, but for completeness they are described in the following table.

Element kind	Property								
	name	heading	mandatory	readOnly	open	size	multiline	cols	rows
input	*	*	°	°	°	°	°	°	°
raw	*	*				°			
button	*	*		°	°	°			
image	*	*				°			

(* = required, ° = optional)

There is no shorthand notation.

Example:

```
var layout =
{
   name : "myTable",
   borders: false,
   columns:
   [
      { kind: "input", heading: "Bool",     name: "field_b" },
      { kind: "input", heading: "Int",      name: "field_i" },
      { kind: "input", heading: "String",   name: "field_s" },
      { kind: "input", heading: "Dropdown", name: "field_p" }
   ]
};
```

Details Island

The table details island is specified in a property named `details` and is itself an object that has two properties.

Name	Type	Description
title	string	The details island title (rendered as a tab element).
pane	array	An array that describes a pane exactly as if it were a card layout.

Details islands cannot be used for raw fields and dynamic drop-downs.

Example:

```
var layout =
```

```
{
    name : "myTable",
    borders: false,
    columns:
    [

        { kind: "input", heading: "Bool",     name: "field_b" },
        { kind: "input", heading: "Int",      name: "field_i" }
        ],
        details:
        {
            title: "Details",
            pane:
            [
                [ "String: ",   { name: "field_s" } ],
                [ "Dropdown: ", { name: "field_p" } ]
            ]
        }
    };
```

Row Actions

Row actions are specified in a property named rowActions at the top level of the table layout specification. This property is an object that has two properties named left and right, each of which is an array of row action names. Adding row actions to the left is as simple as adding the name of the action to the left array.

By default the insert and delete row icons are placed to the left, and no other row actions are enabled.

Example:

```
var layout =
{
    name: "myTable",
    columns: ...,
    rowActions:
    {
        left:  [ "selectRow",
                 "deleteRow" ], // Row select and delete to the left
        right: [ ]              // No actions to the right
    }
};
```

The following is the output.

		Int	Date	String
☑	🗑	10	01/01	Hydrogen
☐	🗑	10	02/01	Helium
☑	🗑	10	03/01	Oxygen
☑	🗑	10	04/01	Argon

Possible action names are shown in the following table.

Action Name	Description	HTML Renderings
insertRow	Insert row action.	
deleteRow	Delete row action.	
selectRow	Row selectors (check boxes for selecting rows).	
Resize	Combined insert and delete row (nicer-looking layout, but same functionality as having insert and delete row separated).	

List Layout

A list layout is identical to a table layout with the following exceptions:

- Details islands are not allowed.
- Mandatory and read-only specifications are ignored.
- Row actions are not allowed.

Composer Layout

The composer layout is specified in a two-dimensional grid—much like the card layout. This is done using a two-dimensional M-Script array where each cell represents one layout element.

Each element may either be a static text or image, or the complete layout of another component—be it a card, a table, or even another composer.

The layout grid is stored in the layout property pane.

```
var layout =
{
    ...,
    pane:
    [
        [ {...}, {...}, ... ],
        [ {...}, {...}, ... ]
    ],
    ...
};
```

Each element is an object that has the following properties.

Name	Type	Description
kind	string	Defines the element kind. Valid values are “text,” “image,” “spacer,” and “component.”
title	string	Defines the title for text elements.
emphasized	bool	Defines whether a text element is rendered as bold or not.
source	string	The URL of a static image.
span	int	The number of columns that an element can span.
layout	object	The complete layout specification of a sub-component.
height	int	The height of a spacer element (in pixels).
width	int	The width of a spacer element (in pixels).

The allowed combinations of properties are shown in the following table.

Element kind	**Property**						
	title	emphasized	source	span	layout	width	height
text	*	°		°			
image			*	°			
spacer						°	°
component				°	*		

(* = required, ° = optional)

The shorthand notation is a text string that represents a text element.

The text, image, and spacer elements are described in the card layout section.

The height of each of the layout grid’s rows can be specified when necessary. This is done using the heights property on the composer layout object. The heights are specified in a table that has one entry for each row of the layout grid, and the heights are specified in percent of the screen size (0-100). The sum of all heights must be 100.

Because a relative size like this must be computed on the basis of something else, the rendering machine must be able to know the size of the enclosing component(s). This means that if any sub-component has a height specified, all of its parent components must also have their heights specified.

Heights do not work with the Netscape browser.

Example:

```
var cardLayout =
{
   name: ...,
   pane: ...
};

var tableLayout =
{
   name: ...,
   columns: ...
};

var layout =
{
   name: "myComposer",

   heights: [ 10, 25, 65 ],  // 10% for text,
                             // 25% for card and 65% for table
   pane:
   [
    [
      "Hello",
         { kind: "image",
           source: "http://www.maconomy.com/Maconomy.gif" }
     ],
     [
         { kind: "component", layout: cardLayout}
     ],
     [
         { kind: "component", layout: tableLayout, span: 2}
      ]
   ]
};
```

Data Specification

The data specification is an object that describes the data that is displayed in a GUI window. Such an object must be passed to calls to gui::open() to provide the initial data and access for a GUI window.

Internally, the GUI holds a copy of the data object, and this copy is kept in sync with the data that is displayed on the target device. The programmer can access a copy of the current internal GUI data specification by calling gui::get().

At the top level the data specification object must have a property for each component that is specified in the component specification (see “Component Specification”). These properties must have the names of the component. If, for instance, the component specification defines three components named myCard (a card component), myTable (a table component), and myComposer (a composer component), the data specification should look like the following.

```
{
   myCard     : {...},
   myTable    : {...},
   myComposer : {...}
}
```

Each property has an object value—a component data object—whose properties depend on the kind of component.

Name	Type	Description
rows	array	An array of records. For a card component, this array holds exactly one record, and for a table component there is one record for each table row. See “The record Object Type.”
fields	object	An object that describes certain attributes of the fields (that is, input fields and inline buttons) in the component, for example, which fields are enabled. See “The fields Object Type.”
buttons	object	An object that describes certain attributes of the buttons in the component, for example, which buttons are enabled. See “The buttons Object Type.”
selectedRows	array	An array with the initial values of “selected rows” check boxes. The array must contain exactly one Boolean value for each of the data rows (a “true” value means that the row is initially selected);
enabled	bool	If this property is true, the component is enabled. Otherwise, it is disabled, which means that all input fields are closed, and all inline buttons are disabled.

The following table illustrates which properties can be used with different component kinds when passing a data specification object to gui::open().

Component Kind	Property				
	rows	fields	buttons	selectedRows	enabled
card	*	°	°		°
table	*	°	°	°	°
composer			°		°

(* = required, ° = optional)

Only the properties that are marked with * or ° are recognized by the GUI when calling gui::open(), and these are the only properties that are maintained by the GUI in the internal copy of the data specification that is available to the programmer by calling gui::get().

Note that the enabled property is always ignored by gui::open(). To disable a component initially, the programmer must call gui::disableComponent() when the initialized event is received.

The record Object Type

The record object type contains data for one record—that is, data for a card component or data for a single row in a table component.

The names of the properties are the names of the fields in the record. Each field has a value, which can either be a simple M-Script value or a fieldValue object.

The properties of a fieldValue object are described in the following table.

Name	Type	Description
value	simple	The value of the field.
selected	int	The index of the literal if the field is a drop-down field.

A simple value, *x*, is a shorthand notation for the object notation

```
{
   value : x
}
```

When a record object is passed to gui::open(), gui::update(), or gui::setRows(), the following rules should be observed:

- For non-drop-down input fields and inline buttons (submit fields) the value property of the fieldValue object is required. The type of the value property must match the field type specified in the component specification (see “Field Specifications”).

  (For inline buttons the value property must have the type string, and its value is used as the button title.)

- For drop-downs and Maconomy pop-up fields either the value property or the selected property of the fieldValue object must be set. If both properties are set, selected takes precedence. If only the value property is set, it is matched against the literal values of the drop-down to determine which literal to select.
- Each field in the component must have its value defined. This means that there must be a property in the record object for each field in the component.
- The null value can be given for any type of input field. This yields an empty field.
- Field names are not case-sensitive.

Example:

```
var data =
{
   myCard :
   {
      rows :
      [
         {
            inlineButton : “Button Title”,
            adropdown : {selected : 2 },
            anInputField : { value : 42 },
            …
         }
      ],
      …
   }
…
};
```

When you call gui::get() the data specification object that is returned conforms to the following rules:

- The field values in records are always presented using fieldValue objects. The simple value shorthand notation is not used.
- For non-drop-down input fields and inline buttons (submit fields) the value property is set.
- For drop-down and Maconomy pop-up fields both the value property and the selected property are set. The value of the value property is the literal value of the selected literal.

Example:

```
{
   myCard :
   {
      rows :
      [
         {
            inlineButton : { value : "Button Title"},
            aDropdown    : { selected : 2, value : "Third literal" },
            anInputField : { value : 42 },
            ...
         }
```

```
        ],
        ...
    }
```

The fields Object Type

The fields object type contains attribute information for fields (that is, input fields and inline buttons) in a card component or columns in a table component.

The names of the properties are the names of fields in the component. Each property value can be either a bool value or a fieldAttr object.

The properties of a fieldAttr object are described in the following table.

Name	Type	Description
enabled	bool	If this property is true, the corresponding field or column is enabled. Otherwise, it is disabled.
disabled	bool	If this property is true, the corresponding field or column is disabled. Otherwise, it is enabled.

A bool value, x, is a shorthand notation for the fieldAttr object notation

```
{
   enabled : x
}
```

When a fields object is passed to gui::open() or gui::setFieldAccess the following rules should be observed:

- If both the enabled property and the disabled property of a fieldAttr object are set when calling gui::open() or gui::setFieldAccess the enabled property takes precedence.
- The fields object need not have properties for all fields or columns in the component. Omitted fields are left unchanged (when calling gui::setFieldAccess()) or enabled (when calling gui::open()).

Example:

```
   var data =
   {
      myCard :
      {
         ... fields :
         {
            inlineButton : false,           // Disabled
            aDropdown    : { disabled : true },
            anInputField : { enabled : false },
            ...
         }
         ...
      }
      ...
};
```

When you call gui::get() the data specification object that is returned always uses a normalized form for the fields object type using fieldAttr objects with the enabled property set.

Example:

```
{
   myCard :
   {
      ...
      fields :
      {
         inlineButton : { enabled : false },
         aDropdown    : { enabled : false },
         anInputField : { enabled : false },
         ...
      }
      ...
   }
   ...
}
```

The buttons Object Type

The buttons object type contains attribute information for buttons (*not* inline buttons) in a card, table, or composer component.

The names of the properties are the names of buttons in the component. Each property value can be either a bool value or a buttonAttr object.

The properties of a buttonAttr object are described in the following table.

Name	Type	Description
enabled	bool	If this property is true, the corresponding button is enabled. Otherwise, it is disabled.
disabled	bool	If this property is true, the corresponding button is disabled. Otherwise, it is enabled.

A bool value, x, is a shorthand notation for the buttonAttr object notation

```
{
   enabled : x
}
```

When a buttons object is passed to gui::open() or gui::setButtonAccess the following rules should be observed:

- If both the enabled property and the disabled property of a buttonAttr object are set when calling gui::open() or gui::setButtonAccess, the enabled property takes precedence.
- The buttons object need not have properties for all buttons in the component.

  Omitted buttons are left unchanged (when calling gui::setButtonAccess()) or enabled (when calling gui::open()).

Example:

```
var data =
{
   myCard :
   {
      ... buttons :
      {
         start : false,         // Disabled
         pause : { disabled : true },
         stop  : { enabled : false },
         ...
      }
      ...
   }
   ...
};
```

When you call gui::get() the data specification object that is returned always uses a normalized form for the buttons object type using buttonAttr objects with the enabled property set.

Example:

```
{
   myCard :
   {
      ...
      buttons :
      {
         start : { enabled : false },
         pause : { enabled : false },
         stop  : { enabled : false },
         ...
      }
      ...
   }
   ...
}
```

Getting Component Information

During normal use of the GUI, it may be necessary to look up the current state of the GUI. This may be to figure out the fields that are included in the layout or testing whether a field is read-only or not. The gui::getComponentInfo function facilitates this. This function returns an object that closely resembles the specification object that is used in gui::open.

The gui::getComponentInfo function can be passed the name of a component or no parameters at all. In the first case, only information for that component is returned—otherwise, information about all components is returned.

Example:

```
   // Get info on one component
var myInfo = gui::getComponentInfo("myComponent");
```

```
   // Get all information
var allInfo = gui::getComponentInfo();
```

The component information contains one property for each component, and as usual the property names correspond to the component names. This also holds true if only one component is queried.

Information Object Specification

```
1  <componentName> : object
2    kind : string
3    buttons : object
4      info : object
5        <buttonName> : object
6          kind : string
7          enabled : bool
8          script : string
9          xsize : int
10          ysize : int
11    fields : object
12      index : array of strings
13      info : object
14        <fieldName> : object
15          index : int
16          kind : string
17          type : string
18          mandatory : bool
19          zeroSuppression : bool
20          enabled : bool
21          open : bool
22          focusable : bool
```

Information Object Description

1. <componentName> contains the component information for the component that has the same name as the property.
2. kind specifies the component kind. The possible values can be found in “Component Specification.”
3. buttons holds information about the buttons that are associated with the component.
4. info holds the actual button information.
5. <buttonName> contains the information about the button that has the same name as the property.
6. kind specifies the button kind. The possible values can be found in “Button Specifications.”
7. enabled identifies whether or not the button is enabled.
8. script is the name of the script that is specified for an “open” button.
9. xsize is the window x size specified for an “open” button.
10. ysize is the window y size specified for an “open” button.

11. fields contains the field information.
12. index is an array of field names. The index in this array corresponds to the layout position of the field.
13. info contains the set of all fields.
14. <fieldName> contains information about the field that has the same name as the property.
15. index holds the layout position of the field.
16. kind holds the field kind as specified in “Field Specifications.”
17. type holds the field type as specified in “Field Specifications.”
18. Mandatory indicates whether or not the field is mandatory.
19. zeroSuppression indicates whether or not the field has zero suppression enabled.
20. enabled indicates whether or not the field is currently enabled. The enabled information corresponds to the current enabling information set in the GUI data.
21. open indicates whether or not the field is open in the layout. This value overrides the enabling information.
22. focusable indicates whether or not focus can be set to the field. A field must be both enabled and open to be focusable. In addition, the component itself must be enabled.

Event Description

As mentioned previously, any system that uses the GUI API should include an event handler much like the following example.

```
var g;
if ((g=gui::get()) != null)
{
    switch (g.action.kind)
    {
        case "enter": ... case "button": ...
    }
}
```

This retrieves the event using gui::get() and then branches on the actual event as defined by the action kind. Previous sections have not discussed the possible event kinds, which are described in the following sections. The first section goes through the properties that are available independently of the action kind, and the following sections go through each of the possible events.

Common Event Data

Independently of what kind of event an application receives, the return value from gui::get() always contains the following properties.

Name	Type	Description
action	object	Event specification. Always contains a kind property and some other properties, depending on the event kind.
focus	object|null	Specifies exactly where the current focus is. See “Focus Specification” for further information. This property is null if no element has focus.
data	object	Holds all of the window data (field values and current element enabling). See “Dialog Data” for further information.
selectedRows	array	An array of Booleans that indicate which of the rows in a table the user may have selected (if the “select row” row action is enabled). The array contains one value for each of the data rows (a true value indicates that the row is selected). The selectedRows value reflects the state of the current focused component.
parent	object	This object contains two properties: data and userValue, which are a copy of the parent window’s data object, and a read/write reference to the parent window’s user data. The parent property is only available for certain events.
userValue	any	The value that was originally passed to gui::open(). If this value is an object or an array, it is possible to modify its properties or elements respectively.

Name	Type	Description
composerData	any	The composer data returned by getSpecification() from a composer package. This property exists only in event data that is handled by an event handler within the composer package. If the composer package did not specify any composer data, this property is null.

Focus Specification

The focus object contains the following properties.

Name	Type	Description
component	string	Name of the component that has focus.
field	string	Name of the field that has focus.
rowIndex	int	Optional row index that identifies the row that has focus. This property is always zero for card components.
dataChanged	bool	Identifies whether or not data in the current record has been changed by the user. This flag is set if, for instance, the user changes anything in a row and then clicks in another row (which generates a setFocus event).

The input focus can be in one place, while the reason for the event can be located elsewhere.

Examples:

- The focus can be in a field in one component, while the reason for the event could be that a button in another component is clicked.
- The focus can be in a field in a table row, while the reason for the event is that a user clicked the “Delete row” button on another row in the same table component.

In both cases the focus specification object reflects where the focus actually is, while the action object property of gui::get()’s return value tells where the button is located.

Dialog Data

The data object is a copy of the internal data that is used by the GUI to store the current user input. The format of the data object is exactly as the initial data supplied to gui::open() as described in “Data Specification.”

The button Event

button Event			
Name	button		
Data	**Name**	**Type**	**Description**

button Event			
	button	string	Name of the activated button.
	component	string	Name of the component in which the button belongs.
	rowIndex	int	Position of inline button in a table. This value is always zero for card components.
Description	This event is received whenever a user clicks a button or selects a menu entry.		
Remarks	It is not possible to distinguish events that are generated by the activation of a button from events that are generated by the activation of a menu entry.		

Example	`case "button":`
	`  switch (action.button)`
	`  {`
	`  case "instructions":`
	`  gui::message(instructions, 300, 600);`
	`  break;`
	`  }`
	`break;`		

The childNotification Event

childNotification Event			
Name	childNotification		
Data	**Name**	**Type**	**Description**
	event	string	Name of the event that was sent from a child window.
	value	any	The data that is associated with the event. This property is optional and depends on the event that is sent.
Description	This event is received when a child window has used gui::notifyParent() to send an event to its parent window.		

childNotification Event	

Example	Child window code:
	`gui::notifyParent("MyEvent", "MyData");`
	Parent window code:
	`case "childNotification":`
	`{`
	`  var data = g.data.card.rows[0];`
	`  data.a.value = "Got event '" + g.action.event`
	`  + "' with data '" + g.action.value`
	`  + "' from child";`
	`  gui::update("card", data);`
	`}`
	`break;`

The deleteRow Event

deleteRow Event			
Name	deleteRow		
Data	**Name**	**Type**	**Description**
	component	string	Name of the component.
	rowIndex	int	Index of the row that should be deleted.
Description	This event is received when a user requests the deletion of a row in a table component. This is done when a user clicks on the "Delete row" icon, but future versions may also generate this event from a keyboard shortcut. It is possible to get a deleteRow event (for the current row) without validation of the current row. It is, for instance, possible to insert a row and then delete it without filling out all of the mandatory fields; however, in the same way it is also possible to enter an invalid date or similar in an existing row and then delete it. The row index can be sent directly to gui::deleteRow().		

Example	`case "deleteRow":`
	`  if (…delete is okay…)`
	`  gui::deleteRow(action.component, action.rowIndex);`
	`break;`		

The enter Event

enter Event			
Name	enter		
Data	**Name**	**Type**	**Description**

enter Event			
	component	string	Focus information — the name of the current component.
	field	string	Focus information — the name of the current field.
	rowIndex	int	Focus information — the index of the current row (zero for cards).
Description	This event is received whenever the user presses the Enter key.		

Example	`var g = gui::get();`
	`…`
	`case "enter":`
	`{`
	`var record = g.data.myCard.rows[0];`
	`gui::searchTransfer({`
	`field_dl: record.field_dl,`
	`field_tl: record.field_tl,`
	`field_sl: record.field_sl,`
	`field_pl: record.field_pl`
	`});`
	`gui::close();`
	`}`
	`break;`		

The fieldChanged Event

fieldChanged Event			
Name	fieldChanged		
Data	**Name**	**Type**	**Description**
	component	string	Name of changed component.
	field	string	Name of changed field.
	rowIndex	int	Index of changed row.
	fieldValue	any	Value of the changed field.
Description	This event is received when a user changes the value of a pop-up field that has been enabled for the fieldClicked event. See “Field-Specific Events” for information about enabling a field for this event.		

fieldChanged Event	

Example	`var g = gui::get();`
	`…`
	`case "fieldChanged":`
	`  {`
	`  // Show field information`
	`  gui::alert( g.action.kind + ": "`
	`  + g.action.component + ","`
	`  + g.action.field + ","`
	`  + string(g.action.rowIndex));`
	`  }`
	`break;`

The fieldChangedViaAutofetch Event

fieldChangedViaAutofetch Even			
Name	fieldChangedViaAutofetch		
Data	**Name**	**Type**	**Description**
	component	string	Name of changed component.
	field	string	Name of changed field.
	rowIndex	int	Index of changed row.
	fieldValue	any	Value of the changed field.
Description	This event is received whenever literals for a dynamic favorite requests to be updated.		

Example	`var g = gui::get();`
	`…`
	`case "fieldChanged":`
	`  {`
	`  // Show field information`
	`  gui::alert( g.action.kind + ": "`
	`  + g.action.component + ","`
	`  + g.action.field + ","`
	`  + string(g.action.rowIndex));`
	`  }`
	`break;`		

The fieldClicked Event

fieldClicked Event			
Name	fieldClicked		
Data	**Name**	**Type**	**Description**
	component	string	Name of clicked component.
	field	string	Name of clicked field.
	rowIndex	int	Index of clicked row.
Description	This event is received when a user clicks on a raw or image field that has been enabled for the fieldClicked event. See “Field-Specific Events” for more information about enabling a field for this event.		

Example	`var g = gui::get();`
	`…`
	`case "fieldClicked":`
	`  {`
	`  // Show click information`
	`  gui::alert( g.action.kind + ":"`
	`  + g.action.com;ponent + ","`
	`  + g.action.field + ","`
	`  + string(g.action.rowIndex));`
	`  }`
	`break;`		

The gotFocus Event

gotFocus Event			
Name	gotFocus		
Data	**Name**	**Type**	**Description**
	component	string	Name of changed component.
Description	This event is received when a user changes focus from one record (row or component) to another. This event must be enabled as described in “Field-Specific Events.” The Field name and row index can be found in the focus information.		

gotFocus Event	

Example	`var g = gui::get();`
	`…`
	`case "gotFocus":`
	`{`
	`// Show focus information`
	`gui::alert( g.action.kind + ": "`
	`+ g.focus.component + ","`
	`+ g.focus.field + ","`
	`+ stringf(g.focus.rowIndex));`
	`}`
	`break;`

The grandTotalClicked Event

grandTotalClicked Event			
Name	grandTotalClicked		
Data	**Name**	**Type**	**Description**
	component	string	Name of clicked component.
	field	string	Name of clicked field.
Description	This event is received when a user clicks on a grand total that has been enabled for the grandTotalClicked event. See “Field-Specific Events” for information about enabling a field for this event.		

Example	`var g = gui::get();`
	`…`
	`case "grandTotalClicked":`
	`{`
	`// Show click information`
	`gui::alert( g.action.kind + ": "`
	`+ g.action.component + ","`
	`+ g.action.field);`
	`}`
	`break;`		

The headingClicked Event

headingClicked Event			
Name	headingClicked		
Data	**Name**	**Type**	**Description**

headingClicked Event			
	component	string	Name of clicked component.
	field	string	Name of clicked field.
Description	This event is received when a user clicks on a heading that has been enabled for the headingClicked event. See “Field-Specific Events” for information about enabling a field for this event.		

Example	`var g = gui::get();`
	`…`
	`case "headingClicked":`
	`{`
	`// Show click information`
	`gui::alert( g.action.kind + ":"`
	`+ g.action.component + ","`
	`+ g.action.field);`
	`}`
	`break;`		

The initialized Event

initialized Event			
Name	initialized		
Data	none		
Description	This event is received right after a call to gui::open() and it indicates that the GUI framework is up and running. This event is received before the initial data is sent to the rendering device. The typical use of this event is to place the cursor correctly using gui::setFocus().		

Example	`case "initialized":`
	`gui::setFocus("myTable", "firstField",0);`
	`break;`		

The insertRow Event

insertRow Event			
Name	insertRow		
Data	**Name**	**Type**	**Description**
	component	string	Name of the component.

insertRow Event			
	rowIndex	int	Index of the row that should be inserted.
Description	This event is received when a user requests the insertion of a new row into a table component. This is done when a user clicks on the “insert row” icon, but future versions may also generate this event from a keyboard shortcut. The row index can be sent directly to gui::insertRow().		
Remarks	The gui::acceptData() function can be used to tell the GUI framework whether or not the user is allowed to change focus from a newly created row without generating a setFocus event, and thereby giving the program a chance to validate the new input.		

Example	`case "insertRow":`
	`{`
	`  var newRecord = { field_1 : null,`
	`  field_2 : null`
	`  };`
	`  gui::insertRow(action.component, newRecord, action.rowIndex);`
	`  gui::setFocus(action.component, "field_1", action.rowIndex);`
	`  doAccept = false;`
	`}`
	`break;`
	
	`if (doAccept)`
	`  gui::acceptData();`		

The searchInitialized Event

searchInitialized Event			
Name	searchInitialized		
Data	**Name**	**Type**	**Description**
	name	string	Name of the search (as specified in the component specification.
	title	string	Title of the search (as specified in the component specification).
Description	This event is received when a search component has been opened for the first time, much like the initialized event.		
Remarks	The GUI framework is not available when this event is received, so a call to gui::open() must be made to create a GUI search window.		

searchInitialized Event	

Example	`case "searchInitialized":`
	`  gui::open("My Search Window",`
	`  "MyScript.ms",`
	`  specification, layout, data, () );`
	`break;`

The setFocus Event

setFocus Event			
Name	setFocus		
Data	**Name**	**Type**	**Description**
	component	string	Name of the requested component.
	field	string	Name of the requested field.
	rowIndex	int	Position of the field in a table. This value is always zero for card components.
Description	This event is received when a user has changed some data in a record and then places the cursor in another record (another row or component). What happens is the following: 1. The user changes some data and then uses either the keyboard or the mouse to move into another record. 2. The GUI detects this and reverts the focus change, because it must be validated by the programmer. 3. The GUI validates the input. If it is correct, the focus change is ignored, and the error is presented to the user. 4. The GUI sends the setFocus event to M–Script when the input is validated. 5. The M-Script program receives the event and must decide whether or not to allow the focus change (this may depend on the new user input). 6. If the focus change is accepted, the M-Script program must call gui::setFocus() to allow it. In addition, gui::acceptData() should be called to indicate that the data is valid. 7. If the focus change is rejected, nothing needs to be done, although an alert stating the reason would be appropriate.		
Remarks	Normally a user is allowed to change focus freely as long as data is left unchanged, but only if gui::acceptData() is called when the data is valid from the application's point of view. If gui::acceptData() is not called, but gui::setFocus() only is used to accept a focus change, the system gets into a state where a setFocus event is generated every time that the user changes focus, even if data is left unchanged.		

setFocus Event	

Example	`var g = gui::get();`
	`…`
	`case "setFocus":`
	`  gui::setFocus(g.action.component,`
	`  g.action.field,`
	`  g.action.rowIndex);`
	`break;`

The subWindowInitialized Event

subWindowInitialized Event	
Name	subWindowInitialized
Data	None
Description	This event is received when a subwindow, opened by an “open” button, is opened, much like the initialized event.
Remarks	The GUI framework is not available when this event is received, so a call to gui::open() must be made to create a GUI window.

Example	`case " subWindowInitialized ":`
	`  gui::open("My Search Window",`
	`  "MyScript.ms",`
	`  specification, layout, data, { } );`
	`break;`

Debugging

You can debug the GUI-II framework in various ways. The following possibilities exist:

- Set the gui_debug = true setting in the initialization file to reveal a debug output frame in the lower part of the window. Output from print statements is visible here.
- Enable GUI logging by setting log = gui in the initialization file. This dumps GUI-specific information to the log file.
- Enable query variable logging by setting log = query in the initialization file. This makes M-Script dump all incoming query values to the log file.
- Set debugDumpFilename in the initialization file to a file to which all output that is generated by M-Script is dumped.

Subroutine Reference

This section provides a complete, alphabetically ordered reference description of all subroutines in the M-Script GUI-II API.

Each subroutine is described in a uniform way in its own section. The first section is a sample entry that illustrates how to read a subroutine reference entry. Please read this example first to get the most out of this section.

Sample Entry

Prototype

Every subroutine is described by its prototype. The prototype tells:

- Whether the subroutine is a procedure (that is, a subroutine that has no return value) or a function (that is, a subroutine that has a return value)
- What the name of the subroutine is
- Which parameters the subroutine takes

A prototype for a function that takes one parameter could look like the following:

```
procedure gui::deleteRow(componentName, rowIndex)
```

Sometimes a subroutine has a few mandatory parameters but accepts an unlimited number of parameters. In this case "..." is used to denote zero or more parameters. A prototype for such a function could look like the following:

```
function maconomy::sql(statement, maxRecs, firstRec, ...)
```

This function takes at least three parameters, but there is no upper limit to the number of parameters that are passed.

If a subroutine can be called in more than one way, more than one prototype is given.

Description

A brief description of the subroutine is given.

Parameters

All subroutine parameters are described. For each parameter, its name, type, and a description of its meaning are given. The type "simple" is used for a parameter that can be of any simple type—that is, not object or array. If either of two types can be used, "|" is used to separate the types.

A parameter description could look like the following.

Parameter Descriptions

Name	Type	Description
statement	string	Any SQL select statement to be executed on the Maconomy server. This statement may contain placeholders ^1, ^2, …. These placeholders are replaced by the arguments that follow firstRec. The placeholder ^1 is replaced by the first argument that follows firstRec, ^2 is replaced by the second argument that follows firstRec, and so on.

Parameter Descriptions		
maxRecs	int | null	The maximum number of records that the function call should return. If this parameter is omitted or null is passed, all records are returned.
firstRec	int | null	The number of the first record to return. The Index of the first record is zero. If this parameter is omitted or null is passed, the function call returns records starting from the first available record.
…	simple	Placeholder strings ^1, ^2, … in statement are replaced by the corresponding remaining parameters. The placeholder ^1 is replaced by the first of these arguments, ^2 by the second, and so on.

Return Value

If the subroutine is a function, the return value is described here. If the return value is a complex object value, a reference is given to a description of the object type.

Throws

Many subroutines throw exceptions if errors occur. These are listed in this subsection.

Exceptions that are not of the type error::stdlib::gui are not listed here This could, for instance, be exceptions that are thrown because a wrong number of parameters is passed to a procedure or because a connection to a Maconomy server is not established.

Remarks

This section gives information on how to use the subroutine. It also informs you about common pitfalls and special considerations that should be taken.

Example

Finally, a small code example of how to use the function is given. An example could look like the following:

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...
        case "enter":
            ...
            gui::update("myCard", ...);
            gui::acceptData();
            break;
        ...
    }
}
```

gui::acceptData

Prototype

```
procedure gui::acceptData()
```

Description

Tells the GUI that the user is allowed to change focus away from the current record.

Parameters

None

Throws

Nothing

Remarks

When data in a record—such as a card or a table row—has changed the user is not allowed to move focus to another record until gui::acceptData() has been called.

Note that a record can be changed in two ways: by the user and programmatically by calling gui::update(), gui::setRows(), or gui::insertRow().

The idea of this procedure is to give the M-Script programmer a way to disallow focus change:

- If data entered by the user cannot be accepted.
- If data sent to the GUI from the M-Script program must be altered by the user and resubmitted.
- If a table row is created empty and must be filled in by the user.

In these situations the procedure should not be called.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...
        case enter:
            ...
            gui::update(mycard, ...);
            gui::acceptData();
            break;
        ...
    }
}
```

gui::addDependency

Prototype

```
procedure gui::addDepenendency(source)
```

```
procedure gui::addDependency(depender, source)
```

Description

The gui::addDependency() function is used to specify dependencies between various components or packages. In the first form it specifies a dependency on the source component/package in the current composer package. This form can only be used inside a getSpecification() call in a composer package. In the second form it specifies a dependency on the source component/package in the depender package.

Parameters Descriptions		
Name	**Type**	**Description**
source	string	Name of either component or package that depender depends on.
depender	string	Name of package that should receive the dependencyUpdate events that are emitted when the source is updated.

Throws

```
error::stdlib::gui
```

Example

```
Public function getSpecification(userValue)
{
    ... gui::addDependency(expenseSheetTable);
}
```

gui::alert

Prototype

```
procedure gui::alert(message)
```

Description

Displays an alert box with an “OK” button on the target device. The user must click “OK” to dismiss the message.

This procedure may be used even outside of the GUI framework. This makes it possible to use alert in situations where the GUI is not completely initialized, or for some other reason not even in use.

Parameter Descriptions		
Name	**Type**	**Description**
message	string	The message to display in the alert box.

Throws

```
error::stdlib::gui
```

Remarks

This procedure can be called multiple times in the same script invocation, causing the alerts to appear one after another.

Example

```
gui::alert("Discount recalculated.");
```

gui::close

Prototype

```
procedure gui::close()
```

Description

Closes the GUI window or replaces its contents.

Name	Type	Description
url	string	The URL of the resource to be displayed in the window. Passing the empty string causes the window to be closed.
target	string	The name of the HTML frame to open the URL in. The values of target can be “_self,” “_parent,” or “_top.” These frame names refer to the current frame (_self), the parent frame (_parent), or the top frame (the main window).
options	object	An object with additional settings for the close call. If the url or target parameters have not been passed as function parameters, these may be set in the options object with the property names url and target. An additional keepSession property can be set to true if the session ID is to be passed on to the next window (the default value is true).

Throws

```
error::stdlib::gui
```

Remarks

Calling the procedure with no parameters simply closes the GUI window.

Using the other prototypes causes the window content to be replaced with the resource that is identified by the url parameter. Passing the empty string for the url parameter causes the window to be closed.

If the keepSession property is true, the session ID is passed on to the page that is referred to by the URL. This is necessary if the URL points to an M-Script program that needs to operate on the same session that the GUI window did.

Example

```
// Replace window content with Maconomy's web page.
gui::("http://www.maconomy.com", {keepSession : false});
```

gui::deleteRow

Prototype

```
procedure gui::deleteRow(componentName, rowIndex)
```

Description

Deletes a row in a table component.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the table component in which a row should be deleted.
rowIndex	int	The index of the row to delete. The first row has index 0.

Throws

```
error::stdlib::gui
```

Remarks

The component name that is passed to this procedure must be the name of a table component. The row index passed must correspond to the index of an existing row.

The typical use of this procedure is to handle a deleteRow event from the GUI. This event is sent when a user clicks the “delete row” button beside a table row. The row is not deleted until the M-Script program calls gui::deleteRow().

The procedure can only be used on table components.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...

        case "deleteRow":
            gui::deleteRow(g.action.component, g.action.rowIndex);
            break;

        ...
    }
}
```

gui::disableComponent

Prototype

```
procedure gui::disableComponent(componentName)
```

```
procedure gui::disableComponent(componentName, setInlineButtons)
procedure gui::disableComponent(componentName, setInlineButtons, setRawImage)
```

Description

Disables all input elements and optionally also all inline buttons in a GUI component.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component to disable.
setInlineButtons	bool	The inline buttons in the component are also disabled iff this parameter is true. If this parameter is left out, it is considered to be false.
setRawImage	bool	The clickable raw fields and images in the component are also disabled iff this parameter is true. If this parameter is left out, it is considered to be false.

Throws

```
error::stdlib::gui
```

Remarks

Disabling a component using this function does not throw away information about open and closed fields. When the component is re-enabled by calling gui::enableComponent(), the input fields and inline buttons that were previously defined as closed are still closed.

To disable or enable individual fields and buttons see the entries for gui::setFieldAccess() and gui::setButtonAccess().

The procedure cannot be used on composer components.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...
        case "button":
            if (g.action.button == "disable")
                gui::disableComponent("myCard");
            else if (g.action.button == "disableAll")
                gui::disableComponent("myCard", true);
            break;
        ...
    }
}
```

gui::enableComponent

Prototype

```
procedure gui::enableComponent(componentName)
procedure gui::enableComponent(componentName, setInlineButtons)
```

Description

Enables all input elements and optionally also all inline buttons in a GUI component.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component to enable.
setInlineButtons	bool	The inline buttons in the component are also enabled iff this parameter is true. If this parameter is left out, it is considered to be false.

Throws

```
error::stdlib::gui
```

Remarks

When a component is re-enabled using this function after being disabled by a call to gui::disableComponent(), the input fields and inline buttons that were already closed when the component was disabled are still closed after re-enabling.

To disable or enable individual fields and buttons, see the entries for gui::setFieldAccess() and gui::setButtonAcfcess().

The procedure cannot be used on composer components.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...

        case "button":
            if (g.action.button == "enable")
                gui::enableComponent("myCard");
            else if (g.action.button == "enableAll")
                gui::enableComponent("myCard", true);
            break;

        ...
    }
}
```

gui::get

Prototype

```
function gui::get()
```

Description

This function returns information about the active GUI window.

Parameter Descriptions

None

Return Value

This function returns an object (described in “Event Description”) or null if no GUI window is active.

Throws

```
error::stdlib::gui
```

Remarks

This function returns information about the active GUI window. This information includes information about:

- The event that caused M-Script to be called.
- The component that currently has the input focus.
- The current data in the window.
- The value passed for the userValue parameter when gui::open() was called.

If no GUI window is active, the function returns null.

If the window data for the active GUI window has been removed from the session or if the same data is submitted twice (this can happen by using the “Back” button in the browser), this function throws an exception.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...
    }
}
```

gui::getComponentInfo

Prototype

```
function gui::getComponentInfo()
function gui::getComponentInfo(component)
```

Description

Returns information about the components used in the current window.

Parameter Descriptions

None—which means all components—or the name of a component to be investigated.

Return Value

This function returns an object as described in “Getting Component Information.” The data that is returned mirrors the current state of the GUI after the specification, layout, and data have been merged (which influences, among other things, read-only information).

Throws

error::stdlib::gui if an unknown component is requested.

gui::getDeviceInfo

Prototype

```
function gui::getDeviceInfo()
```

Description

Returns information about the current target device.

Parameter Descriptions

None

Return Value

This function returns an object of type deviceInfo (see “Getting Device Information”), giving information about the browser make and version, the operating system name and version, and target language (for example, HTML).

Throws

Nothing

Remarks

This function can be used to detect which target device (browser) a user is using if some hand-coded HTML should differ depending on the target device.

Example

```
// Detect browser.
var deviceInfo = gui::getDeviceInfo();

switch (deviceInfo.browser.name)
{
   case "msie":
      ...
      break;
   ...
}
```

gui::highlightField

Prototype

```
procedure gui::highlightField(componentName, field)
procedure gui::highlightField(componentName, field, rowIndex)
```

Description

Highlights a specific field by setting its CSS class name to highlight. The rendering of the highlighted field can be changed by adding a custom style sheet URL to the component (see “Getting Device Information”).

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component to highlight a field in.
field	string	The name of the field to highlight.
rowIndex	int	The row in which the field should be highlighted. This value is ignored for cards.

Throws

```
error::stdlib::gui
```

Example

```
if (g != null)
{
    switch(g.action.kind)
    {
        ...
        case "fieldClicked":
            gui::highlightField(g.action.component,
                                g.action.field,
                                g.action.rowIndex);
        break;
    }
}
```

gui::highlightGrandTotal

Prototype

```
procedure gui::highlightGrandTotal(componentName, field)
```

Description

Highlights a specific grand total by setting its CSS class name to highlight. The rendering of the highlighted grand total can be changed by adding a custom style sheet URL to the component (see “Common Layout Features”).

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component to highlight a field in.
field	string	The name of the field to highlight.

Throws

```
error::stdlib::gui
```

Example

```
if (g != null)
{
    switch(g.action.kind)
    {
        ...
        case "grandTotalClicked":
           gui::highlightGrandTotal(g.action.component,
                                    g.action.field);
        break;

    }
}
```

gui::highlightHeading

Prototype

```
procedure gui::highlightHeading(componentName, field)
```

Description

Highlights a specific heading by setting its CSS class name to highlight. The rendering of the highlighted heading can be changed by adding a custom style sheet URL to the component (see “Common Layout Features”).

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component to highlight a heading in.
field	string	The name of the field heading to highlight.

Example

```
if (g != null)
{
    switch(g.action.kind)
    {
```

```
        ...
        case "headingClicked":
           gui::highlightHeading (g.action.component,
                                  g.action.field);
        break;
    }
        }
```

gui::insertRow

Prototype

```
procedure gui::insertRow(componentName, record)
procedure gui::insertRow(componentName, record, rowIndex)
```

Description

Inserts a new row into a table component.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the table component into which the new row should be inserted.
Record	object	Data for the new table row. This parameter is an object of the type record (see “Record”).
rowIndex	int	The index of the new row. This number must be in the range [0;n] where n is the number of rows in the table before the new row is inserted. If this parameter is omitted, it is considered to have the value n.

Throws

```
error::stdlib::gui
```

Remarks

The row is inserted before the row that has index rowIndex before the insertion occurs. To insert a row in the beginning of the table, pass 0 as the rowIndex parameter. To insert a row at the end of the table, simply omit the rowIndex parameter.

However, the normal use is to call this procedure in response to an insertRow event (see “insertRowEvent”), in which case the natural thing to do is to pass the rowIndex from the action property in the result from gui::get(). This inserts the row at the position that the user expects.

This procedure can only be used on table components.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
```

```
    {
        ...
        case "insertRow":
            {
                var a = g.action;
                gui::insertRow(a.component, ..., a.rowIndex);
            }
            break;
        ...
    }
}
```

gui::message

Prototype

```
procedure gui::message(message)
procedure gui::message(message, width, height)
```

Description

Opens a new window and displays a message.

Parameter Descriptions		
Name	**Type**	**Description**
message	string	The message to display to the user. This string is allowed to contain formatting codes that are specific to the target device, for example, HTML tags.
width	int	The width in pixels of the new window. If this parameter is omitted, the window is given a default width.
height	int	The height in pixels of the new window. If this parameter is omitted, the window is given a default height.

Throws

```
error::stdlib::gui
```

Remarks

This procedure opens a new window, and the string that is supplied in the message parameter is inserted into this empty window. This string is allowed to contain formatting codes—for example, HTML tags in case of an HTML target device. The title of the window is “Message.”

Call gui::getDeviceInfo() to get information about the target device.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
```

```
    {
        ...
        case "button":
            switch (g.action.button)
            {
                case "instructions":
                   {
                      var instructions = ...;
                      gui::message(instructions);
                   }
                   break;
                ...
            }
            break;
        ...
    }
}
```

gui::newWindow

Prototype

```
procedure gui::newWindow(url)

procedure gui::newWindow(url, windowProperties)
```

Description

Opens a new window.

Parameter Descriptions		
Name	**Type**	**Description**
url	string	The URL of the resource to be displayed in the window.
windowProperties	object	Various properties for the window (see “Remarks”).

Throws

```
error::stdlib::gui
```

Remarks

If the contents of the new window are generated by an M-Script program, and this script should use the same session as the existing GUI window, the URL that is passed to this function should be generated using the link() function.

The possible window properties are described in the following table.

Name	Type	Description
width	int	The window width in pixels.

Name	Type	Description
height	int	The window height in pixels.
left	int	The left window position in pixels.
top	int	The top window position in pixels.
toolbar	bool	Show the browser-specific toolbar in the new window (the default is true).
location	bool	Show the browser-specific URL input field (the default is true).
resizable	bool	Allow the window to be resized by the user.
status	bool	Show the browser status bar (the default is true).
scrollbars	bool	Show window scrollbars (the default is true).
menubar	bool	Show the browser menu (the default is true).

The actual property names are defined by the feature names for the JavaScript function window.open. The preceding list may not be complete, so refer to a JavaScript reference to get all possible attributes. The values are transferred directly to JavaScript without any checking.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...

            case "button":
                switch (g.action.button)
                {
                    case "maconomy":
                      gui::newWindow("http://www.maconomy.com");
                      break;

                ...
            }
            break;

        ...
    }
}
```

gui::notifyParent

Prototype

```
procedure gui::notifyParent(eventName)
procedure gui::notifyParent(eventName, eventValue)
```

Description

Sends a childNotification event to the parent window.

Parameter Descriptions		
Name	**Type**	**Description**
eventName	string	The name of the event to send to the parent window.
eventValue	any	A value that accompanies the event.

Remarks

This procedure sends a childNotification event to the parent window—that is, the window that opened this window as either a subwindow or a search window.

Along with the childNotification event, a name (eventName) and an optional arbitrary value (eventValue) are sent.

Example

In the Child Window Handler

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...

        case "button":
            switch (g.action.button)
            {
                case "transfer":
                   gui::notifyParent("transfer", 42);
                   break;

                ...
            }
            break;

        ...
    }
}
```

In the Parent Window Handler

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...

        case "childNotification":
            {
                var a = g.action;
                gui::alert( "Received event '" + a.event
                            + "' with value " + string(a.value) + ".");
            }
            break;

        ...
    }
}
```

gui::open

Prototype

```
procedure gui::open(title, script, spec, layout, data, userValue)
procedure gui::open(package, userValue)
```

Description

Initializes a new GUI window.

Parameter Description		
Name	**Type**	**Description**
title	string	The title of the new window.
script	string	The script that is going to handle events that are generated by the window. If this parameter is the empty string, the script that calls gui::open() will be used.
spec	object	An object that specifies the window's components (see "Component Specification").
layout	object	The layout of the window (see "Layout Specification").
data	object	The initial data for the window (see "Data Specification").
userValue	any	Any value that the programmer wants to associate with the window

Parameter Description		
package	string	The name of a package to use as a composer package at the top level. In this way, title, spec, layout, and data are not required, since these are delivered by the composer package (see “Composer Packages”).

Throws

```
error::stdlib::gui
error::stdlib::gui::unsupportedTarget
```

Remarks

This procedure is used to initialize a GUI window. Consequently, this is the first procedure called when using the GUI API.

The value that is passed to userValue is accessible via the return value from gui::get(). If this value is an object or an array it is possible to modify its properties or elements, respectively, by modifying the value returned from gui::get().

Example

```
var spec      = {...};
var layout    = {...};
var data      = {...};
var userValue = {...};
gui::open("Time Sheet", "tsHandler.ms",
          spec, layout, data, userValue);
```

gui::rawClickableHTML

Prototype

```
function gui::rawClickableHTML(component, id, html)
```

Description

Makes any HTML clickable, which means that the GUI generates a rawClicked event when a user clicks on the HTML.

Clickable HTML is for use in raw elements in both card and array components.

Parameter Descriptions		
Name	**Type**	**Description**
component	string	The name of an existing component. This component receives the rawClicked event.
id	string	Any string that is used to identify the clickable area. Note that neither field name nor row index is supplied with the rawClicked event.
html	string	The HTML that should be made clickable.

Example

```
var html = "<span class=\"red\">RED click</span>";
```

```
html = gui::rawClickableHTML("myComponent", "RED", html);
gui::update("myComponent", { raw: {value:html} });
```

gui::searchTransfer

Prototype

```
procedure gui::searchTransfer(record)
```

Description

Transfers values from a search window to the parent window.

Parameter Descriptions		
Name	**Type**	**Description**
record	object	The record to transfer to the parent window. This is an object of the type record.

Throws

```
error::stdlib::gui
```

Remarks

This procedure transfers values from a search window to the parent window from which the search window was opened. The values in record are transferred to the component (in the parent window) from which the search was initiated. It is allowed to pass only a subset of the fields in the record (as opposed to the update functions).

If any of the fields in the parent window is closed, no data is transferred at all (to ensure integrity of the transfer). It is not possible to detect this situation.

Example

```
var g = gui::get();
if (g != null)
{
   switch(g.action.kind)
   {
      ...

      case "button":
         switch (g.action.button)
         {
            case "transferResult":
               {
                  var record = {...};
                  gui::searchTransfer(record);
               }
               break;

            ...
```

```
            }
            break;

        ...
    }
}
```

gui::setButtonAccess

Prototype

```
procedure gui::setButtonAccess(componentName, buttonAccess)
```

Description

Enables and disables buttons in a component.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component whose buttons' access should be altered
buttonAccess	object	A specification of which buttons should be enabled and disabled. This is an object of the type buttons.

Throws

```
error::stdlib::gui
```

Remarks

This procedure enables and disables buttons in a specified component. Only the buttons whose access should be changed need be listed in the buttonAccess object.

Please note that this procedure cannot be used to enable and disable inline buttons. Use gui::setFieldAccess() instead.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...

        case "button":
            switch (g.action.button)
            {
                case "runAction":
                    ...
                    gui::setButtonAccess("myCard",
                                         {runAction : false} );
```

```
                break;

            ...
        }
        break;

    ...
  }
}
```

gui::setFieldAccess

Prototype

```
procedure gui::setFieldAccess(componentName, fieldAccess)
```

Description

Enables and disables fields—including inline buttons—in a component.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component whose fields' access should be altered.
fieldAccess	object	A specification of which fields should be enabled and disabled. This is an object of the type fields.

Throws

```
error::stdlib::gui
```

Remarks

This procedure enables and disables fields in a specified component. Only the fields whose access should be changed need be listed in the fieldAccess object.

Access to inline buttons is modified by using this procedure, not by using gui::setButtonAccess().

This procedure cannot be used on composer components.

Example

```
var g = gui::get();
if (g != null)
{
   switch(g.action.kind)
   {
      ...

      case "button":
```

```
            switch (g.action.button)
            {
                case "runAction":
                   ...
                   gui::setFieldAccess("myCard",
                                       {weekNo : false} );
                   break;

                ...
            }
            break;

        ...
    }
}
```

gui::setFocus

Prototype

```
procedure gui::setFocus(componentName)
procedure gui::setFocus(componentName, fieldName)
procedure gui::setFocus(componentName, fieldName, rowIndex) procedure
gui::setFocus(componentName, fieldName, {openDropDown: true})
procedure gui::setFocus/componentName, fieldName, rowIndex,
{openDropDown: true})
```

Description

Moves the input focus to a specified field in a specified component. If the object {openDropDown: true} is set, the drop-down opens after it receives focus. This can only be set for fields with dynamic favorites.

Parameter Descriptions

Name	Type	Description
componentName	string	The name of the component to which the input focus should be moved.
fieldName	string	The name of the field to which the input focus should be moved. If fieldName is null or left out, focus is placed in the first focusable field. If no focusable field exists, focus is left unchanged.
rowIndex	int | null	The index of the table row to which the input focus should be moved.

Throws

```
error::stdlib::gui
```

Remarks

The rowIndex parameter must be of type int and cannot be omitted if componentName is the name of a table component. Likewise, the parameter must not be of type int if the referred component is a card component.

The typical use of this procedure is to call it as the response to a setFocus event. To make things easier, the third parameter can always be passed—also for card components—with a value of null. This matches the action.rowIndex property in the return value from gui::get(), which is always present for a setFocus event but with null as value for card components.

This procedure cannot be used on composer components.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...
        case "setFocus":
            {
                var a = g.action;
                gui::setFocus(a.component, a.field, a.rowIndex);
            }
            break;
        ...
    }
}
```

gui::setRows

Prototype

```
procedure gui::setRows(componentName, records)
```

Description

Replaces the contents of a table component.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the table component to modify
records	array	An array of objects of type record that contain the data for one table row each.

Throws

```
error::stdlib::gui
```

Remarks

Calling this procedure replaces the entire contents of a table component.

This procedure can only be used on table components.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        ...

            case "button":
                switch (g.action.button)
                {
                   case "resetTable":
                      {
                        var rows = [
                                      {…},
                                      …
                                    };
                        gui::setRows(“myTable”, rows);
                        gui::acceptData();
                      }
                      break;

                   …
                }
                break;

        …
    }
}
```

gui::setSelectedRows

Prototype

```
procedure gui::setSelectedRows(componentName, selectedRows)
```

Description

Updates the setting of the selected rows in a table component. With this function it is possible to change the state of the “select rows” check boxes displayed for a table.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the table component to modify.
selectedRows	array	An array of Booleans, each of which corresponds to the required state of the selected rows' check boxes (a true value indicates a selected check box).

Throws

```
error::stdlib::gui
```

Remarks

This procedure can only be used on table components.

Example

```
gui::setSelectedRows("myComponent", [true,true,false]);
```

gui::update

Prototype

```
procedure gui::update(componentName, record)
procedure gui::update(componentName, record, rowIndex)
```

Description

Update a record in a component.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component to modify.
record	object	An object of type record that contains the data for the record to update.
rowIndex	int	The index of the table row to update if componentName is the name of a table component. This parameter must be omitted for card components or set to zero.

Throws

```
error::stdlib::gui
```

Remarks

This procedure updates a single record in a card or table component. To replace the entire contents of a table component, use the procedure gui::setRows().

The rowIndex parameter must be passed for table components and must be omitted for card components.

This procedure cannot be used on composer components.

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        case "button":
            switch (g.action.button)
            {
                case "updateRecord": {
                    var record   = {...};
                    var rowIndex = ...;
                    gui::update("myTable", record, rowIndex);
                    gui::acceptData();
                }
            break;
            ...
        }
        break;
    }
}
```

gui::updateGrandTotals

Prototype

```
procedure gui::updateGrandTotals(componentName, record)
```

Description

Update custom grand total values.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component to update.
record	object	An object of the type record that contains the data for the grand totals to update.

Throws

```
error::stdlib::gui
```

Example

```
var g = gui::get();
if (g != null)
{
```

```
    switch(g.action.kind)
    {
        case "button":
            switch (g.action.button)
            {
                case "updateGrandTotals": {
                    var record   = {...};
                    gui::updateGrandTotals("myTable", record);
                  }
                  break;
                ...
            }
            break;
        }
    }
```

gui::updateHeadings

Prototype

```
procedure gui::updateHeadings(componentName, headings)
```

Description

Update table headings.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component to update.
headings	object	An object with property names that correspond to the column names. The values of the properties will be transferred to the headings.

Throws

```
error::stdlib::gui
```

Example

```
var g = gui::get();
if (g != null)
{
    switch(g.action.kind)
    {
        case "button":
            switch (g.action.button)
            {
                case "updateHeadings":
                {
                    var headings = { day1: "Wednesday" };
```

```
                gui::updateGrandTotals("myTable", headings);
            }
            break;
        ...
    }
    break;
}
}
```

gui::updatePartial

Prototype

```
procedure gui::updatePartial(componentName, record)
procedure gui::updatePartial(componentName, record, rowIndex)
```

Description

Update some fields in a component.

Parameter Descriptions		
Name	**Type**	**Description**
componentName	string	The name of the component to modify.
record	object	An object of the type record that contains the data for the fields that should be updated.
rowIndex	int	The index of the table row to update if componentName is the name of a table component. This parameter must be left out for card components or set to zero.

Throws

```
error::stdlib::gui
```

Remarks

This procedure does exactly the same as gui::update(), except that only a subset of the fields is required. Omitted fields are left unchanged by the GUI framework.

gui::updateTrafficLightingGlobals

Prototype

```
procedure gui::updateTrafficLightingGlobals(componentName, record)
```

Description

Updates the traffic lighting global values for the specified component.

Parameter Descriptions		
Name	**Type**	**Description**

Parameter Descriptions		
componentName	string	The name of the component for which the values should be updated.
record	object	Partial data for the new globals. This parameter is an object of the type record.

Throws

`error::stdlib::gui`

Remarks

The record is merged with the existing data, so you need not specify all properties in the globals, only those that need to be updated.

Example

```
gui::updateTrafficLightingGlobals("myTable",
    { currentDate: 1.1.2004 });
```

Composer Package Routines

A few subroutine prototypes are defined for use in composer packages. These routines are callbacks, which the GUI framework invokes if they are found in a composer package, and therefore these functions must obey the prototypes that are expected by the framework. The legal prototypes for each of these functions are described in the following sections.

getComposerInterface

Prototype

```
public function getComposerInterface ()
```

Description

Returns the interface specification for a composer package.

Parameter Descriptions

None

Return Value

An object with the composer package interface as described in “Composer Packages.” This object must as a minimum contain the function getSpecification() for returning the composer package specification.

Remarks

This function is optional. If it is not defined in the package, the set of public functions in the package itself is assumed to be the composer package interface.

Example

```
public function getComposerInterface()
{
    return {
        getSpecification: function(userData, init) { ... },
        ...
    }
}
```

getDependency

Prototype

```
function getDependency(userData)
```

Description

Returns the components on which the current composer package depends. See “Package Dependencies” for a description of composer package dependencies.

Parameter Descriptions		
Name	**Type**	**Description**
userData	value	The user data that is specified to gui::open().

Return Value

An array of strings that identifies the names of the components that the package depends on.

Remarks

This function is optional and can be omitted if the composer package has no dependencies, or if gui::addDependency() is used.

Example

```
public function getDependency(userData)
{
   return [ "myTable" ];
}
```

getSpecification

Prototype

```
function getSpecification(userData)
function getSpecification(userData, init)
```

Description

This function should return the specification of a composer package.

Parameter Descriptions		
Name	**Type**	**Description**
userData	value	The user data that is specified to gui::open(). This is a modifiable reference to the user data, and new data may be inserted into it (assuming that it is an object or array).
init	value	Optional initializing value with the value of the init property in the layout from which the composer package was included. If that property is not defined in the layout, the value defaults to null. This parameter can be omitted if the initializing value has no relevance to the implementation of getSpecification().

Return Value

A composer package specification as described in “Composer Package Specification.”

Remarks

This function must exist in a composer package, either as a public function in the package or in the return value from getComposerInterface().

Example

```
public function getSpecification(userData, init)
{
    return {
        components: { ... },
        layout: { ... },
        data: { ... },
        composerData: ...
    }
}
```

getTitle

Prototype

function getTitle(userData)

Description

A top-level composer package (see Top-Level Conmposer Packages) must have in its interface supply the getTitle() function, which returns the title of the window.

Parameter Descriptions		
Name	**Type**	**Description**
userData	value	The user data that is specified to gui::open().

Return Value

A string that contains the title of the window.

Remarks

This function must exist in the interface of a top-level composer package.

Example

```
public function getTitle(userValue)
{
    return "My title";
}
```

Event Handler List

As seen in the previous section, there are a few more event handlers than there are basic events (as listed in “Event Description.”). A good example is the handleDependencyUpdate() event handler, which has no corresponding event. The complete list of event handlers is described in the following sections.

All event handlers are optional.

Example

```
   // This handler ensures some weather info component is updated
   // every time any event is received from the GUI framework.
public procedure handleBegin(eventData)
{
   var weatherInfo = myGetWeatherInfoFunction();

   gui::update("myWeatherComponent", weatherInfo);
}
```

handleButton

Prototype

```
public function handleButton(eventData)
```

Associated Event Kind

Button

Action Data

Same as for the button event.

Description

This handler is called when a user activates a button in one of the components that are defined by the composer package.

Return Value

True if the handler updated anything; otherwise, false.

Example

```
public function handleButton(eventData)
{
   if (eventData.action.button == "submit")
      maconomy::dialogAction(dialogId, "submit");
      // Data was changed, so return true
   return true;
}
```

handleClose

Prototype

```
public procedure handleClose (eventData)
```

Associated Event Kind

None

Action Data

None

Description

This handler is called when the GUI window in which its components are presented is closed. This happens no matter how the window is closed. This can be when a call to gui::close() or gui::open() (in an existing window) is performed, when a child window is closed by its parent, or when a user closes the browser or visits a new page. This means that the close event handler may be called more or less at any time, so make sure that the handler does not depend on external data (which may have disappeared) or expects the close event to arrive at some certain point in time.

A close handler is typically used to clean up after the component. A typical clean-up action could be to close a dialog used by the component.

Calling a gui:: function from a close event handler is not allowed. Doing so may cause unexpected behavior.

Example

```
   // Cleans up after this component.
public procedure handleClose(eventData)
{
   maconomy::dialogClose(eventData.userValue.component.dialogId);
}
```

handleDataChanged

Prototype

```
public procedure handleDataChanged (eventData)
```

Associated event Kind

None

Action Data

None

Description

This handler is called when a user has modified some data and then submitted it by pressing Enter, clicking a button, or changing focus to another component.

This handler is called before the action event (enter, button, setFocus, and so on).

Because both this handle and the action handler are called (if data is modified), the actual data dependence functionality should be put in this handler. Consider, as an example, a time sheet

where a user can click a “Submit Time Sheet” button. In such an application, you want to make a call to maconomy::dialogUpdateUpper() if the user has changed anything in the upper pane before you react to the button activation, so you put the dialog update in the handleDataChanged handler and the dialog action in the handleButton handler.

Example

```
   // Update Maconomy API depending on the input
public procedure handleDataChanged(eventData)
{
   var data = eventData.data.myCard.rows[0];
   maconomy::dialogUpdateUpper(dialogId, data);
}
```

handleDeleteRow

Prototype

```
public function handleDeleteRow(eventData)
```

Associated Event Kind

```
deleteRow
```

Action Data

Same as for the deleteRow event.

Description

This handler is called every time that a user clicks the “Delete Row” key.

Return Value

True if the handler updated anything; otherwise, false.

Example

```
public function handleDeleteRow(eventData)
{
   if ( rowDeleteIsOk(eventData.action) )
   {
      gui::deleteRow(eventData.action.component,
                     eventData.action.rowIndex);
      gui::acceptData();
      return true;
   }
   else
      return false;
}
```

handleDependencyUpdate

Prototype

```
public procedure handleDependencyUpdate(eventData)
```

Associated Event Kind

None

Action Data

None

Description

This function is called every time any of the components on which the composer package depends are changed. See “Package Dependencies” for more information about component dependencies.

It is not possible to get information about which component was actually changed.

Example

```
public procedure handleDependencyUpdate(eventData)
{
   var summary = calculateSummary(eventData.data);

   gui::update(myCard, summary);
}
```

handleEnd

Prototype

```
public procedure handleEnd (eventData)
```

Associated event Kind

None

Action Data

None

Description

This handler is always called in all composer packages. It is called after all of the other event handlers, and right before the call to gui::get() finishes.

The calling sequence of the handlers in different packages is defined by the component dependency (see “Package Dependencies”). The sequence is opposite that of handleBegin.

Example

```
   // A procedure that calculates a summary based on whatever
   // has been updated by all other event handlers.
public procedure handleEnd(eventData)
{
   var summary = calculateSummary(eventData.data);
   gui::update{"mySummary", summary);
}
```

handleEnter

Prototype

```
public function handleEnter (eventData)
```

Associated Event Kind

enter

Action Data

Same as for the enter event.

Description

This handler is called every time that a user presses Enter.

Return Value

True if the handler updated anything; otherwise, false.

Example

```
public function handleEnter(eventData)
{
   var c = calculateSomething(eventData.data.myComponent);

   gui::update("myComponent", c);

      // Data was changed, so return true
   return true;
}
```

handleFieldChanged

Prototype

```
public procedure handleFieldChanged (eventData)
```

Associated event Kind

fieldChanged

Action Data

Same as for the fieldChanged event.

Description

This handler is called every time that a field for which the fieldChanged event has been enabled is changed.

Example

```
public procedure handleFieldChanged(eventData)
{
   gui::alert("Field " + eventData.action.field +
```

```
            " in component " + eventData.action.component +
            " has been changed. ");
}
```

Example

```
public procedure handleFieldChanged(eventData)
{
   gui::alert("Field " + eventData.action.field +
            " in component " + eventData.action.component +
            " has been changed. ");
}
```

handleFieldChangedViaAutofetch

Prototype

```
public procedure handleFieldChangedViaAutofetch (eventData)
```

Associated event Kind

fieldChangedViaAutofetch

Description

This handler is called every time that a field for which the fieldChangedViaAutofetch event has been enabled is changed.

Example

```
public procedure handleFieldChangedViaAutofetch(eventData)
{
   gui::alert("Field " + eventData.action.field +
            " in component " + eventData.action.component +
            " has been changed. ");
}
```

Example

```
public procedure handleFieldChangedViaAutofetch(eventData)
{
   gui::alert("Field " + eventData.action.field +
            " in component " + eventData.action.component +
            " has been changed. ");
}
```

handleFieldClicked

Prototype

```
public function handleFieldClicked (eventData)
```

Associated Event Kind

fieldClicked

Action Data

Same as for the fieldClicked event.

Description

This handler is called every time that a field for which the fieldClicked event has been enabled is clicked.

Return Value

True if the handler updated anything; otherwise, false.

Example

```
public function handleFieldClicked(eventData)
{
   gui::alert("Field " + eventData.action.field +
              " in component " + eventData.action.component +
              " has been clicked. ");

      // No data was changed, so return false
   return false;
}
```

handleGotFocus

Prototype

```
public function handleGotFocus (eventData)
```

Associated Event Kind

gotFocus

Action Data

Same as for the gotFocus event.

Description

This handler is called every time that a record for which the gotFocus event has been enabled receives focus, independently of whether or not a user has changed any data.

Return Value

True if the handler updated anything; otherwise, false.

Example

```
   // This handler updates summary information every time that the
   // user enters a certain record (for instance a table row)
public function handleGotFocus(eventData)
{
   var summary = calculateSummary(eventData.data);

   gui::update("myComponent", summary);
```

```
        // Data was changed, so return true
    return true;
}
```

handleGrandTotalClicked

Prototype

```
public function handleGrandTotalClicked (eventData)
```

Associated Event Kind

grandTotalClicked

Action Data

Same as for the fieldClicked event, except for a row index that is not available.

Description

This handler is called every time that a grand total for which the grandTotalClicked event has been enabled is clicked.

Return Value

True if the handler updated anything; otherwise, false.

Example

```
public function handleGrandTotalClicked(eventData)
{
    gui::alert("Grand total " + eventData.action.field +
               " in component " + eventData.action.component +
               " has been clicked. ");

        // No data was changed, so return false
    return false;
}
```

handleHeadingClicked

Prototype

```
public function handleHeadingClicked (eventData)
```

Associated Event Kind

headingClicked

Action Data

Same as for the fieldClicked event, except for a row index that is not available.

Description

This handler is called every time that a heading for which the headingClicked event has been enabled is clicked.

Return Value

True if the handler updated anything; otherwise, false.

Example

```
public function handleHeadingClicked(eventData)
{
   gui::alert("Heading " + eventData.action.field +
              " in component " + eventData.action.component +
              " has been clicked. ");

      // No data was changed, so return false
   return false;
}
```

handleInitialized

Prototype

```
public procedure handleInitialized (eventData)
```

Associated Event Kind

Initialized

Action Data

None

Description

This function is called when the GUI framework is initialized and ready for use.

Example

```
public procedure handleInitialized(eventData)
{
   gui::disableComponent("myComponent");
}
```

handleInsertRow

Prototype

```
public function handleInsertRow (eventData)
```

Associated event Kind

insertRow

Action Data

Same as for the insertRow event.

Description

This handler is called every time that a user clicks the “Insert Row” icon.

Return Value

True if the handler updated anything; otherwise, false.

Example

```
public function handleInsertRow(eventData)
{
   var newRow = { … };
   if (insertIsOk(eventData))
      gui::insertRow(eventData.action.component,
                     newRow,
                     eventData.action.rowIndex);

      // Data was changed, so return true
   return true;
}
```

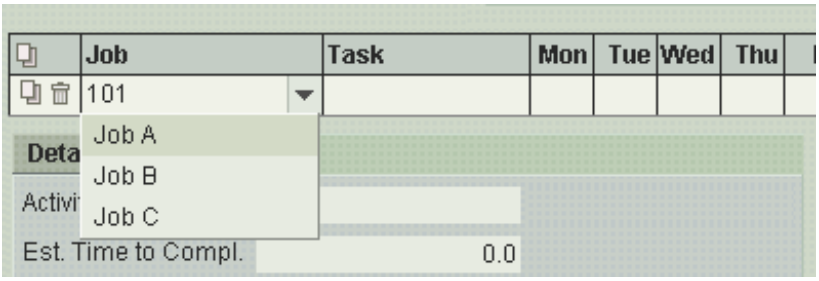
Part 4 – M-Script Favorite Values

This section describes how server-side M-Script can be used for creating a list of Favorite values in Maconomy dialogs to simplify data entry. Favorites are displayed as a combo box (an editable drop-down list box) in the Maconomy Portal and in the Maconomy client for the Java™ platform version 3 and above.

In Maconomy, you can use the CTRL+G search option to look up a value that you want to enter into a field in a dialog window. For instance, when completing a time sheet, you can use CTRL+G to find the job/project number on which your hours should be entered.

However, users often find themselves entering time on the same selection of job numbers. Over time, the job numbers change, but usually the number of jobs in current use is relatively stable. For this reason, Maconomy supports the use of *Favorite Values*.

Favorite values are presented as a list in the fields for which favorite values exist. For instance, if you place the cursor in the **Job No.** field in the Portal Time Sheets component, and favorites are enabled for that field, the standard input field is changed to a combo box—a kind of drop-down list box, from which you can select a value. The values in the list are generated by an M-Script that resides on the server. For instance, the script can select the job numbers that you used for the last two weeks, taking into consideration whether the job numbers are valid in the current context (for example, checking whether you have access to the job, whether the job has been blocked for entries, and so forth). However, you are not forced to select a value from the list; you are free to enter another valid job number. The following figure illustrates how the list of favorites might look in the Portal Time Sheets component.



Although job numbers in time sheets are used as an example throughout this document, you can use the Favorites functionality anywhere in Maconomy, such as for selecting task numbers in expense sheets, favorite items in item catalogs in the e-Procurement Portal, and so on.

Requirements

To use the Favorites functionality, the following requirements must be met:

- Maconomy Server version 33.00.
- M-Script version 13.0.
- An M-Script Developer license is needed for installing custom favorite packages.
- The Favorite Values feature is currently supported in the Maconomy Portal and in the Maconomy client for the Java™ platform, version 3.*x*, when running as a standalone application.

Coding Favorites

Favorite values are calculated by small M-Script packages that reside on the Maconomy server. To calculate favorite values for a specific Maconomy dialog, you must first create an M-Script package (with a major version of 1) with exactly the same name as the internal name of the dialog in question, and then place this package into one of the three favorite package directories on the Maconomy server. The directory to use depends on the package:

- Standard packages supplied by the Maconomy development team:

  <MaconomyHome>/MaconomyDir/Mscripts/Favorites

- Solution-specific packages:

  <MaconomyHome>/CustomizationDir/Solution/Mscripts/Favorites

- Custom-made packages:

  <MaconomyHome>/CustomizationDir/Custom/Mscripts/Favorites

The favorite packages must include a public function named **getFavoriteBuilder**. When that function is invoked, it returns an M-Script object with various call-back methods. This function is what software developers normally call a factory function because it produces objects with specific behaviors. The object that is returned from the factory function must contain one method for each field in the dialog pane for which you want to calculate favorite values. The methods must be named exactly as the corresponding dialog field.

Example

The following simple example package for the dialog TimeSheets returns a favorite job number for a time entry. The numbers in the margin refer to comments in subsequent sections.

```
   #version 13
   package TimeSheets(1.0.0);
      // Callback object for handling favorites in timesheet rows var
      timeSheetRowhandler =
   {
      // Callback method for the job number dialog field
      // Note that method name is identical to dialog field name
1  JobNumber: function(parameters)
      {
         return
         [
            { title: "My Favorite Job", value: "101010" }
         ];
      }
   };
      // Factory function for the creation of Favorite handlers
2  public function getFavoriteBuilder(parameters)
      {
         if (parameters.pane.kind == "upper")
            return {};
         else
            return timeSheetRowhandler;
      }
```

Public Function getFavoriteBuilder

The core factory function (marked as 2)in the preceding example) returns an empty object for the upper pane because you do not want to add any favorites for fields in that pane. For the lower pane, the function returns a callback object with, in this example, exactly one method named JobNumber, which is used to calculate favorite job number values.

The job number calculation is trivial in this case—it just returns an array of the same format that M-Script's GUI framework uses for favorite values. This must be an array of objects with a title and a valueproperty. The title is displayed in the favorites combo box, whereas the value is pasted into the dialog field when you select the favorite value.

The parameter argument that is passed to the factory function and field functions is an object that has the following properties.

	Type	Description
userName	string	The name of the current user
dialog		Object describing the current dialog
dialog.kind	string	The dialog kind, which can be either card or card/table
dialog.name	string	The dialog name
pane		The object that describes the current pane
pane.kind	string	The pane kind, which can be either upper or lower
pane.name	string	The pane name
field	string	The current field name. This property is not available for the factory function, only for the individual field methods.

Return Values

The favorite calculating functions (marked as 1 in the preceding example) must return an array of objects that have associated title and valueproperties.

For instance, you can assign three fixed job numbers in this way:

```
            JobNumber: function(parameters)
            {
                return
    [
        { title: "Job A", value: "101" },
        { title: "Job B", value: "202" },
        { title: "Job C", value: "303" }
    ];
}
```

However, a more useful and common procedure is to insert SQL statements that can select job numbers based on any parameters that you choose. For example, this could be an SQL statement that selects those job numbers to which the current user has access, and on which the user has entered hours at least once over the last two weeks, omitting any blocked jobs.

Favorites with Multiple Target Fields

You might want to be able to select one job and have other, related fields filled in as you select the job. The feature that enables you to do that is described here.

Note, however, that it is not yet fully implemented in all Maconomy clients. It is only used for the Maconomy client for the Java™ platform, where it helps users to make a time sheet entry with job number, task, and activity.

The idea is to return not only a value for the current field, but also values for a set of other fields. To make things backward-compatible with the single item return value, a property named values is added. This property is an object of values for the other targeted fields.

Example

```
JobNumber: function(parameters)
{
    // When selecting a Job favorite, fill in Job No.,
    // Activity, and Task as well
    return
    [
        { title: "Job A", values: { JobNumber: "101", Activity: "Act 1", Task:
        "T-1" } },
        { title: "Job B", values: { JobNumber: "202", Activity: "Act 2", Task:
        "T-2" } },
        { title: "Job C", values: { JobNumber: "303", Activity: "Act 3", Task:
        "T-3" } }
    ];
}
```

The format of the returned value is always converted by the server to a normal form with both a values and a value property.

M-Script Maconomy API

The values that are returned from favorite calculations can be found in the data that is returned from the Maconomy API function **maconomy::dialogGetDef**, if it is called correctly. Because the calculation of favorites adds some overhead to the function, and they are not always needed, **maconomy::dialogGetDef**must specifically be told to get the favorites. You can do this by invoking it in the following way:

```
var dialogDef = maconomy::dialogGetDef(dialogID,
    { getFavorites: true } );
```

The favorite values are then added to the individual field definitions and are ready for use in M-Script's GUI without any modifications.

Debugging

It can be quite difficult to debug these kinds of scripts, because a small misspelling of a package name means that it never gets loaded. If you are in doubt about whether a script works, try to write some incorrect M-Script syntax in the favorite package file. If you then get a parse error while loading the dialog, you know that the favorite package was actually loaded.

You can also throw random exceptions at specific places in the code to see if it ever gets executed.

You cannot use M-Script's log file using io::log().

In addition, note that a future version of the Maconomy server might choose to cache the scripts. This means that the server will not load your script when you make changes to it. In that case, you should restart either the web daemon or the Maconomy server after updating a script.

Part 5 – M-Script Standard Packages

The Maconomy M-Script language can be extended with the use of packages. A number of packages are supplied with the Maconomy system. These packages are called “Standard M-Script Packages” or just “Standard Packages” and are located in the following folder on the Maconomy server:

```
<Maconomy server home>/<tpu-dir>/MScript/Packages/MScript
```

Note that there is one standard package folder per application home. When you install a TPU, MConfig updates the packages to the version of the TPU that you are installing. For more information, see “Standard Packages and Site Packages” in the M-Script Language Reference.

When writing M-scripts, you can address the functions in these packages using the notation `mscript::<package>::<function>.<method>`; for example:

```
mscript::maconomy::server.login(user, passwd)
```

The following packages are supplied with the Maconomy system (as of version 9.0):

- **Interface** — Functions for working with Maconomy dialog data.
- **Algorithms** — A number of functions for finding and manipulating text and for applying functions to the elements of an array.
- **Integration** — Functions for importing, exporting, and manipulating data to/from the Maconomy application.
- **Web services** — Functions and procedures for creating web services.
- **Customization** — Functions for checking whether a given package exists in customized form.
- **Mail** — Functions for interfacing with an SMTP server and sending mail with or without attachments.
- **Semaphore** — Functions for preventing locking conflicts in an M-script by setting up a semaphore (mutex).

Other packages for internal use are part of the standard packages as well.

Maconomy Interface

This section describes the `mscript::Maconomy` and `mscript::maconomy::dialog` packages. These packages offer object-based tools for working with the M-Script Maconomy API.

mscript::maconomy

The `mscript::maconomy` package implements an object abstraction on top of the server connection and login session concepts of M-Script. The package interface contains the `Server()` constructor function, which can be used to construct servers.

function Server([server [, options]])

The `Server` constructor function returns a server object, which contains some simple routines for automating aspects of the Maconomy server communication. When logging in to a Maconomy server through this object, it checks to see if a session exists, and if not, a session is created. If a session was created, it is deleted when logging out through this object.

The server object can be used manually by invoking the login and logout procedures in appropriate places, but it can also act as a context wrapper by assigning the `inner` function to whatever should be performed while logged in, and then invoking the `doInner` function.

The function accepts one or two optional parameters. If you specify parameters, the first must be the serverparameter, possibly followed by the `args` parameter.

Parameters

Name	Type	Description
`server`	server handle or server label	The label of or handle to the Maconomy server. If omitted or `null`, the default server is assumed.
`options`	object	Optional arguments to the server object.

Arguments

The optional argument object `options` can be used to pass arguments to the server object. Currently supported arguments are described in the following table.

Name	Type	Description
`deletesession`	bool	Set to true if the server object should always delete the session when done, regardless of whether it created it.
`inner`	function	Optional function, which is executed in the context of the login if `doInner`is invoked.

Returns

A server object.

Throws

`maconomy::Server({message: string})` if object creation fails.

History

All package versions.

Example

```
uses mscript::maconomy(1) as Maconomy;
var serverLabel = "srv1";
var server = Maconomy::Server(serverLabel);
```

Server Object

This section describes the interface of the server object that is returned by the `Server` constructor function.

login: procedure(user, passwd)

This procedure attempts to perform a login on the Maconomy server with the specified username and password.

Parameters

Name	Type	Description
`user`	string	Name of the Maconomy user to log in.
`passwd`	string	Password of the specified Maconomy user

Throws

Whatever `maconomy::login()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy(1) as Maconomy;
var user = ...;
var passwd = ...;
var server = Maconomy::Server();
server.login(user, passwd);
```

logout: procedure()

This procedure attempts to perform a logout from the Maconomy server. If the server object created the session file or has been given the argument `deleteSession:true`, the session file is deleted.

Throws

Whatever `maconomy::logout()` or `deletesession()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy(1) as Maconomy;
var user = ...;
var passwd = ...;
var server = Maconomy::Server();
server.login(user, passwd);
...
// Perform logout and delete session if 'server' created one.
server.logout();
```

inner: function()

This function can be used to wrap the operations that should be performed while logged in to the server. It is called when calling the `doInner()` function.

Returns

Any value is legal, including `null`.

History

All package versions.

Example

See `doInner`.

doInner: function (user, passwd)

This function first attempts a login to the Maconomy server with the specified username `user` and password `passwd`. If that is successful, the function `inner` is invoked, followed by a controlled logout and possible session deletion. If `inner` throws an exception, the intercepted exception is rethrown after the logout has been completed; otherwise, the value returned by inner is returned by `doInner`.

Parameters

Name	Type	Description
`user`	string	Name of the Maconomy user to log in.
`passwd`	string	Password of the specified Maconomy user.

Returns

The value returned by `inner`.

Throws

Whatever `login()`, `inner()` or `logout()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy(1) as Maconomy;
function innerFunc()
{
    // Operations while logged in:
    ...
}
var server = Maconomy::Server(null, {inner: innerFunc});
server.doInner();
```

Card: function (dialogName [, options])

This function creates a card dialog object connected to the server object's Maconomy server. The name of the dialog is specified in the `dialogName` parameter, and the optional argument `options` will be passed to the dialog object being created.

Parameters

Name	Type	Description
`dialogName`	string	Internal name of the card dialog.
`options`	object	Optional arguments to the card dialog object.

Returns

A new card dialog object.

Throws

`maconomy::Server({message: string})` if the server object does not have a valid server handle. Otherwise, whatever `dialog::Card()` might throw.

History

1.0: First version of `Card (dialogName)`.

1.1: Support for optional arguments to the dialog object.

Example

```
uses mscript::maconomy(1) as Maconomy;
var dialogName = ...;
var args = { readonlyData: true };
var server = Maconomy::Server();
var dialog = server.Card(dialogName, args);
```

CardTable: function (dialogName [, options])

This function creates a card/table dialog object connected to the server object's Maconomy server. The name of the dialog is specified in the `dialogName` parameter, and the optional argument `options` are passed to the dialog object being created.

Parameters

Name	Type	Description
`dialogName`	string	Internal name of the card/table dialog.
`options`	object	Optional arguments to the card/table dialog object.

Returns

A new card/table dialog object.

Throws

`maconomy::Server({message: string})` if the server object does not have a valid server handle. Otherwise, whatever `dialog::CardTable()` might throw.

History

1.0: First version of `CardTable (dialogName)`.

1.1: Support for optional arguments to the dialog object.

Example

```
uses mscript::maconomy(1) as Maconomy;
var dialogName = ...;
var args = { readonlyData: true };
var server = Maconomy::Server();
var dialog = server.CardTable(dialogName, args);
```

mscript::maconomy::dialog

This package implements object wrappers around the dialog data returned by the Maconomy API functions. A dialog object can store the dialog ID and the dialog data for a single dialog window, and can also contain methods representing a subset of the operations possible on card and card/table dialogs.

The advantage of a dialog object over the raw Maconomy API functions is that all data that relates to the dialog is encapsulated within a common structure. In addition, the dialog object also handles common tasks such as restoring dialog data after an error and database rollback.

The dialog object contains two or three properties, depending on whether it represents a card or a card/table dialog.

Name	Type	Description
`common`	object	This object contains methods and data shared by the upper and lower pane of a

Name	Type	Description
		dialog.
`upperPane`	object	This object contains methods operating on the upper pane of a dialog.
`lowerPane`	object	This object contains methods operating on the lower pane of a card/table dialog.

Two constructors, `Card` and `CardTable`, exist to create dialog objects for a card dialog and a card/table dialog, respectively.

function Card(dialogName [, options])

The `Card` constructor function returns a dialog object for the card dialog named `dialogName`. The optional parameter `options` is an object that can be used to specify additional arguments to the constructor.

Parameters

Name	Type	Description
`dialogName`	string	Internal name of the card dialog.
`options`	object	Optional arguments to the card dialog object.

Properties

The `options` parameter supports the following properties.

Name	Type	Description
`serverHandle`	server handle	A handle to the server which should be used. If not specified, the default server handle is assumed.
`readonlyData`	bool	Set to `true` to improve performance by returning dialog data in read-only format.
`disableHLL`	bool	Set to `true` to disable high-level locking for this dialog.

Returns

A new card dialog object.

Throws

`init::Initialize({message: string})` if object creation fails.

History

1.0: First version of `Card (dialogName [, args]).`

1.1: Optional arguments `readonlyData` and `disableHLL`.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var dialogName = ...;
var serverLabel = ...;
var args = {
   serverHandle: maconomy::getServerHandle(serverLabel),
   disableHLL: true
};
var dialog = Dialog::Card(dialogName, args);
```

function CardTable(dialogName [, options])

The `CardTable` constructor function returns a dialog object for the card/table dialog named `dialogName`. The optional parameter `options` is an object that can be used to specify additional arguments to the constructor.

Parameters

Name	Type	Description
`dialogName`	string	Internal name of the card/table dialog.
`options`	object	Optional arguments to the card/table dialog object.

Properties

The `options` parameter supports the following properties.

Name	Type	Description
`serverHandle`	server handle	A handle to the server that should be used. If not specified, the default server handle is assumed.
`readonlyData`	bool	Set to `true` to improve performance by returning dialog data in read-only format.
`disableHLL`	bool	Set to `true` to disable high-level locking for this dialog.

Returns

A new card/table dialog object.

Throws

`init::Initialize({message: string})` if object creation fails.

History

1.0: First version of `CardTable (dialogName [, args])`.

1.1: Optional arguments `readonlyData` and `disableHLL`.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var dialogName = ...;
var serverLabel = ...;
var args = {
   serverHandle: maconomy::getServerHandle(serverLabel),
   disableHLL: true
};
var dialog = Dialog::CardTable(dialogName, args);
```

Common Dialog Operations

The `common` object within a dialog object contains methods and data, which are shared by the upper and lower panes of a dialog. The methods found in this object can be split into two categories:

- Methods that correspond directly to operations on the dialog. The names of these methods start with an uppercase letter and follow the naming convention used in the dialog API.
- Methods that work on the data that is contained within the object. The names of these methods start with a lowercase letter.

GetDef: function()

Returns the definition object for the dialog as defined by `maconomy::dialogGetDef()`. The definition object is cached when first retrieved, so subsequent calls to this method result in very little additional overhead.

Returns

The `dialogDef` object for the current dialog.

Throws

Whatever `maconomy::dialogGetDef()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
var ddef = dialog.common.GetDef();
```

Get: function(upperKey)

Gets the record corresponding to the specified upper key. This method corresponds to `maconomy::dialogGet()`.

Returns

The `dialogReturn` data for the record corresponding to the specified upper key.

Throws

Whatever `maconomy::dialogGet()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
var ddata = dialog.common.Get(upperKey);
```

GetFirst: function()

Gets the first record in a dialog. This method corresponds to `maconomy::dialogGetFirst()`.

Returns

The `dialogReturn` data for the first record.

Throws

Whatever `maconomy::dialogGetFirst()` might throw.

History

Introduced in package version 1.2.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
var ddata = dialog.common.GetFirst();
```

GetLast: function()

Gets the last record in a dialog. This method corresponds to `maconomy::dialogGetLast()`.

Returns

The `dialogReturn` data for the last record.

Throws

Whatever `maconomy::dialogGetLast()` might throw.

History

Introduced in package version 1.2.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
var ddata = dialog.common.GetLast();
```

GetNext: function()

Gets the next record in a dialog. This method corresponds to `maconomy::dialogGetNext()`.

Returns

The `dialogReturn` data for the next record.

Throws

Whatever `maconomy::dialogGetNext()` might throw.

History

Introduced in package version 1.2.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
var ddata = dialog.common.GetNext();
```

GetPrevious: function()

Gets the previous record in a dialog. This method corresponds to `maconomy::dialogGetPrevious()`.

Returns

The `dialogReturn` data for the previous record.

Throws

Whatever `Maconomy::dialogGetPrevious()` might throw.

History

Introduced in package version 1.2.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
var ddata = dialog.common.GetPrevious();
```

Close: procedure()

Close the dialog if it is open; otherwise, do nothing. This method corresponds to `maconomy::dialogClose()` on an open dialog.

Throws

Whatever `maconomy::dialogClose()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
...
dialog.common.Close();
```

Commit: procedure()

Commit changes in the dialog to the database. This method corresponds to `maconomy::commit()`.

Throws

Whatever `maconomy::commit()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
var ddata = dialog.common.Get(upperKey);
...
dialog.common.Commit();
```

Action: function(action [, rowNo])

Performs the specified action on the dialog. The `action` parameter can be either the internal name or the index of the action to perform. The optional argument `rowNo` can be used to specify the index of a lower pane row. This method corresponds to `maconomy::dialogAction()`.

Parameters

Name	Type	Description
`action`	string or int	The internal name or index of the action to perform.
`rowNo`	int	Optional index of a lower pane row.

Returns

The `dialogReturn` data returned by `maconomy::dialogAction()`.

Throws

Whatever `maconomy::dialogAction()` might throw.

History

Introduced in package version 1.2.

Example

```
uses mscript::maconomy::dialog(1.2) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
dialog.common.GetFirst();
var actionIndex = ...;
var ddata = dialog.common.Action(actionIndex);
```

GetState: function()

Returns the current state of the dialog. This method corresponds to `maconomy::dialogGetState()`.

Returns

A string with the current state of the dialog state machine.

Throws

Whatever `maconomy::dialogGetState()` might throw.

History

Introduced in package version 1.2.

Example

```
uses mscript::maconomy::dialog(1.2) as Dialog;
...
var dialog = Dialog::CardTable(...);
...
var state = dialog.common.GetState();
```

GetId: function()

Returns the dialog ID of the dialog. If the dialog is not currently open, it is opened to obtain a dialog ID.

Returns

The dialog ID of the dialog.

History

All package versions.

GetData: function()

Returns the return data for the current record. If no return data is currently available (that is, there is no record), an exception is thrown. The type of the return data is the `dialogReturn` object returned by `maconomy::dialogGet()`.

Returns

The current record data object.

Throws

`maconomy::dialog::getData({message: string})` if there is no record data.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var upperKey = { ... };
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
// Get the record data but don't save it:
dialog.common.Get(upperKey);
// 'dialog.common' has saved the record data internally.
// We can always ask for it when we need it:
var ddata = dialog.common.getData();
```

Upper-Pane Dialog Operations

The `upperPane` object contains methods that operate on the card pane of a card or card/table dialog. This object uses the `common` object described previously to maintain information about dialog ID, return data, and so forth.

GetData: function()

Gets the dialog data for the card pane of the current dialog record. Returns the `upperPane` object of the dialog data object. This method is a shorter way of writing `common.getDialogData().upperPane`.

Returns

The `upperPane` object of the current dialog data object.

Throws

Whatever `common.getDialogData()` might throw.

History

Introduced in package version 1.2.

Example

```
uses mscript::maconomy::dialog(1.2) as Dialog;
```

```
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
dialog.GetFirst();
var upperData = dialog.upperPane.getData();
```

New: function(upperKey)

Creates an empty upper pane record with the specified upper key. This method corresponds to `maconomy::dialogNewUpper()`.

Parameters

Name	Type	Description
`upperKey`	object	The upper key of the new record.

Returns

The `dialogReturn` data for the new record.

Throws

Whatever `maconomy::dialogNewUpper()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var upperKey = { ... };
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
dialog.upperPane.New(upperKey);
```

Put: function(upperData)

Inserts the specified upper data into a newly created upper record. This method corresponds to `maconomy::dialogPutUpper()`.

Parameters

Name	Type	Description
`upperData`	object	The upper pane data to put.

Returns

The `dialogReturn` data for the current record.

Throws

Whatever `maconomy::dialogPutUpper()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var upperData = { ... };
var upperKey = { ... };
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
dialog.upperPane.New(upperKey);
dialog.upperPane.Put(upperData);
```

NewAndPut: function(upperData)

Create an upper record with the specified upper data. The upper data must contain a full upper key. This method corresponds to `maconomy::dialogNewAndPutUpper()`.

Parameters

Name	Type	Description
`upperData`	object	The upper pane data, including a full upper key.

Returns

The `dialogReturn` data for the current record.

Throws

Whatever `maconomy::dialogNewAndPutUpper()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var upperData = { ... };
var dialogName = ...;
var dialog = Dialog::Card(dialogName);
dialog.upperPane.NewAndPut(upperData);
```

Update: function(upperData)

Update the current upper record with the specified upper data. This method corresponds to `maconomy::dialogUpdateUpper()`.

Parameters

Name	Type	Description
`upperData`	object	The upper pane data with which to update the current record.

Returns

The `dialogReturn` data for the current record.

Throws

Whatever `maconomy::dialogUpdateUpper()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var upperData = { ... };
var upperKey = { ... };
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
dialog.common.Get(upperKey);
dialog.upperPane.Update(upperData);
```

Delete: function()

Deletes the current dialog entry. This method corresponds to `maconomy::dialogDeleteUpper()`.

Returns

The `dialogReturn` object without the dialogData property.

Throws

Whatever `maconomy::dialogDeleteUpper()` might throw.

History

Introduced in package version 1.2.

Example

```
uses mscript::maconomy::dialog(1.2) as Dialog;
var dialogName = ...;
var dialog = Dialog::Card(dialogName);
dialog.common.GetFirst();
dialog.upperPane.delete();
```

GetDialogData: function()

Returns the dialog data for the open record. This function is a shorter way of writing `getData().dialogData`.

Returns

The dialog data property in the current record data object.

Throws

Whatever `getData()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var upperKey = { ... };
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
// Get the record data but don't save it:
dialog.common.Get(upperKey);
// 'dialog.common' has saved the record data internally.
// We can always ask for it when we need it:
var ddata = dialog.common.getDialogData();
```

Lower-Pane Dialog Operations

The `lowerPane` object contains methods operating on the table pane of a card/table dialog. This object uses the `common` object to maintain information about dialog ID, return data, and so forth.

GetData: function()

Gets the dialog data for the table pane of the current dialog record. Returns the `upperPane` object of the dialog data object. This method is a shorter way of writing `common.getDialogData().lowerPane`.

Returns

The `lowerPane` object of the current dialog data object.

Throws

Whatever `common.getDialogData()` might throw.

History

Introduced in package version 1.2.

Example

```
uses mscript::maconomy::dialog(1.2) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
```

```
dialog.GetFirst();
var lowerData = dialog.lowerPane.getData();
```

New: function(rowNo)

Creates a new empty row in the lower pane at the specified row number. This method corresponds to `maconomy::dialogNewLower()`.

Parameters

Name	Type	Description
`rowNo`	int	Index of the lower pane row to create.

Returns

The `dialogReturn` data of the new row.

Throws

Whatever `maconomy::dialogNewLower()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var upperKey = { ... };
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
dialog.common.Get(upperKey);
// Create a new row at the bottom of the table:
dialog.lowerPane.New(-1);
```

Put: function(lowerData)

Inserts the specified lower pane data into a newly created lower row. This method corresponds to `maconomy::dialogPutLower()`.

Parameters

Name	Type	Description
`lowerData`	object	The lower pane data to put.

Returns

The `dialogReturn` data for the current row.

Throws

Whatever `maconomy::dialogPutLower()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var lowerData = { ... };
var upperKey = { ... };
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
dialog.common.Get(upperKey);
// Create a new row at the bottom of the table:
dialog.lowerPane.New(-1);
```

NewAndPut: function(lowerData, rowNo)

Creates a row in the lower pane at the specified row number and with the specified lower pane data. This method is a shorter way of performing a `newLower-putLower` sequence on the lower pane.

Parameters

Name	Type	Description
`lowerData`	object	The lower pane data, including a full upper key.
`rowNo`	int	Index of the lower pane row to create.

Returns

The `dialogReturn` data for the current record.

Throws

Whatever `maconomy::dialogNewLower()` or `maconomy::dialogPutLower()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var lowerData = { ... };
var upperKey = { ... };
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
dialog.common.Get(upperKey);
// Create a new row at the bottom of the table:
dialog.lowerPane.NewAndPut(-1, lowerData);
```

Update: function(lowerData, rowKey)

Update the first lower row that matches the row key `rowKey` with the lower pane data `lowerData`. The row key can simply be a row index, or a key object that each row is matched against. This method corresponds to maconomy::dialogUpdateLower().

Parameters

Name	Type	Description
`lowerData`	object	The lower pane data with which to update the current record.
`rowKey`	object or int	The key of the lower pane row to update.

Returns

The `dialogReturn` data for the current record.

Throws

Whatever `maconomy::dialogUpdateLower()` might throw.

History

All package versions.

Example

```
uses mscript::maconomy::dialog(1) as Dialog;
var lowerData = { ... };
var upperKey = { ... };
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
dialog.common.Get(upperKey);
// Update the first row in the table:
dialog.lowerPane.Update(lowerData, 0);
```

Delete: function(rowKey)

Deletes a lower pane row in a dialog. The parameter `rowKey` can be either a row index or a key object matching at least one row in the lower pane. This method corresponds to `maconomy::dialogDeleteLower()`.

Parameters

Name	Type	Description
`rowKey`	object or int	Row key of the row to delete.

Returns

The `dialogReturn` object without the `dialogData` property.

Throws

Whatever `maconomy::dialogDeleteLower()` might throw.

History

Introduced in package version 1.2.

Example

```
uses mscript::maconomy::dialog(1.2) as Dialog;
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
dialog.common.GetFirst();
var rowKey = { ... };
dialog.lowerPane.Delete(rowKey);
```

Algorithms

This reference section describes the `mscript::algorithms` packages. These packages contain utility functions for performing common tasks on values and data structures.

- **`mscript::algoritms::find`** — Contains some basic functions for searching through M-Script objects and arrays for keys or key-value pairs.
- **`mscript::algorithms::mappings`** — Contains some functions for mapping the elements of M-Script objects and arrays.
- **`mscript::algorithms::text`** — Contains some functions for manipulating text strings.

mscript::algorithms::find

function find(value, fields)

This function searches through the structure `fields` and returns the key value of the first field that matches `value`. The structure can be an object or an array, in which case the return value is the property label or the index, respectively.

Comparison is done using the <=>operator so even compound values can be found with this function. If no match is found, `null` is returned.

Parameters

Name	Type	Description
`value`	any value	The value to search for.
`fields`	object or array	The structure through which to search for an element matching `value`.

Returns

The property label or array index of the first element in `fields` matching the specified value, or `null` if no matching element is found.

Throws

`error::runtime` if `fields` is not an object or array.

History

All package versions.

Example

```
uses mscript::algorithms::find(1) as Find;
var fields_1 = {a: "foo", b: "bar"};
dumpvalueln(Find::find("bar", fields_1));
var fields_2 = ["abc","def", "ghi"];
dumpvalueln(Find::find("ghi", fields_2));
dumpvalueln(Find::find("foo", fields_2));
-->
    "b"
    2
    null
```

function find_key(key, value, fields)

This function searches through the structure `fields` and returns the key value of the first field matching the key-value pair given by `key` and `value`. The structure can be an object or an array, in which case the return value will be the property label or the index, respectively.

Comparison is done using the <=>operator so even compound values can be found with this function. If no match is found, `null` is returned.

Parameters

Name	Type	Description
`key`	any value	The key part of the key-value pair to search for in the elements of `fields`. If these elements are objects, the key should be a property label. If the elements are arrays, the key should be a valid index into these arrays.
`value`	any value	The value part of the key-value pair to search for.
`fields`	object or array	The structure through which to search for an element matching the key-value pair.

Returns

The property label or array index of the first element in `fields` that has a matching key-value pair, or `null` if no matching element is found.

Throws

`error::runtime` if `fields` or its elements are not either objects or arrays.

History

All package versions.

Example

```
uses mscript::algorithms::find(1) as Find;
var fields_1 = [{x:1, y:2},{y:2},{y:1}];
dumpvalueln(Find::find_key("y", 1, fields_1));
var fields_2 = {a:[0,1,2], b:[1,2,3], c:[2,3,3]};
dumpvalueln(Find::find_key(2, 3, fields_2));
dumpvalueln(Find::find_key(1, 3, fields_2));
-->
    "b"
    2
    null
```

function find_if(match, fields)

This function searches through the structure `fields` and returns the key value of the first field for which the function `match` returns `true`. The structure can be an object or an array, in which case the return value will be the property label or the index, respectively.

The function `match` must take an element of `fields` as a single argument and return a Boolean indicating whether it is a match or not. If no match is found, `null` is returned.

mscript::algorithms::mappings

function map(mapping, fields)

This function iterates through the elements of the structure `fields` and returns a copy of the structure where the function `mapping` has been applied to every element. The original structure remains unchanged.

Parameters

Name	Type	Description
`mapping`	function	Function that takes elements from the structure `fields` and returns the mapped element.
`fields`	object or array	The structure whose elements are to be mapped.

Returns

A new structure of the same type as `fields`, with each element mapped by the `mapping` function.

Throws

`error::runtime` if fields is not an object or array.

History

All package versions.

Example

```
uses mscript::algorithms::mappings(1) as Mappings;
var fields = [0,1,2,3];
dumpvalue(Mappings::map(function(val){return val+1;}, fields));
dumpvalue(fields);
-->
    [1,2,3,4]
    [0,1,2,3]
```

function map_this(mapping, fields)

This function iterates through the elements of the structure `fields` and applies the function `mapping` to every element. The original elements of the structure are overwritten in the process.

Parameters

Name	Type	Description
`mapping`	function	Function that takes elements from the structure fields and returns the mapped element.
`fields`	object or array	The structure whose elements are to be mapped.

Returns

A new structure of the same type as `fields`, with each element mapped by the `mapping` function.

Throws

`error::runtime` if `fields` is not an object or array.

History

All package versions.

Example

```
uses mscript::algorithms::mappings(1) as Mappings;
var fields = [0,1,2,3];
Mappings::map_this(function(val){return val+1;}, fields);
dumpvalue(fields);
-->
    [1,2,3,4]
```

mscript::algorithms::text

function findFirstOf(text, pos, tokens)

This function performs a search in `text` for one of the specified `tokens`, starting at the specified position `pos`. If several of the tokens match, the position of the token starting at the lowest position will be returned.

Parameters

Name	Type	Description
`text`	string	The text to be searched for `tokens`.
`pos`	int	The position in `text` from where the search should start.
`tokens`	array	An array of strings containing the tokens to search for.

Returns

`Null` if none of the tokens were found, otherwise an object with the following properties:

- **`first`** — The position of the first character in the token.
- **`length`** — The length of the matching token.
- **`index`** — The array index of the matching token.

Throws

None.

History

All package versions.

Example

```
uses mscript::algorithms::text(1) as Text;
var text = "This is the text to search in";
var tokens = ["text", "search"];
// Find occurrence of "text" at position 12
dumpvalueln(Text::findFirstOf(text, 0, tokens));
-->
    {
        first:12,
        length:4,
        index:0
    }
```

function findLastOf(text, pos, tokens)

This function performs a reverse search in `text` for one of the specified `tokens`, starting at the specified position `pos`. If several of the tokens match, the position of the token starting at the highest position is returned.

Parameters

Name	Type	Description
`text`	string	The text to be searched for `tokens`.
`pos`	int	The position in `text` from where the search should start.
`tokens`	array	An array of strings containing the tokens to search for.

Returns

`Null` if none of the tokens were found, otherwise an object with the following properties:

- **`first`** — The position of the first character in the token.
- **`length`** — The length of the matching token.
- **`index`** — The array index of the matching token.

Throws

None.

History

All package versions.

Example

```
uses mscript::algorithms::text(1) as Text;
var text = "This is the text to search in";
var tokens = ["text", "search"];
// Find occurrence of "search" at position 20
dumpvalueln(Text::findLastOf(text, 0, tokens));
-->
    {
        first:20,
        length:6,
        index:1
    }
```

function wrap(text, lineLength, blankChars, delimChars, newlineToken, ...)

This function wraps the specified `text` to a number of lines of a maximum length `lineLength`. Individual words are split if they contain one of the specified delimiter characters and they are longer than the specified line length. Existing line breaks in the text are taken into account when calculating the position of the new line breaks.

Parameters

Name	Type	Description
`text`	string	The text to be wrapped.
`lineLength`	int	The maximum line length of the wrapped text.
`blankChars`	string	A string consisting of all the characters that should be interpreted as blanks, that is, characters on which the text can be wrapped. A line break may replace a blank.
`delimChars`	string	A string consisting of all the characters that should be interpreted as delimiters, that is, characters on which the text can be wrapped. A line break may occur immediately after a delimiter.
`newlineToken`	string	The text that is inserted to create a line break, the new line character.
`...`	string	Additional string arguments with newline tokens that—if present in the source text—should be taken into account when calculating the new line breaks.

Returns

The wrapped text.

Throws

None.

History

All package versions.

Example

```
uses mscript::algorithms::text(1) as Text;
var text = 'TXT'
This is a long text which should be wrapped. This is a long text which
   should be wrapped. This is a long text which should be wrapped.
This is a long text which should be wrapped. This is a long text which
   should be wrapped.
TXT;
var lineLength = 40; // Break after at most 40 characters.
var blankChars = " \n"; // Space and existing newline characters are
                        // blanks.
var delimChars = "-"; // Break on hyphens. (not used in this example)
var newlineToken = "\n"; // Insert newline character to break lines.
```

```
       print("+---------+---------+---------+---------+\n"); // 40 characters.
       print("^1", Text::wrap(text, lineLength, blankChars, delimChars,
              newlineToken));
-->
   +---------+---------+---------+---------+
   This is a long text which should be
   wrapped. This is a long text which
   should be wrapped. This is a long text
   which should be wrapped.
   This is a long text which should be
   wrapped. This is a long text which
   should be wrapped.
```

function wrapText(text, lineLength)

This function corresponds to calling the wrap function with the following values for the arguments `blankChars`, `delimChars`, and `newlineToken`:

- **`blankChars`** — " \t\n\r"
- **delimChars** — "-_.,:;"
- newlineToken — "\n"
- Additional `newlineTokens` — "\r"

Calling this function wraps the specified text to a number of lines of a maximum length `lineLength`.

Parameters

Name	Type	Description
`text`	string	The text to be wrapped.
`lineLength`	int	The maximum line length of the wrapped text.

Returns

The wrapped text.

Throws

None.

History

All package versions.

Example

```
uses mscript::algorithms::text(1) as Text;
var text = 'TXT'
This is a long text which should be wrapped. This is a long text which
   should be wrapped. This is a long text which should be wrapped.
```

```
This is a long text which should be wrapped. This is a long text which
    should be wrapped.
TXT;
var lineLength = 40; // Break after at most 40 characters.
print("+---------+---------+---------+---------+\n"); // 40 characters.
print("^1", Text::wrapText(text, lineLength));
-->
    +---------+---------+---------+---------+
    This is a long text which should be
    wrapped. This is a long text which
    should be wrapped. This is a long text
    which should be wrapped.
    This is a long text which should be
    wrapped. This is a long text which
    should be wrapped.
```

function wrapHTML(text, lineLength)

This function corresponds to calling the wrap with the following values for the arguments `blankChars`, `delimChars`, and `newlineToken`:

- **blankChars** — " \t\n\r"
- **delimChars** — "-_.,:;"
- newlineToken — "
"
- Additional newlineTokens — "
"

Calling this function wraps the specified `text` to a number of lines of a maximum length `lineLength` by adding HTML line breaks (`<br>`).

Parameters

Name	Type	Description
`text`	string	The text to be wrapped.
`lineLength`	int	The maximum line length of the wrapped text.

Returns

The wrapped text.

Throws

None.

History

All package versions.

Example

```
uses mscript::algorithms::text(1) as Text;
```

```
var text = 'TXT'
This is a long text which should be wrapped. This is a long text which
   should be wrapped. This is a long text which should be wrapped.<br />
This is a long text which should be wrapped. This is a long text
   which<br>should be wrapped.
TXT;
var lineLength = 40; // Break after at most 40 characters.
print("^1", Text::wrapHTML(text, lineLength));
-->
   This is a long text which should be<br>wrapped. This is a long text
   which<br> should be wrapped. This is a long text<br>which should be
   wrapped.<br />This is long text which should be<br>wrapped. This is
   a<br>long text which should be wrapped
```

Integration Framework

This reference section describes version 1 of the M-Script Integration Framework. The integration framework is part of the official M-Script package collection distributed with M-Script as of Maconomy version 7.1.

The aim of the integration framework is to offer a collection of components which can be used as building blocks for writing scripts dealing with the import, export and manipulation of data to/from the Maconomy application.

Basic Concepts

The integration framework is built around a few simple concepts. With the primary focus being on data import and export, the framework is centered on data and the flow of data from one location to another.

This flow is represented by pipes, which are used to connect data sources with data sinks. Any component capable of writing data to a pipe is a source in this respect, and in the same way any component capable of reading data from a pipe is a sink. Components capable of doing both are referred to as filters.

The following subsections address each of these concepts in more detail.

Pipe

As a metaphor, the pipe notion is straightforward: when you put something in one end, it will come out the other. The most important properties of a pipe are the things that it does not do:

- A pipe does not alter the data flowing through it.
- A pipe does not alter the sequence of the data flowing through it.
- A pipe does not split data received in one write operation, nor does it combine data from separate write operations.

In other words, if you write “ABC” to a pipe, “ABC” is what you get at the other end, not “BAC” or “123.” If you first write “ABC” and then “DEF,” you will not get “ABCDEF” at the other end. Rather, you will first get “ABC” and then “DEF.”

Parameters

The pipe is an object that contains two properties.

Name	Type	Description
`first`	source	A reference to the first component in the pipe (the source component).
`last`	sink	A reference to the last component in the pipe (the sink component).

Pipes are constructed by calling the `Pipe()` function.

Source Components

A source component is capable of writing data to a pipe, regardless of the source of the data.

When a source component has been connected to a pipe, you can ask it to write its data onto that pipe. It may choose to do this in one big chunk or in smaller pieces, depending on its intended purpose. It may also choose not to write anything at all.

Parameters

A source component is an object with the following properties:

Name	Type	Description
`sink`	sink	A reference to the next component in the pipe.
`send`	proc(data)	Call this function to send `data` to the next component in the pipe.
`execute`	proc([arg])	The method that is called when the pipe is executed.
`finish`	procedure()	The method that must be called when there is no more data.

A default source component can be obtained by calling the `NewSource` constructor function. Predefined source components are located in the packages beneath the package root `integration::sources`.

Sink Components

A sink component is capable of reading data from a pipe. It continues to read from the pipe until it is notified by the source component that no further data will be sent.

A sink component is free to do whatever it wants with the data it receives. It may simply forget about it, or write it to some external storage such as a file or a network stream.

Parameters

A sink component is an object with the following properties.

Name	Type	Description
`receive`	proc(data)	The method which is called when the

Name	Type	Description
		connected pipe has `data` available.
`done`	procedure()	The method which must be called when there is no more data.

A default sink component can be obtained by calling the `NewSink` constructor function. Sink components are located in the packages beneath the package root `integration::sinks`.

Filter Components

A filter component shares traits with both sources and sinks; it is capable of reading data from a pipe and of writing data to a pipe. Unlike sources and sinks, the source and target of the data are well defined.

In a filter, the data read from one pipe is the basis for the data written to the other pipe. Within the filter, the data may, however, have been reorganized, altered, or, as the name suggests, filtered.

Parameters

A filter component is an object with the following properties.

Name	Type	Description
`sink`	sink	A reference to the next component in the pipe.
`send`	proc(data)	Call this function to send `data` to the next component in the pipe.
receive	***proc(data)***	***The method that is called when the connected pipe has data available.***
`execute`	proc([arg])	The method that is called when the pipe is executed.
`done`	procedure()	The method that must be called when there is no more data.
`finish`	procedure()	The method that must be called when there is no more data.

A default filter component can be obtained by calling the `NewFilter` constructor function. Filter components are located in the packages beneath the package root `integration::filters`.

General Utilities — mscript::integration

This section describes some general utilities that form the foundation for the M-Script Integration Framework. These utilities are declared in the package `mscript::integration`.

function Pipe(source,..., sink)

The `Pipe` constructor function takes any number of component arguments, and returns a `pipe` object. The first argument must have the source component signature, and the last argument

must have the sink component signature. All intermediate components must have the filter component signature. Since the signature of a filter component is compatible with the signatures of source as well as sink components, the first and last arguments can also be filter components.

Parameters

Name	Type	Description
`source`	source	The source component that should be at the head of the pipe.
`...`	filters	Any number of additional filter components that should be in the pipe.
`sink`	sink	The sink component that should be at the tail of the pipe.

Returns

A `pipe` object. if invoked with no arguments, `null` is returned.

Throws

`integration::Pipe({message: string})` is thrown in case of an error.

History

All package versions.

Example

```
#version 14
uses mscript::integration(1) as Integration;
// Create three components:
var mySource  = ...;
var myFilter = ...;
var mySink    = ...;
// Connect the components in a pipe:
var myPipe = Integration::Pipe(
   mySource,
   myFilter,
          mySink
      );
```

function Execute(pipe [, arg])

The `Execute` function takes a `pipe` object and optionally an argument, which should be passed to the source component. It then executes the pipe by invoking the `execute` method on the first component in the pipe.

If the last component in the pipe is not a sink component, an extra pipe is connected to this last component, and the data is collected and returned in an array. Otherwise `null` is returned.

Parameters

Name	Type	Description
`pipe`	a pipe object	The pipe that should be executed.
`arg`	any value	An optional argument that should be passed to the `execute` method of the first component in the pipe.

Returns

If the last component in the pipe is a sink component, `null` is returned. Otherwise, an array containing the values collected from the last component in the sink is returned.

Throws

`integration::Execute({message: string})` is thrown if the pipe is somehow invalid.

Any exception thrown by the components in the pipe.

History

All package versions.

Example

```
...
var myPipe = Integration::Pipe(...);
// Execute the pipe:
Integration::Execute(myPipe);
```

function Send(pipe [, val, ...])

The `Send` function takes a `pipe` object and zero or more values that should be piped to the first component in the pipe. This component must hence be a filter or a sink.

If the last component in the pipe is not a sink component, an extra pipe is connected to this last component and the data is collected and returned in an array. Otherwise, `null` is returned.

This function does not send the end-of-data message to the component, so remember to call `Finish` when all data has been sent. Some filter and sink components await this message before performing final processing of the input data.

Parameters

Name	Type	Description
`pipe`	pipe object	The pipe that should receive the values.
`val, ...`	any type	Zero or more values that should be sent to the pipe.

Returns

If the last component in the pipe is a sink component, `null` is returned. Otherwise, an array containing the values collected from the last component in the sink is returned.

Throws

`integration::Send({message: string})` is thrown if the pipe is somehow invalid.

Any exception thrown by the components in the pipe.

History

Version 1.1 and on.

Example

```
uses mscript::integration(1.1) as Integration;

var pipe = Integration::Pipe(...);
...
// Send the values val_1...val_n to the pipe:
Integration::Send(pipe, val_1, ..., val_n);
...
Integration::Finish(pipe);
```

function Finish(pipe)

The `Finish` function sends the end-of-data message to the first component in the specified pipe. The only times that you need to call this function is when using `Send`, or when a forced shutdown of a pipe is necessary.

If the last component in the pipe is not a sink component, an extra pipe is connected to this last component and the data collected and returned in an array. Otherwise, `null` is returned.

Parameters

Name	Type	Description
`pipe`	pipe object	The pipe that should receive the values.

Returns

If the last component in the pipe is a sink component, `null` is returned. Otherwise, an array containing the values collected from the last component in the sink is returned.

Throws

`integration::Finish({message: string})` is thrown if the pipe is somehow invalid.

History

Version 1.1 and on.

Example

```
uses mscript::integration(1.1) as Integration;
Var pipe = Integration::Pipe(...);
...
// Send the values val_1...val_n to the pipe:
```

```
Integration::Send(pipe, val_1, ..., val_n);
...
Integration::Finish(pipe);
```

function NewSource([options])

The `NewSource` constructor function returns an empty source component. The returned component can then be extended to implement a new source component. The empty source component contains no data, and will simply send the end-of-data message to the connected filter or sink component. To extend this behavior you will have to redefine the execute method to something meaningful.

Parameters

Name	Type	Description
`options`	object	An optional argument containing initial values for one or more properties in the component.

Returns

A new source component.

Throws

`init::Initialize({message: string})` if initialization of the component fails.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
var mySource = Integration::NewSource() +
  {
     execute: procedure()
     {
       // Send some integer numbers:
       for (var i = 1; i <= 10; ++i)
     }
     print("Source:\tSending data:\t^1\n", i);
     this.send(i);
     }
     print("Source:\tDone\n");
     // Tell the next component we're done:
     this.finish();
   }
```

```
};
```

function NewSink([options])

The `NewSink` constructor function returns an empty sink component. The returned component can then be extended to implement a new sink component.

The empty sink component does not know how to process the data it receives, and simply forgets about it. To extend this behavior you must redefine the `receive` method to something meaningful. If you need to perform special actions at shutdown, too, you must also redefine the `done` method.

Parameters

Name	Type	Description
`options`	object	An optional argument containing initial values for one or more properties in the component.

Returns

A new sink component.

Throws

`init::Initialize({message: string})` if initialization of the component fails.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
var mySink = Integration::NewSink() +
  {
     receive: procedure(value)
     {
        print("Sink:\tReceiving data:\t^1\n", value);
     },
     done: procedure()
     {
        print("Sink:\tDone\n");
     }
   };
```

function NewFilter([options])

The `NewFilter` constructor function returns an empty filter component. The returned component can then be extended to implement a new filter component.

The empty filter component passes the data it receives unchanged on to the next component in the pipe. To extend this behavior you must redefine the `receive` method to something meaningful. If you need to perform special actions at shutdown too, you must also redefine the `done` method

Parameters

Name	Type	Description
`options`	object	An optional argument containing initial values for one or more properties in the component.

Returns

A new filter component.

Throws

`init::Initialize({message: string})` if initialization of the component fails.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
var myFilter = Integration::NewFilter() +
  {
    receive: procedure(value)
    {
       print("Filter:\tReceiving data:\t^1\n", value);
       // Send on a modified value:
       this.send(10-value);
    },
    done: procedure()
    {
       print("Filter:\tDone\n");
       // Remember to tell the next component:
       this.finish();
     }
  };
```

Putting it Together

If we combine all of this example code we get a complete pipe consisting of a source component, one filter component, and a sink component.

```
#version 14
uses mscript::integration(1) as Integration;
```

```
// Create three components:
var mySource = Integration::NewSource() +
  {
    execute: procedure()
    {
      // Send some integer numbers:
      for (var i = 1; i <= 10; ++i)
      {
        print("Source:\tSending data:\t^1\n", i);
        this.send(i);
      }
      print("Source:\tDone\n");
      // Tell the next component we're done:
      this.finish();
    }
  };
var myFilter = Integration::NewFilter() +
        {
          receive: procedure(value)
          {
             print("Filter:\tReceiving data:\t^1\n", value);
             // Send on a modified value:
             this.send(10-value);
          },
          done: procedure()
          {
            print("Filter:\tDone\n");
            // Remember to tell the next component:
            this.finish();
           }
         };
     var mySink = Integration::NewSink() +
        {
          receive: procedure(value)
          {
             print("Sink:\tReceiving data:\t^1\n", value);
          },
          done: procedure()
          {
            print("Sink:\tDone\n");
```

```
    }
  };
// Connect the components in a pipe:
var myPipe = Integration::Pipe(
  mySource,
  myFilter,
  mySink
 );
 // Execute the pipe:
Integration::Execute(myPipe);
```

The properties of this pipe are:

- The source component sends the numbers from 1 to 10 into the pipe.
- The filter component echoes the numbers it receives and then sends on 10 less the number received.
- The sink component acknowledges the numbers that it receives, and nothing else. Executing this small pipe thus gives the following output.

```
Source: Sending data:   1
Filter: Receiving data: 1
Sink:   Receiving data: 9
Source: Sending data:   2
Filter: Receiving data: 2
Sink:   Receiving data: 8
Source: Sending data:   3
Filter: Receiving data: 3
Sink:   Receiving data: 7
...
Source: Sending data:   9
Filter: Receiving data: 9
Sink:   Receiving data: 1
Source: Sending data:   10
Filter: Receiving data: 10
Sink:   Receiving data: 0
Source: Done
Filter: Done
```

```
Sink:    Done
```

Note how the communication between the components happens completely synchronously. There is no threading and hence no non-deterministic aspects in this type of pipe.

Source Components

This section describes the predefined source components, which are currently distributed with the integration framework. To make it as easy as possible to find components related to a specific task, the components have been grouped according to their field of operation:

- **`mscript::integration::sources::dialog`** describes components that operate on dialogs in the Maconomy API.
- **`mscript::integration::sources::streams`** describes components that operate on input streams.
- **`mscript::integration::sources::files`** describes components that operate on input files.
- **`mscript::integration::sources::xml`** describes components that operate on XML data.

mscript::integration::sources::dialog

Dialog source components offer a way of letting upper and lower pane data from the Maconomy dialog model flow directly into the integration framework. This functionality is built on top of the dialog object abstraction found in the `mscript::maconomy` packages.

function ReadFromDialog(dialog)

The `ReadFromDialog` constructor function augments a dialog object with source component capabilities. Depending on its type, the dialog will get a lower pane and/or an upper pane source interface. The `upperPane` and `lowerPane` objects within the dialog object can hereafter be used like all other source components, reading upper and lower dialog data from the Maconomy API

When executing an upper pane source component, you must pass an appropriate upper key to it. The dialog window corresponding to this upper key is then opened, and the pane data read in as the data source.

When executing a lower pane source component, you do not have to pass it an upper key if one has already been given through the associated upper pane source component. In this case the dialog window is already open and the lower pane rows will be written to the pipe one by one. If the dialog has not been opened by the upper pane source component, it is allowed to pass the appropriate upper key to the lower pane source component.

Normally you should always pass an upper key when executing an upper pane source component. However, if you know that the particular dialog already has an appropriate upper key you can force the source component to reuse that key by passing it `null` as the upper key.

Parameters

Name	Type	Description
`dialog`	dialog object	The dialog object that should be augmented with source component interfaces.

Returns

The augmented dialog object.

Throws

`integration::source::dialog::ReadFromDialog({message: string})` if dialog is not a valid dialog object. This exception type is also thrown by the generated source component in case of an error.

History

All package versions.

Example

```
#version 14
uses mscript::integration(1) as Integration;
uses mscript::integration::sources::dialog(1) as DialogSrc;
uses mscript::maconomy::dialog(1) as Dialog;
// First create a card/table dialog object:
var dialogName = ...;
var dialog = Dialog::CardTable(dialogName);
// Then add source components capabilities to it:
DialogSrc::ReadFromDialog(dialog);
// Now we can use the panes in dialog as source components:
var myUpperPipe = Integration::Pipe(
  dialog.upperPane,
...
);
var myLowerPipe = Integration::Pipe(
dialog.lowerPane,
...
);
// Execute the upper pipe on the given upper key:
var upperKey = { ... };
Integration::Execute(myUpperPipe, upperKey);
// Execute the lower pipe on the same upper key:
Integration::Execute(myLowerPipe);
// Execute the lower pipe on another upper key:
var upperKey2 = { ... };
Integration::Execute(myLowerPipe, upperKey2);
// Execute the upper pipe on this new upper key:
Integration::Execute(myUpperPipe, null);
```

mscript::integration::sources::streams

This package contains basic source components for reading data from an input stream and writing it to the pipe.

function ReadFromStream(istream [, options])

The `ReadFromStream` constructor function returns a source component, which reads one line at a time from the specified input stream, and writes the lines as strings to the pipe. The line delimiter is '`\n`' by default, but can be changed by setting the property `delim` to another single character in the optional argument object `options`.

Parameters

Name	Type	Description
`istream`	input stream	The input stream to which the source component should be connected.
`options`	object	An optional argument containing initial values for one or more properties in the component.

The `options` parameter supports the following property.

Name	Type	Description
`delim`	string	The line delimiter to use when reading from the input stream.

Returns

A new source component.

Throws

`init::Initialize({message: string})` is thrown if the optional argument is invalid.

`integration::sources::file::ReadFromStream({message: string})` is thrown by the resulting source component in case of an error during execution.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::sources::streams(1) as StreamSrc;
// Test input:
var istream = io::istring("This is a \n test");
// Construct the component with space as the delimiter:
var reader = StreamSrc::ReadFromStream(istream, {delim:" "});
var result = Integration::Execute(Integration::Pipe(reader));
```

```
dumpvalue(result, {escape:true});
-->
  [ "This", "is", "a", "\n", "test" ]
```

function ReadFromTabStream(istream [, options])

The `ReadFromTabStream` constructor function returns a source component, which reads tabular data from the specified input stream and writes each line as a string array to the pipe. The column delimiter is '`\t`' by default, but can be changed by setting the property `delim` to another single character in the optional argument object `options`.

Parameters

Name	Type	Description
`istream`	input stream	The input stream to which the source component should be connected.
`options`	object	An optional argument containing initial values for one or more properties in the component.

The `options` parameter supports the following property.

Name	Type	Description
`delim`	string	The column delimiter to use when reading from the input stream.

Returns

A new source component.

Throws

`init::Initialize({message: string})` is thrown if the optional argument is invalid.

`integration::sources::file::ReadFromTabStream({message: string})` is thrown by the resulting source component in case of an error during execution.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::sources::streams(1) as StreamSrc;
// Tab-delimited test input:
var istream = io::istring(TAB);
A  B  C
D  E  F
TAB
```

```
// Construct the component with the default delimit ("\t"):
var reader = StreamSrc::ReadFromTabStream(istream);
var result = Integration::Execute(Integration::Pipe(reader));
dumpvalue(result);
-->
  [ ["A","B","C"], ["D","E","F"] ]
```

mscript::integration::sources::files

File source components read data from files and write it to the pipe. Both `ReadFromFile` and `ReadFromTabFile` described in the following are merely convenient wrappers around the basic stream source components for the special case where the stream is connected to a file.

function ReadFromFile(filename, mode [, options])

The `ReadFromFile` constructor function attempts to open the file `filename` with the specified `mode`. If this succeeds, the resulting file stream is handed to an instance of the function `ReadFromStream(istream [, options])` component which is then returned. This component automatically closes the file when all data has been read. The mode parameter corresponds to the second parameter to `file::open` and should at least contain `"r."`

Parameters

Name	Type	Description
`filename`	string	The name of the file to read from.
`mode`	string	The file mode to use when opening the file.
`options`	object	An optional argument containing initial values for one or more properties in the component. The supported properties are as described for the function `ReadFromStream(istream [, options])`.

Returns

A new source component.

Throws

The exceptions thrown by file::open and function `ReadFromStream(istream [, options])`.

History

All package versions.

Example

```
uses mscript::Integration(1) as Integration;
uses mscript::integration::sources::files(1) as FileSrc;
var filename = ...;
```

```
var reader = FileSrc::ReadFromFile(filename, "r");
...
```

ReadFromTabFile(filename, mode [, options])

The `ReadFromTabFile` constructor function attempts to open the file `filename` with the specified `mode`. If this succeeds, the resulting file stream is handed to an instance of the function `ReadFromTabStream(istream [, options])` component, which is then returned. This component automatically closes the file when all data has been read. The options parameter corresponds to the second parameter to `file::open` and should at least contain `"r."`

Parameters

Name	Type	Description
`filename`	string	The name of the file to read from.
`mode`	string	The file mode to use when opening the file.
`options`	object	An optional argument containing initial values for one or more properties in the component. The supported properties are as described for function `ReadFromTabStream(istream [, options])`.

Returns

A new source component.

Throws

The exceptions thrown by `file::open` and function `ReadFromTabStream(istream [, options])`.

History

All package versions.

Example

```
uses mscript::Integration(1) as Integration;
uses mscript::integration::sources::files(1) as FileSrc;
var filename = ...;
var reader = FileSrc::ReadFromTabFile(filename, "r");
             ...
```

mscript::integration::sources::xml

XML source components parse XML data from an input stream, and write it to the pipe.

DOMParse(istream [, options])

The `DOMParse` constructor function returns a source component, which reads XML-formatted data from the specified input stream `istream` and writes the resulting M-Script DOM object to the pipe.

Because the XML standard allows just one top-level tag, at most one such DOM object is written to the pipe.

By default, leading and trailing whitespace between the tags is stripped from the resulting object, but this can be changed by passing the additional argument `filter` to `DOMParse` in the optional second parameter. The value of `filter` should be a function that takes an M-Script DOM object as a parameter and returns the filtered DOM object.

Parameters

Name	Type	Description
`istream`	input stream	The input stream from which the XML data should be read.
`options`	object	An optional argument containing initial values for one or more properties in the component.

The `options` parameter supports the following property.

Name	Type	Description
`filter`	function(obj)	A function that should be used to filter the XML data. As a parameter it should take an M-Script DOM object, and it should return the filtered DOM object.

History

All package versions.

throws

`init::Initialize({message: string})` in case of an error while creating the source component.

The exceptions from `xml::DOMParse` function.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::sources::xml(1) as XML;
// First specify some XML input:
var istream = io::istring(XML);
  <?xml version="1.0"?>
  <character>
    <name first="Wile E."
          Last=”Coyote” />
    <address road="Route 66, Southwest Desert"
             State=”AZ”
```

```
        Country="USA" />
    <extra>
      <dateOfBirth>September 16, 1949</dateOfBirth>
      <height>55" plus ears</height>
      <weight>115 lbs.</weight>
    </extra>
  </character>
XML
// Execute DOMParse with the default filter:
dumpvalue(
   Integration:Execute(
     Integration::Pipe(
       XML::DOMParse(istream)
   )
 )
);
-->
  [
    { type:"tag", name:"character", attrib:{}, content:
      [
        { type:"tag", name:"name", attrib:
            { first:"Wile E.", last: "Coyote" }, content:[]
        }
       ],
       ...
     }
   ]
```

Sink Components

This section describes the predefined sink components, which are currently distributed with the integration framework. To make it as easy as possible to find components that are related to a specific task, the components have been grouped according to their field of operation:

- Components that operate on dialogs in the Maconomy API.
- Components that operate on output streams.
- Components that operate on output files.
- Components that operate on XML data.

mscript::integration::sinks::dialog

Dialog sink components offer a way of letting data in the integration framework flow directly into the upper and lower panes of the Maconomy dialog model. This functionality is built on top of the dialog object abstraction found in the `mscript::maconomy` packages.

function WriteToDialog(dialog)

The `WriteFromDialog` constructor function augments a dialog object with sink component capabilities. Depending on its type, the dialog gets a lower pane and/or an upper pane sink interface. The `upperPane` and `lowerPane` objects within the dialog object can hereafter be used like all other sink components, writing upper and lower dialog data to the Maconomy API. Both pane sink components expect to receive valid dialog data rows as specified in the M-Script Maconomy API Reference.

When a dialog row is received by an upper pane sink component it first checks that the data includes a full upper key as specified by the dialog definition. It then uses this key to either create a dialog record or to update an existing dialog record with the received dialog row data.

Similarly, the lower pane sink component tries to insert the received rows in the lower pane of the specified dialog record. If the row includes a complete lower pane key, the component may try to update an existing lower pane row; otherwise, it inserts a new row.

The exact behavior of both upper and lower sink components can be configured to one of four strategies:

- **`New`** — Insert a new row only. If the pane key already exists the insert will fail.
- **`New_Update`** — First try to insert row as new. If the pane key already exists, update the existing row instead.
- **`Update`** — Update existing row only. If the pane key does not exist, the insert will fail.
- **`Update_New`** — First try to update existing row. If the pane key does not exist, insert new row instead.

The default strategy is `New_Update` for the upper pane and `Update_New` for the lower pane. Unless you know what you are doing there is no reason to change this. It can however be possible to tweak performance by changing strategy, but before doing so, a few things must be brought to mind:

- When inserting rows that do not contain a complete pane key, the `update` operations are always omitted with only a small run-time overhead. When inserting new rows with a complete pane key, the `Update_New` strategy requires an extra dialog operation per row and thus infers a performance penalty. On the other hand, using `New_Update` when the row already exists infers a similar performance overhead.
- Special attention should be paid to dialog panes for which “special” rules apply to the pane key (such as the lower pane in Time Sheets). Using the `New_Update` strategy on such panes does not give the expected behavior because the lower pane key of existing rows (line number) changes to make room for new rows with the same key. In these situations, `Update` and `Update_New` are the only strategies that will result in existing rows with matching pane keys to be updated.
- The strategy for a given dialog pane sink component can be changed by assigning the new strategy to the strategy property found in the pane object. The four possible strategies are defined in a public enumeration in the package.

Name	Type	Description
`dialog`	dialog object	The dialog object that should be augmented with sink component interfaces.

Returns

The augmented dialog object.

Throws

`integration::sinks::dialog::WriteToDialog({message: string})` if dialog is not a valid dialog object.

integration::sinks::dialog::upperPane({message: string}) and integration::sinks::dialog::lowerPane({message: string}) on errors while writing data to the pane sink components. Possible errors include mismatch in dialog data and insert strategy; for example, writing a new upper row with the `Update` strategy.

History

All package versions.

Example

```
#version 7
uses mscript::integration(1) as Integration;
uses mscript::integration::sinks::dialog(1) as DialogSink;
uses mscript::maconomy(1) as Maconomy;
uses mscript::maconomy::dialog(1) as Dialog;
// First create server and card/table dialog object:
var dialogName = ...;
var userName = ...;
var passwd = ...;
var server = Maconomy::Server();
var dialog = server.CardTable(dialogName);
// Then add sink component capabilities to it:
DialogSink::WriteToDialog(dialog);
// Change the upper pane strategy to 'New':
dialog.upperPane.strategy = DialogSink::New;
// Now we can use the panes in 'dialog' as sink components:
var myUpperPipe = Integration::Pipe(
  dialog.upperPane
);
var myLowerPipe = Integration::Pipe(
  dialog.lowerPane
);
```

```
// Login on the server:
server.login(username, passwd);
// Send a row to the upper pipe:
var upperRow = { ... };
try
    Integration::Send(myUpperPipe, upperRow);
catch integration::sinks::dialog::upperPane()
  print("Upper row already exists!\n");
// Send a row to the lower pipe:
var lowerRow = { ... };
Integration::Send(myLowerPipe, lowerRow);
// Logout on the server:
server.logout();
```

mscript::integration::sinks::streams

This package contains basic sink components for writing data to an output stream.

function PrintToStream(ostream [, options])

The `PrintToStream` constructor function returns a sink component, which attempts to write the data it receives from the pipe on the specified output stream `ostream`. Each data item is followed by a delimiter string, which by default is `"\n"` but can be changed by specifying the additional argument `delim` in the optional second parameter `options`.

Parameters

Name	Type	Description
`ostream`	output stream	The output stream to which the data should be written.
`options`	object	An optional argument containing initial values for one or more properties in the component.

The `options` parameter supports the following properties.

Name	Type	Description
`delim`	string	The delimiter that should be written after each data item. The default delimiter is a single new line character.

Returns

A new sink component.

Throws

`init::Initialize({message: string})` in case of an error while creating the sink component.

`integration::sinks::streams::PrintToStream({message: string})` in case of an error while printing a value to the stream.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::sinks::streams(1) as StreamSink;
uses mscript::integration::sources::streams(1) as StreamSource;
// First specify an input text stream:
var istream = io::istring("This is a test");
// Use stdout as the output stream:
var ostream = io::stdout();
// Create a pipe which reads the input text and writes each
// word to the stream sink component:
var pipe = Integration::Pipe(
  Streamsource::ReadFromStream(istream, {delim:" "}),
  StreamSink::PrintToStream(ostream, {delim:" * "})
);
// Execute the pipe:
Integration::Execute(pipe);
-->
  This * is * a * test
```

function DumpToStream(ostream)

The `DumpToStream` constructor function returns a sink component, which will dump the data it receives on the specified output stream `ostream`. The data is dumped using the built-in `file::dumpvalue` function without padding or extra delimiters.

Parameters

Name	Type	Description
`ostream`	output stream	The output stream to which the data should be dumped.

Returns

A new sink component.

Throws

`init::Initialize({message: string})` in case of an error while creating the sink component.

`integration::sinks::streams::DumpToStream({message: string})` in case of an error while dumping a value to the stream.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::sinks::streams(1) as StreamSink;
// First specify some data to dump:
var data = { a: "Hello world", b: 3.1415 };
// Dump the data on stdout via the DumpToStream component:
Integration::Send(
  Integration::Pipe(
    Streamsink::DumpToStream(io::stdout())
  ), data
);
-->
  {
     a:"Hello world",
     b:3.1415
  }
```

function PrintToTabStream(ostream [, options])

The `PrintToTabStream` constructor function returns a sink component, which writes arrays as tabular data on the specified output stream `ostream`. The default column delimiter is `"\t"` but this can be changed by specifying another single character in the additional argument `delim` in the optional second parameter `options`.

Parameters

Name	Type	Description
`ostream`	output stream	The output stream to which the data should be written.
`options`	object	An optional argument containing initial values for one or more properties in the component.

The options parameter supports the following properties.

Name	Type	Description

Name	Type	Description
`delim`	string	The delimiter that should be written after each data item. The default delimiter is a single tab character.

Returns

A new sink component.

Throws

`init::Initialize({message: string})` in case of an error while creating the sink component.

`integration::sinks::streams::PrintToTabStream({message: string})` in case of an error while printing the tabular values to the stream.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::sinks::streams(1) as StreamSink;
// First specify some tabular data to print:
var tabdata = [ "A", "B", "C" ];
// Use stdout as the output stream:
var ostream = io::stdout();
// Print the data with the default delimiter:
Integration::Send(
  Integration::Pipe(
     Streamsink::PrintToTabStream(ostream)
  ),
  tabdata
);
// Print the data with a comma delimiter:
Integration::Send(
  Integration::Pipe(
    StreamSink::PrintToTabStream(ostream, {delim:", "})
  ),
  tabdata
);
-->
  A       B       C
  A, B, C
```

mscript::integration::sinks::files

File sink components receive data from the pipe and write it to a file. Both `PrintToFile` and `PrintToTabFile` described in the following are merely convenient wrappers around the basic stream sink components for the special case where the stream is connected to a file.

function PrintToFile(filename, mode [, options])

The `PrintToFile` constructor function attempts to open the file `filename` with the specified `mode`. If this succeeds, the resulting file stream is handed to an instance of the `PrintToStream` component, which is then returned. This component automatically closes the file when all data has been written. The `mode` parameter corresponds to the second parameter to `file::open` and should at least contain `"w."` With the `options` parameter you can specify an alternative character to use as data delimiter.

Properties

Name	Type	Description
`filename`	string	The name of the file to write to.
`mode`	string	The file mode to use when opening the file.
`options`	String	An optional argument containing initial values for one or more properties in the component. The supported properties are as described for function `PrintToStream(ostream [, options])`.

Returns

A new sink component.

Throws

The exceptions thrown by `file::open` and `PrintToStream`.

History

All package versions.

Example

```
uses mscript::Integration(1) as Integration;
uses mscript::integration::sinks::files(1) as FileSink;
var filename = ...;
var reader = FileSink::PrintToFile(filename, "w");
...
```

function DumpToFile(filename, mode)

The `DumpToFile` constructor function attempts to open the file `filename` in the specified `mode`. If this succeeds, the resulting file stream is handed to an instance of the `DumpToStream` component, which is then returned. This component automatically closes the file when all data has been written.

Parameters

Name	Type	Description
`filename`	string	The name of the file to write to.
`mode`	string	The file mode to use when opening the file.

Returns

A new sink component.

Throws

The exceptions thrown by `file::open` and `PrintToStream`.

History

All package versions.

Example

```
uses mscript::Integration(1) as Integration;
uses mscript::integration::sinks::files(1) as FileSink;
var filename = ...;
var reader = FileSink::DumpToFile(filename, "w");
...
```

function PrintToTabFile(filename, mode [, options])

The `PrintToTabFile` constructor function attempts to open the file `filename` with the specified `mode`. If this succeeds, the resulting file stream is handed to an instance of the `PrintToTabStream` component, which is then returned. This component will automatically close the file when all data has been written. The `mode` parameter corresponds to the second parameter to `file::open` and should at least contain `"w."` With the `options` parameter you can specify an alternative character to use as data delimiter.

Parameters

Name	Type	Description
`filename`	string	The name of the file to write to.
`mode`	string	The file mode to use when opening the file.
`options`	string	An optional argument containing initial values for one or more properties in the component. The supported properties are as described for function `PrintToTabStream(ostream[, options])`.

Returns

A new sink component.

Throws

The exceptions thrown by `file::open` and `PrintToTabStream`.

History

All package versions.

Example

```
uses mscript::Integration(1) as Integration;
uses mscript::integration::sinks::files(1) as FileSink;
var filename = ...;
var reader = FileSink::PrintToTabFile(filename, "w");
...
```

mscript::integration::sinks::xml

function DOMPrint(ostream [, options])

The `DOMPrint` constructor function returns a component, which accepts zero or more M-Script DOM objects and writes them to the specified output stream `ostream`. The component can be configured to place the received objects in a given context, thus making the output valid XML by ensuring that there is only one top-node. By default no top-node is added, so if more than one DOM objects are passed to the sink, the output will not in itself be valid XML.

To specify a top-node, you should specify the properties `top` and `content` in the optional argument `options`. The value of the `top` property must be a valid M-Script DOM object, while the `content` property must refer to `top.content`. When doing this, the DOM objects are concatenated onto the content array of the top DOM object and this object appears as the top-node of the XML data.

You can also set the top-node after creation by calling the `setTop(topObj)` method in the component.

Parameters

Name	Type	Description
`ostream`	output stream	The output stream to which the data should be written.
`options`	object	An optional argument containing initial values for one or more properties in the component.

The `options` parameter supports the following properties.

Name	Type	Description
`top`	object	The M-Script DOM object that should be

Name	Type	Description
		used as the top node of the XML structure.
`content`	array	The content array into which the DOM objects should be inserted. When the `top` property is specified, this property should be set to `top.content`.

Returns

A new sink component.

Throws

`init::Initialize({message: string})` in case of an error while creating the sink component. Also thrown by the `setTop` method if the passed parameter is not a valid M-Script DOM object.

The exceptions thrown by `xml::DOMPrint` in case of an error while writing XML data to the stream.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::sinks::xml(1) as XMLSink;
uses mscript::integration::filters(1.1) as Filters;
// First specify the DOM objects to print:
var domObjects = [
  "<?xml version=\"1.0\"?>",
  "\n",
  { type:"tag", name:"character", attrib:{},
    Content:
    [
      "\n  ",
        { type:"tag", name:"name", content:[],
          attrib:
            { first:"Wile E.", last: "Coyote" }
        },
      ...
    ]
  }
];
// Use stdout as the ostream:
```

```
var ostream = io::stdout();
// Create the pipe:
var pipe = Integration::Pipe(
  Filters::Split(), // (see section 5.1)
  XMLSink::DOMPrint(ostream)
);
// Send the DOM objects to the pipe:
Integration::Send(pipe,      domObjects);
Integration::Finish(pipe);
-->
  <?xml version="1.0"?>
  <character>
    <name first="Wile E." last="Coyote" />
    ...
  </character>
```

Filter Components

This section describes the predefined filter components, which are currently distributed with the integration framework. To make it as easy as possible to find components related to a specific task, the components have been grouped according to their field of operation:

- Basic components with general applicability.
- Components for transforming the structure of objects and arrays.
- Components for sorting data.
- Components for tokenizing text; that is, extracting structured information from a complex text format.

mscript::integration::filters

This package defines a number of components designed to perform one small task each. These tasks may seem either trivial or irrelevant at first glance, but are intended to reflect operations typically encountered in a solution space. Described briefly, these tasks are:

- **Mappings** — The translation of one value into another.
- **Filtering** — Removal of values satisfying a given criterion.
- **Composition** — Collection components into one value.
- **Decomposition** — Splitting a value into its components.

function Map(mapping)

The `Map` constructor function returns a filter component that applies the specified `mapping` function to all of the values that it receives and sends on the new value that is returned by this function.

Parameters

Name	Type	Description

Name	Type	Description
mapping	function	The function that should be applied to each value that is received by the filter component. It must accept one argument, which is the received value, and return a new value, which is to be sent on.

Returns

A new filter component.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::filters(1.1) as Filters;
// First specify some data and a mapping on this data:
var dataIn  = [ 3 , 2 , 1 , 0 ];
var dataMap = ["a","b","c","d"];
// Create the mapping function:
function mapping(index)
{
  return dataMap[index];
}
// Create the pipe:
var pipe = Integration::Pipe(
  Integration::Filters::Split(),
  Integration::Filters::Map(mapping)
);
// Send the input data to the pipe:
dumpvalue(
  Integration::Send(pipe, dataIn)
);
-->
  [ "d", "c", "b", "a" ]
```

function RemoveIf(match)

The `RemoveIf` function applies the function `match` to each value it receives, and the value is only sent on if the return value from this function is `false`.

Name	Type	Description
`match`	function	The function that should be applied to each value that is received. It must accept one argument, which is the received value, and return `true` if it should be removed, and `false` otherwise.

Returns

A new filter component.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::filters(1.1) as Filters;
// First specify some input data:
var dataIn = [ 4, 7, 2, 8, 5, 6, 0, 1];
// Filter away numbers less than five:
function match(value)
{
  return value < 5;
}
// Create the pipe:
var pipe = Integration::Pipe(
  Filters::Split(),
  Filters::RemoveIf(match)
);
// Send the input data to the pipe:
dumpvalue(
  Integration::Send(pipe, dataIn)
);
-->
  [ 7, 8, 5, 6 ]
```

function Split()

The `Split` constructor function returns a filter component, which iterates through the elements of compound values (objects and arrays) and sends each one on separately. For objects this means losing the property names. Non-compound values are sent on unaltered.

Parameters

None.

Returns

A new filter component.

History

Version 1.1.0 and on.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::filters(1.1) as Filters;
// First specify some input data:
var dataIn = { a: "foo", b: "bar", c: 3.1415 };
// Create the pipe:
var pipe = Integration::Pipe(
  Filter::Split()
);
// Send the input data to the pipe:
dumpvalue(
  Integration::Send(pipe, dataIn)
);
-->
  [ "foo", "bar", 3.1415 ]
```

function Collect()

The `Collect` constructor function collects all of the values that are received in one array, preserving their original sequence. When all values have been received, the array is sent on as one value.

Parameters

None.

Returns

A new filter component.

History

Version 1.1.0 and on.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::filters(1.1) as Filters;
// First specify some input data:
var dataIn = { a: "foo", b: "bar", c: 3.1415 };
```

```
// Create the pipe:
var pipe = Integration::Pipe(
  Filter::Split()
  Filter::Collect()
);
// Send the input data to the pipe:
dumpvalue(
  Integration::Send(pipe, dataIn)
);
-->
  [
   [ "foo", "bar", 3.1415 ]
  ]
```

mscript::integration::filters::transform

This package defines components for transforming the structure of objects and arrays. A transformation is essentially a mapping, except that the transformation is specified for each element within the received values, and hence is only applicable to compound data structures such as objects and arrays. For these structures it does, however, offer an easy way to perform certain types of element-by-element modification with a minimum amount of coding.

Transformations are particularly useful on data structures where each element is unique and requires unique treatment. This includes manipulation of object properties of different types and also extraction of a subset of a compound data structure. In other words, transformations are *not* well suited for batch operations where many elements should be modified in the exact same way. The basic filtering components `Split`, `Map` and `Collect` should be used in these situations.

The two transformation components `TransformToObject` and `TransformToArray` work in much the same way, so to avoid long repetition in the description of these components we start off by introducing the core concepts of how to specify a transformation.

Transformations

A transformation is a specification of a mapping of keys and values from one domain to another. Each mapping is specified by a transformation object. The transformation is a collection of such transformation objects.

The elements in objects and arrays are identified by keys. In an object, the keys are property names, while in arrays they are the indexes of the array elements. In the mapping it is possible to associate individual key values with the transformation object that specifies how this particular key value should be transformed. The transformation object can contain the following properties.

Properties

Name	Type	Description
`key`	string or int	The new key value for this element.
`Value`	func(value)	A mapping function that takes the current value of the element and returns the new

Name	Type	Description
		value of the element.

Either of these properties can be omitted if the key value and/or value should remain unchanged. If specified, the key property should be a string if the transformed value is an object, and an integer index if the transformed value is an array. The expected type of the input values is decided by how these transformation objects are grouped together:

- If the transformation objects are put into an array, the input must also be an array, and in this case the index of each transformation object within the array corresponds to the key value on which that transformation object operates.
- If the transformation objects are put into an object, the input must also be an object, and in this case the property name of each element indicates the key value on which that transformation object operates.

Example

```
// These mappings operate on arrays and return objects
// (index 0 becomes property "foo" etc.):
var oa = [ {key:"foo"}, {key:"bar"} ];
// These mappings operate on objects and return arrays
// (property "foo" becomes index 0 etc.):
var ao = { foo:{key:0}, bar:{key:1} };
// These mappings operate on arrays and return arrays
// (swapping the first two elements):
var aa = [ {key:1}, {key:0} ];
// These mappings operate on objects and return objects
// (changing property "foo" into "oof" etc.):
var oo = { foo:{key:"oof"}, bar:{key:"rab"} };
// These mappings operate on arrays of numbers and perform
// sign change on the first two elements:
function rev_sign(val) { return -val; }
var m = [ {value: rev_sign}, {value: rev_sign} ];
```

Predefined Action Functions

By default, if a key in the input value is not found among the transformation objects, that key and its associated value are transferred unchanged to the output. It is however possible to instead filter away these values by setting the property `undefined` within the transformation component to another value. The property is expected to point to an action function, which takes a single key value argument, and returns either a transformation object or a predefined action code.

The predefined action functions are described in the following table.

Name	Type	Description
`Keep`	function(key)	Returns an empty transformation object indicating that the key value should be transferred unchanged to the result.

Name	Type	Description
`Ignore`	function(key)	Returns the action code `Omit`, which causes the transformation engine to omit that key value from the result.
`Reject`	function(key	Rejects the undefined key value by throwing an exception of the type `integration::filters::transform::Reject({message : string}).`

The default value of the property undefined is the function `Keep`, which for all key values instructs the transformation engine to keep that value. By changing the value to one of the other predefined functions you can modify this aspect of the behavior as described previously. You can also write a new function of the appropriate type and thereby define an entirely different behavior.

function TransformToObject(mappings [, options])

The `TransformToObject` constructor function returns a filter component that is capable of performing modification and filtering on the elements within an object or array, while mapping these elements onto an object structure.

The first argument `mappings` is a collection of transformation objects. Any key fields specified within these objects must be property names of type string. If the `mappings` are from an array structure, each element must contain a property key. In addition, the property `undefined` must be changed to also return a transformation object that contains a property key. The optional second parameter `options` can be used to change the value of the property `undefined`.

Parameters

Name	Type	Description
`mappings`	object or array	A transformation.
`options`	object	An optional argument that contains initial values for one or more properties in the component.

The `options` parameter supports the following properties.

Name	Type	Description
`undefined`	function	An action function.

Returns

A new filter component.

Throws

`init::Initialize({message: string})` in case of an error while creating the filter component.

`integration::filters::transform::Input({message: string})` will be thrown by the component if it encounters an invalid transformation.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::filters::transform(1) as Transform;
// First specify some input data:
var input = { a:3.1415, b:"foo", c:42 };
// Create the pipe:
var pipe = Integration::Pipe(
  Transform::TransformToObject(
     {
     a:{key:"pi"},
     c:{value: function(val) { return val-1; }}
   },
   { undefined: Transform::Ignore }
  )
);
// Send the input data to the pipe:
dumpvalue(Integration::Send(pipe, input)[0]);
// Now change the filter to keep key values w/o mappings:
pipe.first.undefined = Transform::Keep;
// Send the input data to the pipe once more:
dumpvalue(Integration::Send(pipe, input)[0]);
-->
  { pi:3.1415, c:41 }
  { pi:3.1415, b:"foo", c:41 }
```

function TransformToArray(mappings [, options])

The `TransformToArray` constructor function returns a filter component that is capable of performing modification and filtering on the elements within an object or array, while mapping these elements onto an array structure.

The first argument `mappings` is a collection of transformation objects as described previously. Any key fields specified within these objects must be array indexes. If the mappings are from an object structure, each element must contain an array index key. In addition, the property `undefined` must be changed to also return a transformation object containing an array index key. The optional second parameter `options` can be used to change the value of the property `undefined`.

Parameters

Name	Type	Description
`mappings`	object or array	A transformation.
`options`	object	An optional argument that contains initial values for one or more properties in the component.

The `options` parameter supports the following properties.

Name	Type	Description
`undefined`	function	An action function.

Returns

A new filter component.

Throws

`init::Initialize({message: string})` in case of an error while creating the filter component.

`integration::filters::transform::Input({message: string})` is thrown by the component if it encounters an invalid transformation.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::filters::transform(1) as Transform;
// First specify some input data:
var input = { a:3.1415, b:"foo", c:42 };
// Create the pipe:
var pipe = Integration::Pipe(
  Transform::TransformToArray(
      {
      a:{key:0},
      b:{key:1},
      c:{key:3} // skipping index 2
    },
    {
      // Reject additional input keys:
      undefined: Transform::Reject
    }
```

```
  )
);
// Send the input data to the pipe:
dumpvalue(Integration::Send(pipe, input)[0]);
// Try to send an extra key along:
try
  dumpvalue(Integration::Send(pipe, input+{d:0})[0]);
catch integration::filters::transform::Reject(e)
  dumpvalue(e.value.message);
-->
  [ 3.1415, "foo", null, 42 ]
  "no transformation defined for key d"
```

mscript::integration::filters::sort

This package defines filter components for sorting the data received from the pipe according to a given ordering. Unlike most other filter components, the components described here are not order-preserving.

function QuickSort(compare)

The `QuickSort` constructor function returns a filter component, which collects and sorts the received data using the efficient quick-sort algorithm as implemented in `mscript::quicksort(1)`. The data is collected internally until the end-of-data message is received, after which it is sorted and sent on to the pipe.

The parameter compare must be an object with the following property.

Name	Type	Description
`lessThan`	function(a,b)	A function that must accept two arguments and return `true` if the first element is less than the second (which means it must be before the second element. Otherwise, it must return `false`.

Returns

A new filter component.

Throws

`integration::filters::sort::QuickSort` if compare is invalid.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;
uses mscript::integration::filters(1.1) as Filters;
```

```
uses mscript::integration::filters::sort(1) as Sort;
// First specify some input data:
var data = [ 4, 7, 3, 5, 8, 2, 3 ];
// Create the comparison object:
var compare = { lessThan: function(a,b) { return a<b; } };
// Create the pipe:
var pipe = Integration::Pipe(
  Filters::Split(),
  Sort::QuickSort(compare)
);
// Send the input data to the pipe (will return nothing):
dumpvalue(
  Integration::Send(pipe, data)
);
// Send the end-of-data message (will sort the data):
dumpvalue(
  Integration::Finish(pipe)
);
-->
  []
  [ 2, 3, 3, 4, 5, 7, 8 ]
```

mscript::integration::filters::tokenize

This package defines filter components, which take raw text strings as input and extract tokens according to a syntax description.

function GetTokens(patterns [, options])

The `GetTokens` constructor function returns a filter component, which is capable of matching text strings against the specified syntax patterns, and of returning parts of these strings as arrays of string tokens.

The syntax pattern parameter `patterns` is an array of syntax patterns. Each pattern can be either a simple text string or a regular expression as used by M-Script's built-in `regex` package. These patterns represent the substrings of interest in the input text. Each syntax pattern searches forward in the text until a match is found or the text is exhausted. Any number of characters may be skipped by a pattern, and every time a pattern finds its match, the next pattern continues after the text that was matched by the previous pattern. The syntax patterns are greedy; that is, they match as much of the text as possible.

If a syntax pattern is a regular expression that contains subexpressions, the text that is matched by each of these subexpressions becomes separate tokens. For simple text patterns and regular expressions that do not have subexpressions, each pattern results in exactly one token.

There are several ways of modifying the behavior of the component, either through the optional parameter `options` or directly in the syntax pattern declarations.

Name	Type	Description
`patterns`	array	An array of syntax patterns. Each pattern can be either a regular expression or a plain string.
`options`	object	An optional argument that contains initial values for one or more properties in the component.

The `options` parameter supports the following properties.

Name	Type	Description
`fail`, `remain`	function	Action functions that can be customized globally for the component.

Returns

A new filter component.

Throws

`init::Initialize({message: string})` in case of an error while creating the filter component.

`integration::filters::tokenize::GetTokens({message: string})` if the input does not match the syntax patterns.

History

All package versions.

Example

```
uses mscript::integration(1) as Integration;

uses mscript::integration::filters::tokenize(1) as Tokenize;

uses mscript::integration::sources::streams(1) as StreamSource;

// First specify some input: var
input = io::istring(TXT); token_1
token_2 skip LeftRight

more_1  more_2  ignore-all-this Left ignore Right

TXT

// Create patterns for the above input:

var patterns =

[

  /([\w]+_1)[\s]*([\w]+_2)/,

  "Left",

  "Right"

];
```

```
// Create the pipe:
var pipe = Integration::Pipe(
  StreamSource::ReadFromStream(input),
  Tokenize::GetTokens(patterns)
);
// Execute the pipe:
dumpvalue(
  Integration::Execute(pipe)
);
-->
[
  [ "token_1", "token_2", "Left", "Right" ]
  [ "more_1",  "more_2",  "Left", "Right" ]
]
```

Customizing the Behavior of GetTokens

The behavior of the component is customizable through three action functions. Each of these functions has been given a reasonable default implementation, and the package also offers other commonly needed variations, which can be used "out of the box."

The action functions can be changed either in the individual syntax pattern, or globally for the entire component. If specified in the syntax pattern, the functions override the component-global functions for that specific syntax pattern only. The functions are described in the following table.

Name	Type	Description
`match`	function	Action when a syntax pattern finds a match.
`fail`	function	Action when a syntax pattern fails to find a match.
`remain`	function	Action when all syntax patterns have been matched.

Only `fail` and `remain` can be defined globally for the component. All three functions can, however, be defined for the individual syntax pattern. This is done by replacing the simple string or regular expression pattern with an object in which the following properties may be defined.

Name	Type	Description
`pattern`	string or regex	The actual syntax pattern. This property is mandatory.
`Match, fail, remain`		Optional specification of action functions for this syntax pattern.

Predefined Action Functions

The `match` function has default implementations for both string patterns and regular expressions. It is those implementations that move the cursor just beyond the text that is matched by the last syntax pattern. Similarly, `fail` has a default implementation, which simply returns `null`, thereby aborting operations if the pattern fails to match, and `remain` is implemented so that any text beyond the last match is ignored.

The following is an overview of all of the predefined action functions in the package.

Name	Type	Description
`StringMatch`	function	Default implementation of `match` for string patterns.
`RegexMatch`	function	Default implementation of `match` for regular expression patterns.
`RejectFail`	function	Default implement of `fail` that aborts operations if a syntax pattern fails.
`IgnoreFail`	function	Alternative implement of `fail` that skips on to the next syntax pattern.
`IgnoreRemain`	function	Default implementation of `remain` that ignores text beyond a successful match by all syntax patterns.
`RejectRemain`	function	Alternative implementation of `remain` that aborts operations if there is text beyond the final match.

Writing New Action Functions

Normally you should not find a need to move beyond the predefined action functions described previously. Should you decide to venture into this territory anyway, here is a brief introduction to the inner workings of the action functions.

Name	Type	Description
`match`	function(line, cursor, pos, matches)	This function must return a cursor position offset from where to resume matching, or `null` if the match should be disregarded.
`fail`	function(line, cursor, pos, pattern)	This function must return a cursor position offset from where to resume matching, or `null` if matching should not be resumed.

Name	Type	Description
`remain`	function(line, cursor)	This function must return a cursor position offset indicating how much of the remaining input should be disregarded. If this offset does not position the cursor at or beyond the end of the input, the remaining input is considered unmatched by the patterns.

The parameters that occur in the three action functions are described in the following table.

Name	Type	Description
`line`	string	The input currently being matched against.
`cursor`	int	The index in the input from where the match started.
`pos`	int	The index in the input from where the syntax pattern found a match.
`matches`	array of strings	The tokens that were matched by the syntax pattern.
`pattern`	string or regex	The syntax pattern that was attempted.

Maconomy Web Services

The Maconomy Web Services Framework allows M-Script programmers to expose any M-Script package as a web service, including WSDL generation, SOAP handling, and HTTP GET handling.

Background

Web services is an emerging technology that makes it possible for different systems to communicate and exchange data via the Internet. Web services is a “machine-to-machine” language. There is no human “interference,” and there is no visual interface. A web service is a program that other programs can “talk with,” a program that is always available (hence the name “service”), and it is a program that can be accessed via the Internet (hence the name “web”).

Web services use XML, a text-based format, to pack data for sending and receiving. XML is used both for packaging data and to describe a web service. The description makes it possible for external applications to decode (automatically) what the web service can do and how it can be accessed.

The use of web services is quite effective for connecting heterogeneous systems, but when it comes to performance, it is a bit slow. Data must first be packed via XML, then transported via the Internet to a web server, then unpacked from XML again and transformed to M-Script data—and then finally the program itself can be run.

Web Service Vocabulary

Term	Description
HTTP	Hyper Text Transfer Protocol. The standard Internet protocol used for communicating with a web server.
SOAP	Simple Object Access Protocol. A standard for packing document data into an XML structure and distributing it over the Internet.
Web Service	Any program that you can access over the Internet via a web server. This is a very broad definition that includes Google, the Amazon online store, and any other service that you can find on the Internet. However, people usually use the term “web service” to refer to services that use HTTP, SOAP, and WSDL techniques.
WSDL	Web Service Description Language. An XML standard that makes it possible to describe any web service in such detail that tools can create programs from the description.
XML	eXtensible Mark-up Language. A text-based standard for splitting a document into various pieces that are recognizable by a computer program.

Installation

The web service framework is divided into four components:

- A few global packages for M-Script that are located in the namespace mscript::soap (installed by the TPU’s standard M-Script installation)
- The framework files located in <root>/MaconomyWS/Framework
- The actual services located in <root>/MaconomyWS/<application-version>
- The M-Script interpreter MaconomyWS.exe (which is just another copy of the standard M-Script executable)

Here <root> is the root of the web service installation and <application-version> is an identifier for the application version. For a system with Maconomy application W8.0 installed this looks like the following.

```
<root>
   + cgi-bin
      + Maconomy
         - MaconomyWS.exe      // M-Script interpreter
         - MaconomyWS.I        // M-Script initialization file
+ MaconomyWS
   + Framework
      + Services            // Standard framework services
         - wssession.1.ms
         - ...
      - wsdl.ms
      - soap.ms
      - ...
   + W_8_0
      + Standard            // Maconomy standard services here
```

```
      + Solution            // Maconomy solution services here
      + Custom              // Place your custom services here
   - WebServices.I           // Framework initialization file
```

The Standard/Solution/Custom directory structure follows the Maconomy standard of letting different interesting parties of an installation create their own web services.

M-Script Web Services Initialization File

The MaconomyWS.I file must identify the preceding directories like the following.

```
// Points to where the framework files "wsdl.ms", "soap.ms", and
// others are installed.
SearchPath=<root>\MaconomyWS\Framework

// The default file system root can be set to anything,
// the web service framework does not depend on it.
// "MaconomyWS" points to the top directory of the
// Web service installation.
FileSystemRoot=Tmp=c:\tmp;
             MaconomyWS=<root>\MaconomyWS

// The default package root can be set to anything,
// the web service framework does not depend on it.
// "WSApplication" points to the location of the application-specific
// packages. The standard installation uses the "Examples" directory.

// You should change this to suit your needs.
// "WSFramework" points to the web service framework
PackageRoot=C:\AnyDirectory;
             WSApplication=<root>\MaconomyWS\Examples;
             WSFramework=<root>\MaconomyWS\Framework
```

Framework Initialization File

The web service framework initialization files `WebServices.I` contains one M-Script object that has various properties that are used by the framework, which are described in the following table.

Name	Type	Description
log	object	Object with the following properties
.filename	string	Name of log file (relative to the settings in M-Script's initialization file
.enabled	bool	Used to enable and disable logging

Logging

If logging is enabled, all web service activities are logged in an XML structure like the following.

```
<serviceStart> | <serviceDisabled> | <serviceUnknown>
   <service>serviceName</service>
```

```
   <operation>operationName</operation>
   <username>userName</username>
   <date>YYYY-MM-DD</date>
   <time>HH:MM:SS</time>
   <address>IP-Number of client</address>
</serviceStart> | </serviceDisabled> | </serviceUnknown>
```

To ease reading of the XML file with non-XML parsers, the data is placed on one line for each invocation (so that there are no new lines as shown in the preceding example).

The date and time format depend on the settings in M-Script's initialization file.

Writing a Web Service

Writing a web service can be as simple as writing a standard M-Script package and then placing it in a proper directory on the web server. The package itself then becomes the service, and all of the public functions in it become operations on the service. It is almost as simple as that; the only thing to add is a public variable named serviceDescription. This variable must be an object that defines a few global properties of the service.

Consider the following example. The following piece of code is about the smallest complete functional web service anyone can write.

```
#version 13

package minimal(1);

public var serviceDescription =
{
       // The service namespace defines which namespace to put the
       // service data into. Used among other things as the
       // "target namespace" for the WSDL generator
       // Choose a namespace that fits your purpose.
   serviceNamespace: "my.domain.com",

       // A short documentation text inserted as a "<documentation>"
       // tag in the service part of the WSDL
   documentation: "A minimal Web service."
};

       // If no function specific information is supplied then the Web
       // service framework assumes both input and output types are
       // strings. So this "concatenate" function works on strings only.
public function concat(a,b)
{
   return a + b;
}
```

This service package consists of two parts: the global properties of the service (namespace and documentation) and one function, concat, which corresponds to the one and only operation that is allowed on this service.

From a programmer's point of view this is all that is needed. Web service description, WSDL, SOAP, and HTTP GET interfaces are generated automatically from the service package. Encoding and decoding of XML data is handled by the framework.

Assuming that the web service framework is installed on your web server with the base URL http://your.host.com/cgi-bin/Maconomy/MaconomyWS.exe, you can now access your web service description (the WSDL) as http://your.host.com/cgi-bin/Maconomy/MaconomyWS.exe/wsdl.ms?service=minimal. The output is an XML WSDL document that is ready to use with either Microsoft's .NET tools or a similar Java tool set.

Deployment

To expose your M-Script package as a web service you must place it in the <root>/MaconomyWS/<application-version>/Custom directory as described in the installation guide. Moving it into that directory is all that is needed.

Adding a Type System

A problem in the web service framework is that M-Script is dynamically typed, which makes it impossible to derive a proper type safe interface specification of the web service. Hence, some kind of type system must be encoded using the existing features of M-Script. For this, add an operation information object to the service package. This is an optional object that can be added for each function in the package. For a function with the name f the information object must be named fInfo.

The info object can contain properties that define the types of both input and output to the function. The input types are specified by the property inputTypes, which must be an object with one property for each input parameter name. The output type is specified directly in the outputType property.

The following example extends the previous example with a typed function. This example implements an add function that takes two real values, adds them together, and returns the result.

```
       // The meta data for the "add" function. This must be a public
       // object named as "<functionName>Info" in order for the Web service
       // framework to locate it.
public var addInfo =
{
       // Here we specify the data type of each of the parameters to the
       // function. The "inputTypes" object contains one property for
       // each of the parameters of the function.
   inputTypes:
   {
          // The 'a' and 'b' values
       a: "real",
       b: "real"
   },

       // Here we specify the output type of the function. In this case
       // it is a real.
   outputType: "real",
```

```
        // Documentation to be inserted as a "<documentation>" tag in the
        // operation description in the WSDL.
        documentation: "Add two real numbers and return result."
};

        // Here at last we get to write our Web service function.
        // It is a plain M-Script function which in this case adds a and b.
public function add(a, b)
{
    return a + b;
}
```

The add function is actually exactly the same as the concat function in the previous example. The only difference here is that the framework decodes the operation information object and uses the correct XML schema types in both the WSDL and the SOAP interface.

Specifying a Type

A type is specified using an object that has at least one property that is named type that states the required type, for example:

```
inputTypes:
{
    a:
    {
        type: "real"  // Specify 'a' as type real.
    }
}
```

For the simple types bool, int, date, and so on, you can use a shorthand notation where the type object is substituted with a string that states the type:

```
inputTypes:
{
    a: "real"  // Shorthand notation for the previous example
}
```

The possible type names are exactly the same as used by M-Script, as shown in the following table.

Type Name	Resulting XML Schema Definition
bool	xsd:boolean
int	xsd:long
real	xsd:double
amount	xsd:double
date	xsd:date

Type Name	Resulting XML Schema Definition
time	xsd:time
string	xsd:string
array	special
object	special

Arrays

The array type needs some additional information about the type of the elements in the array, and to add this you must use the object notation for the type. The type object must contain the type property with the value array, as well as a property named elements that contains the type specification of all of the elements in the array, for example:

```
inputTypes:
{
    a:
    {
        // Specify an array of integers
        type: "array",
        elements: "int"
    }
}
```

It is of course possible to specify nested types:

```
inputTypes:
{
    a:
    {
        // Specify a two-dimensional array of integers
        type: "array",
        elements:
        {
            type: "array",
            elements: "int"
        }
    }
}
```

Objects

The object type needs some additional information about the type of the elements in the object, and to add this you must use the object notation for the type. The type object must contain the type property with the value object, as well as a property named elements that contains the type specification for each of the properties in the object.

```
inputTypes:
{
```

```
    a:
    {
        // Specify an object with two properties 'x' and 'y' of
        // type integer and date. type: "object",
        elements:
        {
            x: "int",
            y: "date"
        }
    }
},
```

It is of course also possible to mix simple types, arrays, and objects:

```
inputTypes:
{
    // 'a' is an object with two properties 'x' and 'y',
    //   'x' is again an object with properties 'q' and 'w'
    //   - both of these are integers,
    //  'y' is an array of integers.
    a:
    {
        type: "object",
        elements:
        {
            x:
            {
                type: "object",
                elements:
                {
                    q: "int",
                    w: "int"
                }
            },
            y:
            {
                type: "array",
                elements: "int"
            }
        }
    }
},
```

Empty Return Types

You cannot use procedures as web services operations, nor can you use null as a return value. Because of this, every operation must return some non-null value. It is suggested that you use a Boolean true value as the return value for functions that have no return values.

Using M-Script Sessions

The web services framework allows the use of M-Script sessions in the services. However, to do so, you must only define session variables inside functions— the framework cannot handle global session variables. The session ID is automatically transported in the SOAP header, but only if the service operation requires it. Thus in the operation information object you must add a property named requiresSession and set it to true. This forces the framework to include the session ID in both WSDL and SOAP headers.

The following is an example of a service that sets and gets a variable that is stored in the session.

```
#version 13
package session_use(1); // Cannot name it "session" - that's a keyword
public var serviceDescription =
{
   serviceNamespace: "maconomy.com",
   documentation: "Testing session handling."
};


/*------------------------------------------------------------ Operation: start
   Input:     none
   Output:    session identifier to be placed in URL.
------------------------------------------------------------*/
public var startInfo =
{
   inputTypes: {},
   outputType: "string",
   documentation: 'DOC'
Initiates a session.
DOC
};

public function start()
{
   newsession();
   return currentsession();
}

/*------------------------------------------------------------
   Operation: stop
   Input:     none
   Output:    none (bool true)
------------------------------------------------------------*/
public var stopInfo =
{
      // This property tells the framework to require a
      // session ID in header
   requiresSession: true,
```

```
   inputTypes: {},

   outputType: "bool",

   documentation: 'DOC'
Ends a session (deletes the session storage).
DOC
};

public function stop(val)
{
   deletesession();
   return true;
}

/*-------------------------------------------------------------
   Operation: set
   Input:     a string to store
   Output:    the previously stored value (empty first time)
-------------------------------------------------------------*/
public var setInfo =
{
      // This property tells the framework to require a
      // session ID in header
   requiresSession: true,

   inputTypes:
   {
      val: "string"
   },

   outputType: "string",
   documentation: 'DOC'
   Saves the passed value in a session variable and returns the previous
   value.
   DOC
   };

public function set(val)
{
      // Global session variable in local scope!
   session var store = "";
   var oldStore = store;
```

```
    store = val;

    return oldStore;
}

/*-------------------------------------------------------------
    Operation: get
    Input:     none
    Output:    the stored value.
-------------------------------------------------------------*/
public var getInfo =
{
        // This property tells the framework to require a
        // session ID in header
    requiresSession: true,

    inputTypes: {},
    outputType: "string",
    documentation: "Returns the saved session variable."
};

public function get()
{
        // Global session variable in local scope!
    session var store = "";
    return store;
}
```

HTTP Get Operations

The use of SOAP over HTTP POST is not always the best solution for all uses. It is a very useful solution for connecting heterogeneous systems together when you have skilled IT developers working with the various .NET or Java frameworks. But in some scenarios you might want to serve Maconomy data to less skilled web site administrators reusing already defined web services. To do this you can transform a normal HTTP GET request into a web service request and thereafter transform the generated output to readable HTML.

HTTP GET transformations for web services are by default (and always) listed in the auto-generated WSDL. This makes it possible for remote tools to inform the service requester of this possibility. The HTTP GET requests are handled via the script get.ms. Input parameters to the service are passed on the URL using query variables that have exactly the same names as the input parameters to the service operation. This means that HTTP GET can only be used for services with simple input types (not arrays and objects).

The service name and operation must also be named on the URL. For this you must use the query variables service and op.

For a function such as the following one, this results in a URL like:

```
http://yourhost.com/MaconomyWS.exe/get.ms?service=math&op=add&a=10&b=20:
```

```
public function add(a, b)
{
   return a + b;
}
```

The HTML output is fairly simple and is based on a generic algorithm for rendering of all M-Script types. This is surely inappropriate for any serious use, so the web service framework adds a possibility to transform the output using M-Script.

Output Transformations

The output transformation is done by a function that is stored in a property named outputWrapper on the operation information object. This function receives the M-Script value that is returned by the web service handler and is expected to print its output to the standard output.

The operation information object can also contain an optional property named outputContentType that holds the MIME type of the output. This information is used in both the WSDL generation and in the HTTP GET output as the ContentType HTTP header.

The following is an example:

```
public var fInfo =
{
   inputTypes:
   {
      a: "int",
      b: "int"
   },

   outputType: "int",

      // The HTTP GET output of this operation is plain text outputContentType:
      "text/plain",

      // The output wrapper is an anonymous procedure that
      // simply prints the result.
   outputWrapper: procedure(a)
   {
      print("The result is: ^1\n", a);
   },

   documentation: "Adding two integers and pretty print result"
};

public function f(a,b)
{
   return a + b;
}
```

Exception Handling

The web service framework also handles exceptions that are thrown by any service. It does this by transforming the exception data into a SOAP fault package with the exception information passed as XML.

Access Control

Some web services might require access control at a lower level than what the Maconomy application offers—for instance, by blocking a web service before the request is forwarded to the Maconomy server to save server processing time.

At this point of time the access control is rather limited. All that is required is a username. No passwords are checked, and no encryption or digital signatures are supported. The username must be supplied in the SOAP header as described by the OASIS Web Service Security standard[3] as shown in the following example:

```
<soap:Header>
   <Security>
      <UsernameToken>
         <Username> John Doe</Username>
      </UsernameToken>
   </Security>
</soap:Header>
```

Web service operations that require a username must set the property requiresUserAccessToken to true in the operation information object:

```
public var operationInfo =
{
   requiresUserAccessToken: true,

   inputTypes: ...,
   outputType: ...,
   documentation: ...
};
```

The username can then be used to block the web service as described in the next section. If you want to permit access based on the IP number of the client you can use getenv("REMOTE_ADDR") to fetch the address.

Passing the Username in HTTP Get Requests

The username must be placed on the URL as the query variable sh_username, for example:

```
http://.../get.ms?service=math&op=add&a=10&b=20&sh_username=John+Doe
```

Note the use of the plus sign as a space. This is the standard encoding used on URLs.

Disabling Web Services

A web service can decide for itself whether it is enabled or not; the framework does not supply any external methods for disabling a web service. It does, however, check a web service for being

---

See http://www.oasis-open.org/committees/tc_home.php?sb_abbrev=wss.

enabled or not. To do this it looks for a function that is stored on the enablingFunction property of the operation information object.

To implement the enabling function you must include the M-Script package mscript::soap::api because this package defines an enumerated type named ServiceEnabling. This enumeration contains three different values that the enabling function may return, as described in the following table.

Value	Description
ServiceEnabled	The service is enabled and ready to run.
ServiceDisabled	The service is disabled and not available.
ServiceMissing	The service is disabled, and the framework should not generate a WSDL for it, or in any other way show any knowledge of it.

The service enabling function is passed the operation name as the first parameter, and the current user name as the second (possibly null).

The following is an example of a service with an enabling function.

```
#version 13


package testEnabling(1);


uses mscript::soap::api(1) as soapAPI;

public var serviceDescription =
{
    serviceNamespace: "maconomy.com",
    documentation: "Testing enabling.",
        // The enabling function switches on the operation name in
        // order to decide whether the service is enabled or not.
        enablingFunction: function(operationName, username)
    {
        // Simple username check
        if (username != "John Doe")
            return soapAPI::ServiceDisabled;

        switch (operationName)
        {
            case "f":
                return soapAPI::ServiceEnabled;
            case "g":
                return soapAPI::ServiceDisabled;
            case "h":
                return soapAPI::ServiceMissing;
            default:
                return soapAPI::ServiceEnabled;
        }
```

```
    }
};
```

Standard Services

The web services framework supplies a few services itself. These are referred to as std::serviceName. Thus to use the std::wssession services you use a URL like the following:

```
http://yourhost.com/.../MaconomyWS.exe/wsdl.ms?service=std::wssession
```

std::wssession::login

std::wssession::login			
Prototype	function login(username, password)		
Description	Creates a session and logs in to the Maconomy server using the supplied username and password. Returns a session identifier (string) to be passed to the subsequent Maconomy operations as the SOAP header sessionid. The session ID can also be passed directly on the URL as …/soap.ms?sessionid=xyz (where xyz is the ID). However, it is not allowed to combine both methods.		
Parameters	**Name**	**Type**	**Description**
	username	string	Maconomy user name
	password	string	Maconomy user password
Returns	The session identifier as a string.		

std::wssession::logout

std::wssession::login	
SOAP header	The current session identifier
Prototype	procedure logout()
Description	Logout of the current Maconomy session.
Parameters	None

Debugging

Debugging a web service can sometimes be difficult because there is no output that is visible to a user. Often the tools only tell you “something went wrong,” but not why, where, and when. Fortunately M-Script offers a suite of logging facilities that can help in these situations.

Logging Scripts and Errors

You must add the following line to your M-Script's initialization file to log all M-Script executions:

```
log = scripts
```

With this setting you also get errors logged. The output is placed in MaconomyWS.log in the same directory as MaconomyWS.exe.

Logging Output

You can log all output from M-Script in a file using this setting in the initialization file:

```
debugDumpFilename=..some filename...
```

The output is placed in the specified file.

Logging Incoming Data

To log all of the data that is sent to M-Script using HTTP POST (which is what SOAP does) you must add this setting in the initialization file:

```
log = postdata
```

The output is placed in MaconomyWS.postdata.log in the same directory as MaconomyWS.exe.

Other Packages

A number of other packages are included with the Maconomy installation. The following sections describe the most important of these.

mail

The mail package supports sending mails with or without attachments. The built-in package POP3 supports receiving mails. For more information, see “POP3” in the M-Script Language Reference.

procedure setDebugStream(stream)

This procedure is used to enable disabling debug mode.

procedure setDebugStream(stream)			
Parameters	**Name**	**Type**	**Description**
	stream	output stream	If this parameter is different from null, debugging is enabled and is written to the specified stream.
History	All package versions		

Example	`uses mscript::mail(3) as Mail;`
	`var stream = io::stdout();`
	`Mail::setDebugStream(stream);`		

procedure setup(serverName, serverPort)

This procedure is used to specify the server name and port number used by the mail system.

procedure setup(serverName, serverPort)			
Parameters	**Name**	**Type**	**Description**

procedure setup(serverName, serverPort)			
	serverName	string	Mail server name
	serverPort	int	Mail server port number
History	All package versions		

Example	`uses mscript::mail(3) as Mail;`
	`var serverName = "mail.maconomy.com";`
	`var serverPort = 25;`
	`Mail::setup(serverName, serverPort);`		

procedure send(to,from,subject,message,attachments,charset)

This procedure is used to send an e-mail with optional attachments to a list of recipients.

procedure send(to,from,subject,message,attachments,charset)			
Parameters	**Name**	**Type**	**Description**
	to	string or object	Required. You can specify two types of parameters for this function (string or object): ▪ **string** — This is a semicolon-separated list of email addresses and names. Example: `"John Mylesjm@maconomy.com:; Wayne Dwight<wd@maconomy.com"` ▪ **object** — An object that contains one or more of (to: string, cc: string, bcc: string). Each string entry in the object is a semicolon-separated list of email addresses and names as shown.
	from	string	Required. Email and name of sender (for example `"My Selfmy@domain.com"`).
	subject	string	Subject string, for example `"New message for you. "`
	message	string	Message body. The newline character is "\n." Lines cannot have the dot (.) character as the only content. Example: `"Hi, \nHere is my message for you. \nBest regards, \nMy."`.

procedure send(to,from,subject,message,attachments,charset)			
	attachments	object	An array of objects of attached files. Note that when sending attachments, performance is not optimal. A null value can be used if no attachments exist. The object structure is: name: string content_type: string (optional) Data:input-stream raw:bool.raw is optional and false by default. Raw is used to specify whether the attachment should be sent raw as is. Otherwise, it may be encoded as an entity. Example: `[{name: "Att1.txt", content_type: "text/plain", data:input-stream, raw:false }]`
	charset	string	An optional string that specifies the character set used when encoding header fields and message body. If charset is not specified or is null, no words in header fields will be encoded. The default value is null. Example: `"ISO-8859-1."`
Throws	mscript::mail. Exception value is an object with the structure: `message:` *string* mscript::mail::connect. Exception value is an object with the structure: `message:`*string* `serverName:` *string* `serverPort:` *int*		
History	Basic functionality: All package versions, with the following features added later: ▪ 2.2.0: raw property added for attachments ▪ 2.4.0: charset property added		

procedure send(to,from,subject,message,attachments,charset)	
Example	uses mscript::mail(3) as Mail; var attFile = file::open(…); var attachments = [{name:"Att1.txt", content_type: "text/plain", data:attFile, raw:false{]; Mail::send({to:"Someonesomeone@domain.com;someone.else@domain.com" , cc:cc.to@domain.com}, "My Selfmy@domain.com", "New message for you", "Hi, \nHere is my message for you. \nBest regards, \nMy.",attachments, "ISO-8859-1");

procedure sendRaw(to, from, message)

This procedure is used to send an email to a list of recipients. The message body is sent raw, and can (apart from the message itself) contain headers, subject, and attachments. The sendRaw procedure simply passes message to the mail server.

procedure sendRaw(to, from, message)			
Parameters	**Name**	**Type**	**Description**
	to	string or obj	Required. You can specify two types of parameters for this function (string or object): ▪ **string** — This is a semicolon-separated list of email addresses and names. Example: `"JohnMylesjm@maconomy.com; Wayne Dwight<wd@maconomy.com>"` ▪ **object** — An object that contains one or more of (to: string, cc: string, bcc: string). Each string entry in the object is a semicolon-separated list of email addresses and names as shown.
	from	string	Required. Email and name of sender (for example, `"MySelfmy@domain.com"`).
	message	string	Message body. The newline character is "\n." Lines cannot have the dot (.) character as the only content. Example: `"Hi, \nHere is my message for you. \nBest regards, \nMy."`

procedure sendRaw(to, from, message)	

Throws	mscript::mail. Exception value is an object with the structure:
	`message: string`
	mscript::mail::connect. Exception value is an object with the structure:
	`message: string`
	`serverName: string`
	`serverPort: int`

Example	`uses mscript::mail(3) as Mail;`
	`var rawMessage = …;`
	`Mail::sendRaw("Someonesomeone@domain.com", "My Selfmy@domain.com", rawMessage);`

getTrace

Using functions in this package, you can generate an XML document that has information about which .I files, packages, and so on, are used in the Portal, and which scripts, and so on, are currently loaded. The document is transformed using XSL-T and displayed in the Portal. This information is for internal use by Customer Support Services.

ATLLibrary

The functions in the packages in the Test folder are used in connection with automated testing. These packages are for internal use.

customization

The customization package offers generic functions for finding any customized .I files or packages in a Maconomy application file structure. This is especially useful when customizing M-scripts on the server.

This package uses the name of the current application folder set by server-side M-Script. It then traverses the folders /CustomizationDir/Custom, /CustomizationDir/Solution, and /MaconomyDir below the application home and looks for packages with a given name.

Note that when working with life cycles, you can use the LifeCycles package to perform the functions in the customization package with an implicit specification of the LifeCycles file type. The LifeCycles package is located in the folder customization below the maconomy package folder.

Predefined Types

A number of file types are predefined to make it easier to search for certain customized packages. In the functions that are described in the following table, you can specify these predefined types in the type parameter to extend the search to also include the folders listed in this section.

Type	Searches Folders	Added
LifeCycles	/MScripts/LifeCycles	v.1.0.0
Reports	/Reports	v.1.0.0

Workflows	/MScripts/Workflows	v.1.0.0
GeneralMonitors	/MScripts/Workflows/GeneralMonitors	v.1.1.0

function findFilePath(type, filename)

This function tries to find a file with the name `filename` in the folders mentioned in the preceding general description of this package.

function findFilePath(type, filename)			
Parameters	**Name**	**Type**	**Description**
	type	string	The type of file (package) to search for
	filename	string	The name of the package for which to search for a customized version
Returns	*String* — fully qualified path to the found file		
Throws	mscript::maconomy::customization::findFilePath({message: string}) if no file with the given name could be found in any of the defined folders.		
History	All package versions.		

Example	`uses mscript::customization(1) as Customization;`
	`var custFile = "customization.1.ms";`
	`result = Customization::findFilePath("", custFile);`		

function getConfigurationFromFile(filename)

This function reads and returns a customization object (the TAC file information) from the specified file filename. The object is obtained by opening and reading the TAC file header from the file.

function getConfigurationFromFile(filename)			
Parameters	**Name**	**Type**	**Description**
	filename	string	Fully qualified name of the file from which to get the configuration object
Returns	*String* — the customization information read from the file.		
Throws	error::stdlib::file::open if the file cannot be opened for reading. error::stdlib::file::readvalue if a valid value cannot be read from the file.		
History	All package versions.		

Example	`uses mscript::customization(1) as Customization;`
	`var custFile = "customization.1.ms";`
	`custInfo = Customization::getConfigurationfromFile(custFile);`		

function getConfiguration(type, filename)

This function reads and returns a customization object (the TAC file information) from the specified file filename of the type type. The functionality is similar to the preceding “function getConfigurationFromFile(filename).”

function getConfiguration(type, filename)			
Parameters	**Name**	**Type**	**Description**
	type	string	The type of file (package) to search for.
	filename	string	Fully qualified name of the file from which to get the configuration object
Returns	*String* — the customization information read from the file.		
Throws	mscript::maconomy::customization::getConfiguration({message: string}) if no package with the specified path could be found in any of the defined folders. error::stdlib::file::open if the file cannot be opened for reading. error::stdlib::file::readvalue if a valid value cannot be read from the file.		
History	All package versions.		

Example	`uses mscript::customization(1) as Customization;`
	`var report = "customercard.mrl";`
	`custInfo = Customization::getConfiguration("Reports", report);`		

function getPackage(type, path)

This function dynamically loads and returns a handle to a package that has the specified type and path. Note that relative paths are not supported.

function getPackage(type, path)			
Parameters	**Name**	**Type**	**Description**
	type	string	The type of file (package) to search for
	path	string	Path to the package (for example, foo::bar(1)) relative to the folders defined by type.
Returns	A handle to the loaded package		
Throws	`mscript::maconomy::customization::getPackage({message: string})` if no package with the specified path could be found in any of the defined folders Any exception thrown by loadpackage (See the M-Script Language Reference)		

function getPackage(type, path)	
History	All package versions

Example	`uses mscript::customization(1) as Customization;`
	`var report = "customercard.mrl";`
	`packageHandle = Customization::getPackage("Reports", report);`

function findPackage(type, path)

This function returns the full path of a package that has the specified type and path. Note that relative paths are not supported.

function findPackage(type, path)			
Parameters	**Name**	**Type**	**Description**
	type	string	The type of file (package) to search for
	path	string	Path to the package (for example, foo::bar(1)) relative to the folders defined by type.
Returns	A handle to the loaded package		
Throws	`mscript::maconomy::customization::getPackage({message: string})` if no package with the specified path could be found in any of the defined folders Any exception thrown by loadpackage (See the M-Script Language Reference)		
History	Version 1.0.0		

Example	`uses mscript::customization(1) as Customization;`
	`var report = "customercard.mrl";`
	`pathHandle = Customization::findPackage("Reports", report);`		

semaphore

The semaphore package is useful when you need to prevent locking conflicts in an M-Script. Using the package, you can define semaphores (also known as mutexes). A mutex is short for “mutual exclusion object.” A mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. When a program is started, a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished.

This package contains functions for creating a mutex object and provides methods for working with the object.

function CreateNamedMutex(mutex_name)

This function returns a mutex object working on a mutex with name `mutex_name`. The mutex is created if it does not already exist.

function CreateNamedMutex(mutex_name)			
Parameters	**Name**	**Type**	**Description**
	mutex_name	string	A string that identifies this mutex
Returns	A mutex object		
History	All package versions		

Example	`uses mscript::semaphore(1) as Mutex;`
	`var mutex_name = "MyMutex";`
	`mtxHandle = CreateNamedMutex(mutex_name));`		

function GetNamedMutex(mutex_name)

This function returns a mutex object working on a mutex with name mutex_name. If the mutex does not already exist, the function returns an error.

function CreateNamedMutex(mutex_name)			
Parameters	**Name**	**Type**	**Description**
	mutex_name	string	A string that identifies this mutex
Returns	A mutex object		
History	All package versions		

Example	`uses mscript::semaphore(1) as Mutex;`
	`var mutex_name = "MyMutex";`
	`mtxHandle = GetNamedMutex(mutex_name));`		

function CreateMutex()

This function returns a mutex object working on a new mutex with a unique name.

function CreateNamedMutex(mutex_name)	
Returns	A mutex object
History	All package versions

Example	`uses mscript::semaphore(1) as Mutex;`
	`var mutex_name = "MyMutex";`
	`mtxHandle = CreateMutex(mutex_name));`

A mutex object supports the following methods.

procedure lock()

This procedure locks the mutex and ensures exclusive access to the resource protected by the mutex. A number of attempts are made to acquire the lock, controlled by the property repeat of type int in the mutex object.

procedure lock()	
Throws	Semaphore::mutex::lock({message:string}) if the mutex cannot be locked
History	All package versions

Example	`uses mscript::semaphore(1) as Semaphore;`
	`var mutex = Semaphore::CreateNamedMutex("myMutex");`
	`try`
	`{`
	`  mutex.lock(); // lock the mutex.`
	`}`
	`catch Semaphore::mutex::lock(e)`
	`  ; // lock() failed`

procedure unlock()

This procedure unlocks the mutex and allows other processes to access the resource protected by the mutex. If the lock is not currently held, the function fails silently.

procedure unlock()	
Throws	Semaphore::mutex::unlock({message:string}) if the mutex cannot be unlocked
History	All package versions

Example	`uses mscript::semaphore(1) as Semaphore;`
	`var mutex = Semaphonre::CreateNamedMutex("myMutex");`
	`try`
	`{`
	`  mutex.lock(); // We are now in the critical region.`
	`  mutex.unlock(); // unlock the mutex.`
	`}`
	`catch Semaphore::mutex::unlock(e)`
	`  ; // unlock() failed`

procedure destroy()

This procedure unlocks the mutex and removes the underlying mutex object. Because mutex objects are implemented using lock files, it is especially important to delete mutexes with auto-generated names (see “function CreateMutex()”) when they are no longer needed, to avoid building up old lock files in the file system.

procedure destroy()	
Throws	Semaphore::mutex::destroy({message:string}) if the mutex cannot be unlocked or destroyed
History	All package versions

procedure destroy()	

Example	uses mscript::semaphore(1) as Semaphore;
	var mutex = Semaphonre::CreateNamedMutex("myMutex");
	try
	{
	  mutex.lock(); // We are now in the critical region.
	  mutex.destroy(); // unlock and destroy the mutex.
	}
	catch Semaphore::mutex::destroy(e)
	  ; // destroy() failed

Deltek is the leading global provider of enterprise software and information solutions for professional services firms, government contractors, and government agencies. For decades, we have delivered actionable insight that empowers our customers to unlock their business potential. Over 14,000 organizations and 1.8 million users in approximately 80 countries around the world rely on Deltek to research and identify opportunities, win new business, optimize resource, streamline operations, and deliver more profitable projects. Deltek – Know more. Do more.®

deltek.com