

Contents

- [1. Overview](#)
- [2. Quick start](#)
- [3. What's new](#)
- Interface
 - [4. Main window](#)
 - [5. Toolbars in QM window](#)
 - [6. Menu: File](#)
 - [7. Menu: Edit](#)
 - [8. Menu: Run, Tools, Output](#)
 - [9. Keyboard shortcuts \(QM navigation\)](#)
 - [10. Properties](#)
 - [11. Folder properties, file properties](#)
 - [12. Options: General](#)
 - [13. Options: Triggers](#)
 - [14. Options: Security](#)
 - [15. Options: Files](#)
 - [16. Icons](#)
 - [17. QM macro list files](#)
 - [18. Command line parameters](#)
- QM items
 - [19. QM items](#)
 - [20. Macro](#)
 - [21. Function](#)
 - [22. Pop-up menu](#)
 - [23. Toolbar](#)
 - Menu and toolbar settings
 - [24. Menu and toolbar icons](#)
 - [25. Toolbar right-click menu](#)
 - [26. Toolbar properties](#)
 - [27. Extended toolbars](#)
 - [28. Menu properties](#)
 - [29. Autotext list](#)
- Triggers
 - [30. Triggers: keyboard \(hot key\)](#)
 - [31. Triggers: mouse](#)
 - [32. Triggers: window](#)
 - [33. Triggers: QM events](#)
 - [34. Triggers: file](#)
 - [35. Triggers: event log](#)
 - [36. Triggers: process](#)
 - [37. Triggers: accessible object](#)
 - [38. Triggers: Shell menu](#)
 - [39. Triggers: user-defined](#)
 - [18. Command line parameters](#)
 - Filter functions
 - [40. Filter functions](#)
 - [41. Filter function examples](#)
- Creating macros and programs
 - [42. Tutorial](#)
 - [43. Recording](#)
 - [44. Syntax](#)
 - [45. Programming with QM](#)
 - [46. Help and type info](#)
 - [47. Debugging](#)
 - [48. Errors](#)
 - [49. Threads](#)
 - [50. Extending QM](#)
 - [51. Make exe](#)
- [52. Reference](#)
- Mouse, keys/text, dialogs
 - [53. lef, rig, mid, dou](#)
 - [54. mou](#)

- [55. xm, ym](#)
- [56. key](#)
- [57. out](#)
- [58. paste](#)
- [59. ifk](#)
- [60. inp](#)
- [61. inpp](#)
- [62. mes](#)
- [63. Dialog editor](#)
- File
 - [64. run](#)
 - [65. cop, ren](#)
 - [66. del](#)
 - [67. Flags for cop, ren and del](#)
 - [68. mkdir](#)
 - [69. dir](#)
 - [70. iff](#)
 - [71. zip](#)
- Window, control
 - [72. act](#)
 - [73. clo](#)
 - [74. min, max, res](#)
 - [75. mov, siz](#)
 - [76. hid](#)
 - [77. win, wintest](#)
 - [78. ifa](#)
 - [79. Some window API](#)
 - [80. men](#)
 - [81. but](#)
 - [82. id](#)
 - [83. child, childtest](#)
 - [84. Accessible objects](#)
 - [85. acc, acctest](#)
 - [86. htm](#)
 - [87. scan](#)
- Time
 - [88. wait](#)
 - [89. wait for](#)
 - [90. spe](#)
 - [91. perf](#)
 - [92. tim](#)
 - [93. OLE types: DATE](#)
- Internet
 - [94. web](#)
 - [64. run](#)
 - [86. htm](#)
- Misc
 - [95. Comments](#)
 - [96. bee](#)
 - [97. opt](#)
 - [98. getopt](#)
 - [99. deb](#)
 - [100. mac](#)
 - [101. net](#)
 - [102. dis](#)
 - [103. atend](#)
 - [104. shutdown](#)
 - [105. pixel](#)
 - [87. scan](#)
 - [106. rset, rget](#)
 - [107. qmitem](#)
 - [108. newitem](#)
 - [109. sizeof](#)
 - [110. uuidof](#)
 - [111. share](#)
 - [112. lock](#)

- [113. Extensions, dll, categories](#)
- [114. Dll functions exported by QM](#)
- [115. IStringMap interface](#)
- [116. ICsv interface](#)
- [117. IXml interface](#)
- [118. QM ComboBox and drop-down list controls](#)
- [119. Grid control \(QM_Grid\)](#)
- [120. Math functions from C run-time library](#)
- [121. Services reference](#)
- Flow control
 - [122. goto](#)
 - [123. if, else](#)
 - [78. ifa](#)
 - [59. ifk](#)
 - [70. iff](#)
 - [124. iif](#)
 - [125. sel, case](#)
 - [126. rep](#)
 - [127. for, break, continue](#)
 - [128. foreach](#)
 - [129. err](#)
 - [130. ret](#)
 - [131. end](#)
 - [132. call](#)
- Operators, expressions
 - [133. Operators](#)
 - [134. Unary operators](#)
 - [135. Operator priority](#)
 - [136. Expression type and precision](#)
- Variables, constants, numbers, strings
 - [137. Numbers and strings](#)
 - [138. Strings with variables](#)
 - [139. def \(define a named constant\)](#)
 - [140. Variables](#)
 - [141. Variable declaration](#)
 - [142. Variable storage and scope](#)
 - [143. Variable storage and scope \(2\)](#)
 - [144. Predefined variables](#)
 - [145. OLE types](#)
 - [146. Safe array](#)
 - [147. Pointer, reference, array](#)
 - [148. Memory allocation functions](#)
- Functions, types
 - [149. Functions](#)
 - [150. Function tips](#)
 - [151. Calling user-defined functions](#)
 - [152. function](#)
 - [153. dll](#)
 - [154. type](#)
 - [155. User-defined type variable usage](#)
 - [156. Unions](#)
 - [157. Classes](#)
 - [158. Classes tutorial](#)
 - [159. Categories](#)
 - [160. ref](#)
- COM
 - [161. COM \(about\)](#)
 - [162. COM support in QM](#)
 - [163. Using COM components](#)
 - [164. typelib](#)
 - [165. interface](#)
 - [166. Interface pointer variables](#)
 - [167. COM object creation functions](#)
 - [168. Calling COM functions](#)
 - [169. Events](#)
 - [145. OLE types](#)

- [170. OLE types: CURRENCY, DECIMAL, VARIANT](#)
- [171. OLE types: BSTR](#)
- [93. OLE types: DATE](#)
- [146. OLE types: ARRAY](#)
- Compiler directives
 - [172. #if, #else, #endif](#)
 - [173. #ifdef, #ifndef](#)
 - [174. #compile](#)
 - [175. #err](#)
 - [176. #opt](#)
 - [177. #set](#)
 - [178. #out, #warning, #error](#)
 - [179. #exe](#)
 - [180. #region](#)
 - [181. #ret](#)
 - [182. #sub](#)
- Strings
 - [183. Strings](#)
 - [184. len](#)
 - [185. empty](#)
 - [186. val](#)
 - [187. numlines](#)
 - [188. find](#)
 - [189. findw](#)
 - [190. findt](#)
 - [191. findl](#)
 - [192. tok](#)
 - [193. tok and pointer-based array](#)
 - [194. findc, findcr, findcs, findcn](#)
 - [195. findb](#)
 - [196. matchw](#)
 - [197. findrx](#)
 - Regular expressions
 - [198. Regular expressions](#)
 - [199. Regular expression syntax](#)
 - [200. Regular expression options](#)
 - [201. Callout callback function](#)
 - [202. FINDRX](#)
 - [203. REPLACERX, replacement callback function](#)
- str
 - [183. Strings](#)
 - [204. addline](#)
 - [205. all](#)
 - [206. beg, begi, end, endi, mid, midi](#)
 - [207. dllerror](#)
 - [208. dospath](#)
 - [209. encrypt, decrypt](#)
 - [210. escape](#)
 - [211. findreplace](#)
 - [212. fix](#)
 - [213. format](#)
 - [214. from](#)
 - [215. fromn](#)
 - [216. getclip, setclip, getsel, setsel](#)
 - [217. getfile, setfile](#)
 - [218. getl](#)
 - [219. getmacro](#)
 - [220. setmacro](#)
 - [221. getpath, getfilename](#)
 - [222. getstruct, setstruct](#)
 - [223. gett](#)
 - [224. getwintext, setwintext, getwinclass, getwinexe](#)
 - [225. insert](#)
 - [226. lcase, ucase](#)
 - [227. left, right, get, geta](#)
 - [228. remove](#)
 - [229. replace](#)

- [230. replacerx](#)
- [231. searchpath, expandpath](#)
- [232. set](#)
- [233. swap](#)
- [234. str properties \(lpstr, len, nc, flags\)](#)
- [235. time](#)
- [236. timeformat](#)
- [237. trim, ltrim, rtrim](#)
- [238. unicode, ansi](#)
- Other info
 - [239. Character codes](#)
 - [240. Color](#)
 - [241. Coordinates](#)
 - [242. Declarations](#)
 - [243. DPI-scaled windows](#)
 - [244. Expressions](#)
 - [245. F1 help and the tips pane](#)
 - [246. File search paths](#)
 - [247. Flags](#)
 - [248. Format fields](#)
 - [249. Globally unique identifiers](#)
 - [250. High-order word, low-order word](#)
 - [251. Key codes](#)
 - [252. License](#)
 - [253. Line breaks](#)
 - [254. Links in mes, inp and other dialogs](#)
 - [255. Make macro smaller](#)
 - [256. MSDN Library \(Windows API\)](#)
 - [257. Names of QM identifiers](#)
 - [258. Network setup](#)
 - [259. Portable QM](#)
 - [260. QM file management functions, macro settings and resources](#)
 - [261. Resources](#)
 - [262. Setup command line parameters](#)
 - [263. Table of delimiters](#)
 - [264. Trigger coding](#)
 - [265. Type conversions, etc](#)
 - [266. Type declaration for functions, parameters, etc](#)
 - [267. Unicode, UTF-8](#)
 - [268. Unlock computer](#)
 - [269. VARIANT members table](#)
 - [270. Virtual-key codes](#)
 - [271. Wildcard](#)
 - [272. What's new in versions 2.1.0 - 2.3.2](#)
 - [273. Window enumeration callback function](#)
 - [274. Window expressions](#)
 - [275. Window styles](#)
 - [276. Windows keyboard shortcuts](#)
 - [277. Windows Vista, 7, Windows 64-bit](#)

Overview

Purpose and some main features

Quick Macros - automation software for Windows. Some features:

- Many macro commands, including user interface automation, launching programs, file management, text processing and custom dialogs.
- Many triggers, including hotkeys, mouse, scheduler, toolbars and menus.
- Records keyboard and mouse actions.
- Programming language with functions, classes, full API support.
- You can create programs for various purposes, and run them in Quick Macros or as exe files.

Program info

Version: 2.4.8.

Requirements: Windows XP/Vista/7/8/10.

Web site: <http://www.quickmacros.com>

[License \(252\)](#): shareware, 30-day evaluation period.

[Order info](#)

See also: [Network setup \(258\)](#), [Setup command line parameters \(262\)](#)

Quick start

[What is a macro](#)
[How Quick Macros works](#)
[Quick Macros window](#)
[Creating new macro, menu or toolbar](#)
[Adding macro commands](#)
[Recording](#)
[Adding menu items and toolbar buttons](#)
[Launching macros, menus and toolbars](#)
[How to know that a macro is running](#)
[How to end a running macro](#)
[How to assign a toolbar to a window](#)
[How to create an "auto-hide" toolbar](#)
[Managing toolbars](#)
[How to create text-replacement \("autotext"\) macros](#)
[Can macros run in any program?](#)
[Can two macros run simultaneously?](#)
[Can I automate background windows?](#)
[Can I run a macro without running QM?](#)
[Can the macro run when an error occurs?](#)
[What is the System folder and other default items in the list?](#)
[What is the list that pops up when I type dot \(.\)?](#)
[Disabling commands, adding comments](#)
[Using strings; inserting newlines and quotes](#)
[Repeating](#)
[Using variables](#)
[Top 20 commands](#)

[Tutorials \(42\)](#)
[Video tutorials](#)
[Reference \(52\)](#)
[Programming in QM \(if, goto, functions, etc\) \(45\)](#)

What is a macro


A [macro \(20\)](#) is a list of [commands \(52\)](#) that are executed when the macro runs. The commands can perform the same actions as you can do manually: type text, click menus, run files, etc. To launch a macro, you can assign it a [trigger \(10\)](#) (eg a hotkey), or place it in a toolbar, or use some other way. That is, instead of manually doing the same sequence of actions again and again, you can place it in a macro, and launch it with just a single click or keystroke.

How Quick Macros works

In the [Quick Macros window \(4\)](#) you can create macros and [items \(19\)](#) of other types (functions, toolbars, etc). You create a macro for each task you want to automate. You can assign it a [trigger \(10\)](#), eg a hotkey.

While Quick Macros is running, it watches for these triggers. On a trigger event (eg a hotkey pressed) it executes the macro to which the trigger is assigned. Quick Macros also manages your toolbars and menus.

Quick Macros window

To show the [Quick Macros window \(4\)](#), click  icon in the system notification area (by the clock) or in the Start menu. To hide - click the X (Close) button.

Window parts:

Top - menu and toolbars.

Left - list of [QM items \(19\)](#) (macros, functions, menus, toolbars, etc).

Middle - code editor. Here you edit text (list of commands) of the currently selected QM item.

Bottom - output (errors and other messages), find, tips, status bar.

Creating new macro, menu or toolbar

You create a macro for each task you want to automate. To create new macro, click the New Macro button on the toolbar,

2. Quick start

and type a name in the small field that appears at the top of the list. To create an [item \(19\)](#) of other type (function, toolbar, etc), use the popup menu that appears when you click the small arrow beside the New Macro button.

Adding macro commands

Macros are [stored as simple text \(44\)](#). Usually each command is in a separate line.

To add macro commands, you can use dialogs from the floating toolbar. However QM does not provide dialogs for all commands. You can find commands and dialogs using the [Find help, functions, tools \(4\)](#) field (above the code editor), or in [Reference \(52\)](#), or in [lists \(46\)](#).

The simplest way to add keyboard and mouse commands - record.

Recording

You can [record \(43\)](#) keys and mouse. You can record complete macro, or only some parts.

To start recording, click the Record button on the toolbar, or press Ctrl+Shift+Alt+R. When recording is finished, click Insert or ... button in QM Recording dialog.

Recorded macro often is not perfect. You have to review and possibly edit it. For example, recorded macro may run too fast, and you have to insert [delays \(88\)](#) or change the wait time in [wait](#) commands. Recorded window names often contain document name, which causes error when running the macro when there is open other document. In this case, remove document name.

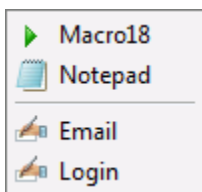
When recording, it is recommended to use the keyboard instead of the mouse (where possible), especially to select menu items (use Alt and underlined letters). It is because menus and other objects next time may be in another place, even when recorded mouse coordinates are relative to window (default).

Adding menu items and toolbar buttons

[Menu text \(22\)](#) is simply a list of macro commands. Each command is preceded by some text (label), space and colon. The same with [toolbars \(23\)](#). Each line creates menu item or toolbar button, which will execute the command.

Example:

```
Macro18 :mac "Macro18"
Notepad :run "notepad.exe"
-
Email :key "my@ema.il" * text.ico
Login :key "abcd"; key T; key "1234"; key Y * text.ico
```



To quickly add a macro, drag and drop it from the list to the menu/toolbar text. To add a file, use the Run File dialog from the floating toolbar, or drag the file from the desktop or Windows Explorer. To add other commands, use other dialogs from the floating toolbar. To add a separator, type - or |. To add [icons \(24\)](#), use the Icons dialog.

Launching macros, menus and toolbars

You can use various ways to launch a macro:

The Run button on the QM toolbar.

A [hotkey \(30\)](#), [mouse \(31\)](#), [window \(32\)](#), or [other \(10\)](#) trigger. To assign a trigger click the Properties button on the toolbar. Place the macro in a custom [menu \(22\)](#) or [toolbar \(23\)](#). Open the menu or toolbar, and drag the macro to the code editor. You can attach toolbars to windows.

Schedule the macro to run at a certain time. Click the Schedule button in the Properties dialog.

Create a shortcut on the desktop. Click the Shortcut button in the Properties dialog.

Other programs can launch QM macros using [command line \(18\)](#).


You can use the [mac \(100\)](#) command to launch it from another macro.

[Autotext list \(29\)](#) commands are executed when you type certain text.


2. Quick start

Menus, toolbars and functions can be launched using the same ways as macros.

How to know that a macro is running

When a [macro \(20\)](#) is running, the QM tray icon is red . When [QM items \(19\)](#) of other types (functions, menus, etc) are running, the tray icon does not change. It also does not change if the macro has option 'Run simultaneously'. You can see running items in the 'Running items' pane in QM window or in the Threads dialog in QM tray menu.

How to end a running macro

When a macro is running, the QM tray icon is red . To end running macro manually, press Pause key (you can change this in Options). If input is blocked ([BlockInput](#) is used in the macro), at first press Ctrl+Alt+Delete.

If it is a [function \(21\)](#), you cannot end it with Pause. Also if the macro has option 'Run simultaneously'. Then use the 'Running items' pane in QM window or the Threads dialog in QM tray menu. Or, if you use the [AddTrayIcon](#) function, you can Ctrl+click the tray icon added by it.

Or, you can use special code in the macro or function. Example:

```
rep
  ifk(F12) ret ;;end if F12 is pressed
...
```

How to assign a toolbar to a window

Assign it a [window trigger \(32\)](#). To assign a window trigger, select Window in the Properties dialog, drag the "Drag" picture and drop onto the window. May need to edit or remove window name, eg remove the name of the currently open document.

How to create an "auto-hide" toolbar

Click menu File -> New -> Templates -> Toolbar Top or Bottom, or Toolbar Left or Right. To launch it, you for example can assign a [mouse trigger \(31\)](#). Or launch it [at startup \(33\)](#).

Managing toolbars

You can drag [toolbars \(23\)](#) with the right mouse button. To drag or delete buttons, Shift+Drag. You can drag macros, files and Internet links and drop onto a toolbar. Use the right-click menu to change toolbar style or open to edit.

If you have lost a toolbar (it is running but invisible), right click it in the 'Running items' pane in QM window and select 'Move Here' or 'Reset'.

How to create text-replacement ("autotext") macros

Don't need to create separate macros for each text replacement. Create one or several [autotext lists \(29\)](#) that contain multiple replacements. Assign Autotext or Keyboard trigger.

Can macros run in any program?

Macros can run in any Windows program. You can make a macro to run only in a certain program(s).

In some programs, triggers and some macro commands may not work. The reason usually is either security-related or the program uses a nonstandard input method. Make sure that: 1. In Options -> General is selected Administrator or uiAccess, not User. 2. In Options -> Triggers is checked low level hook for keyboard and mouse.

Can two macros run simultaneously?

By default, multiple [macros \(20\)](#) cannot run simultaneously. If a macro (macro2) is launched while a macro is already running, macro2 will not run. Several instances of the same macro also will not run simultaneously.

A macro can run simultaneously with other macros if you select 'Run simultaneously' in Properties -> Macro properties. [Read more \(20\)](#).

Multiple [functions \(21\)](#) always can run simultaneously, as well as several instances of the same function.

Can I automate background windows?

Macros cannot send keys and mouse clicks to a background window without activating it. However, it is often possible (but not so easy, and will not always succeed) to replace keyboard and mouse commands with other commands that usually can work in the background. To insert such commands, use dialogs from the floating toolbar's "Windows, Controls" menu.

Can I run a macro without running QM?

Yes. You can [create executable programs \(51\)](#) from macros and functions.

Can macro run when an error occurs?

When a macro is launched, at first it is [compiled \(47\)](#). This includes error checking. If the macro contains [errors \(48\)](#), it is not executed. Error description is displayed in the QM output, and error place is highlighted.

When macro runs, some commands may fail (eg due to a missing file or window). Then macro ends. To continue on run-time error, you can use [err \(129\)](#). Example:

```
run "abc.exe" ;;macro would end if file "abc.exe" does not exist
err ;;allows macro to continue when an error occurs in the preceding command
```

What is the System folder and other default items in the list?

Many QM features are written in QM language. It includes the floating toolbar and its dialogs, the Dialog Editor, and many functions that you can use in macros. All this is in the System folder.

The Samples folder is only for learning. You can delete it.

What is the list that pops up when I type dot (.)?

The [list \(46\)](#) contains functions and other identifiers that you can use in macro. Double click an item in the list to insert it in macro. At the top of the list are categories - collections of related functions. When you double click a category in the list, pops up another list containing functions from that category.

Disabling commands, adding comments

Lines that begin with a space are disabled. This also can be used to add [comments \(95\)](#). To quickly disable/enable single line, right click the selection bar (thin gray bar at the left of the macro text). To disable/enable several lines, select them and right click the selection bar. To add comments at the end of line, use two semicolons.

Example:

```
comments
key Cv ;;press Ctrl+V
```

Using strings: inserting newlines and quotes

[Strings \(137\)](#) (text) must be enclosed in double quotes. In strings, use " (two ') for ". Use [] for new line. Example:

```
out "This is[]a multiline string[]with 'double quotes'."
```

The same without replacing quotes etc:

```
str s=
  This is
  a multiline string
  with "double quotes".
out s
```

In dialogs (in the floating toolbar), type exact text, without quotes. To use a variable in a dialog field where text is required, enclose it in parentheses (except when there is a checkbox or other control to specify that it is a variable).

See also: [variables in strings \(138\)](#)

Repeating

Assume you want to repeatedly execute two commands:

```
lef 100 200
key Y
```

Insert [rep \(126\)](#) before. Then select the commands and press Tab key. This tab-indent the selected lines:

```
rep
  lef 100 200
  key Y
```

Repeat 10 times:

```
rep 10
  lef 100 200
  key Y
```

See also: [Programming in QM \(45\)](#)

Using variables

You can use [variables \(140\)](#) almost everywhere in code. To store numeric integer values, use variables of type int. To store floating-point values - double. To store text - str. Example:

```
without variables
lef 100 200 "Notepad"

use variables
str s="Notepad" ;;declare str variable s, and store "Notepad"
int x y ;;declare int variables x and y
x=100
y=x*2
lef x y s
```

To share a variable between macros, declare it with +:

```
int+ global_var
```

You can use variables in dialogs (floating toolbar). To use a variable in a text field, enclose it in parentheses. Parentheses are not used in fields that accept numeric values.

See also: [Programming in QM \(45\)](#)

Top 20 commands

Creating and understanding macros is easier if you know the following 20 commands. Some of them can be entered using dialogs, or recorded, but often it is quicker to write them directly.

lef. rig (53)	Click mouse left or right button. To enter these commands, you can record, or use the Mouse dialog. Examples: <pre>lef 100 200 ;;left click at 100x200 pixels lef 90 40 "Notepad" ;;left click at 90x40 pixels in Notepad window</pre>
key (56)	Press keyboard keys. Type text. To type text, enclose the text into quotes. To enter other keys (251) , you can record, or use the Keys dialog. Examples: <pre>key "Australia" ;;type text '"Australia" ;;type text (key can be replaced with ') key F12 ;;press F12 key LLLL ;;press Left Arrow 4 times key Cv ;;press Ctrl+V key A{ep} ;;press Alt+E+P</pre>
paste (58)	Paste text. This command also can be entered through the Text dialog. Examples: <pre>paste "New Zealand" ;;paste text "New Zealand" ;;the same (the paste keyword can be omitted)</pre>

	<code>paste s ;;paste variable s</code>
out (57)	<p>Display something (numbers, strings, variables, etc) in QM output. Useful when debugging (47) macros or learning/testing various functions. This command also can be entered through the Text dialog. Examples:</p> <pre>out "I am here" ;;display text out i ;;display variable i</pre>
mes (62)	<p>Display something in a message box. This command also can be entered through the Message Box dialog. Examples:</p> <pre>mes "Important information" if(mes("Continue?" "" "YN")='N') ;;if the user clicks No ret ;;exit</pre>
run (64)	<p>Run file, open document, web page, etc. This command also can be entered through the Run File dialog or other dialogs from the same menu. Or, you can drag and drop a file from the desktop or Windows Explorer. Examples:</p> <pre>run "C:\WINDOWS\system32\notepad.exe" run "http://www.quickmacros.com"</pre>
act (72)	<p>Activate window. This command also can be entered through the Window dialog. Example:</p> <pre>act "Notepad"</pre>
win (77)	<p>Find window and return window handle that can be used in other commands. This command also can be entered through the Find Window Or Control dialog. Example:</p> <pre>int hwnd=win("Internet Explorer" "IEFrame") act hwnd</pre>
wait (88), wait for (89)	<p>Insert simple delay, or wait for some event (e.g., window). This command also can be entered through the Wait dialog. Examples:</p> <pre>wait 5 ;;wait 5 seconds 0.5 ;;wait 0.5 second (wait can be omitted) wait 30 "Internet Explorer" ;;wait max 30 s for Internet Explorer</pre>
mac (100)	<p>Launch a macro. This command usually is used in menus and toolbars. To enter it quickly, drag and drop the macro from the list. Example:</p> <pre>mac "Macro9"</pre> <p>Note: <code>mac</code> is not used to call functions. The launched macro is independent from the current macro, and the current macro does not wait for it.</p>
err (129)	<p>Continue macro if an error occurs. Example:</p> <pre>wait 2 "Window" ;;wait for Window max 2 s, and then throw error err ;;on error continue</pre>
ret (130)	<p>Exit (end current macro or function). Example:</p> <pre>if(i=0) ret ;;if variable i is 0, exit</pre>
goto (122)	<p>Go to another line. Example:</p> <pre>if(i>=10) goto g1 ;;if variable i is >= 10, go to the g1 label out "This line is executed only if i is < 10" g1 out "This line is executed always"</pre>
if, else (123)	<p>Execute or skip commands if certain condition is true. The commands must be preceded by a tab, unless they are in the same line. Example:</p> <pre>if i<5 ;;if variable i is less than 5, execute the following two commands _out "variable i is < 5" _i+1 else ;;else execute the following one command _out "variable i is >= 5"</pre>
rep, break (126)	<p>Repeatedly execute commands. The commands must be preceded by a tab, unless they are in the same line. Examples:</p> <pre>rep 10 ;;press Right Arrow 10 times _key R</pre> <p>Repeat while variable i is <= 5:</p> <pre>int i ;;declare variable i rep _i+1 if i>5 _out "i is > 5" break ;;exit the loop _out i</pre>

2. Quick start

int
str
(140)

Declare variables. Use **int** for numeric integer variables. Use **str** for string variables. Examples:

```
str s="Notepad" ;;declare str variable s, and store "Notepad"  
int x y ;;declare int variables x and y  
int+ g_var ;;declare global int variable g_var
```

What's new

QM 2.4.8 (2018-12-25)

1. Fixed bug: memory leak in GetFilesInFolder, FE_Dir (used with foreach to enumerate files), Dir.dir, IEnumFiles.
2. Function RunConsole2: added flags for UTF-8 and some other text encodings. Also now does not fail when the program is in the 64-bit system folder and not in the 32-bit folder.
3. Functions IntGetFile, Http.Get, Http.GetUrl: added parameter sendHeaders.

QM 2.4.7 (2017-10-12)

Main new features

1. Works better with new versions of Firefox and Chrome web browsers.

All new features

1. Functions that can use Firefox and Chrome web page element attributes and HTML now support 64-bit Firefox/Chrome too. Previously only 32-bit. Note: with Chrome it works only if is installed Firefox of same 32/64 bit as Chrome. These functions are Acc.Find, Acc.FindFF, Acc.WebX functions, FFNode class functions.
2. Fixed: With Firefox 56, Acc.Find and similar functions are slow (always > 1 s) when using option "in web page" or "as Firefox node".

QM 2.4.6 (2017-10-05)

Main new features

1. Bug fixes.

All new features

1. Fixed:
 - With new Firefox versions, Acc.Find and similar functions are slow (always > 1 s) when using option "in web page" or "as Firefox node".
 - With new Chrome versions, cannot enable Chrome web page accessible objects. [More info \(84\)](#).
 - With some new Chrome versions, in the "Find accessible object" dialog, the Window field contains text "Invalid window".

QM 2.4.5 (2017-02-27)

Main new features

1. Bug fixes.

All new features

1. Fixed:
 - str.setsel and str.setclip: if text contains Unicode characters, sets wrong clipboard text, possibly throws exception or corrupts memory.
 - [lef/rig/mid/dou](#) flag 4 ignored when window is 0.
 - And more.

QM 2.4.4 (2016-11-06)

Main new features

1. Bug fixes.

All new features

1. More options in:
 - Several System functions have new flags or parameters. Search for text "2.4.4".
2. New functions:
 - `HtmlDoc.GetTable2D`.
3. Fixed:
 - Current Chrome version: often cannot capture accessible objects.
 - Current Windows 10 version: attached toolbars are visible on all virtual desktops.
 - [hid](#)- restores maximized or minimized window.
 - Editor: no member info for some variables if the number of variables exceeds about 240.
 - And more.

QM 2.4.3.8 (2016-02-18)**Main new features**

1. Works better on Windows 8-10.
2. Shared sub-functions.
3. Several new functions; more features in [scan](#) and some other functions.
4. Bug fixes.

All new features

1. New in QM language:
 - Shared [sub-functions \(182\)](#).
 - Private/protected [class \(157\)](#) members can be accessed from any QM item named `ClassName_X`.
2. New functions and controls:
 - [IsWindowCloaked](#), [WinTest \(114\)](#), [DT_SetControl](#), [DT_GetControl](#).
 - [ICsv \(116\)](#) member functions [CellInt](#), [CellHex](#), [AddRow1](#), [AddRow3](#), [AddRowsCSV](#), [ReplaceRowsCSV](#).
 - [ShowDropDownList \(118\)](#).
 - Controls [QM_ComboBox \(118\)](#) and [QM_Edit \(118\)](#).
3. More options in:
 - [scan \(87\)](#): multiple images, search in accessible object, can be faster. Also in related functions - [wait C \(89\)](#) and [pixel \(105\)](#).
 - Other functions: [RealGetNextWindow](#), [GetMainWindows](#), [GetWindowIcon](#), [CloseWindowsOf](#), [ShowMenu](#), [ICsv.Find](#).
4. Some enhancements in dialogs: Find, Dialog Editor, Toolbar properties.
5. Fixed:
 - Problems on Windows 10 or/and 8 with: [DPI-scaled windows \(243\)](#), Win10 app windows, topmost windows, shell menu icons, unlock computer, mouse triggers after sleep, and more.
 - Incorrect behavior on RT error in a callback function. For example, on error in a dialog procedure, [ShowDialog](#) returns 1 (must end thread immediately).
 - Incorrect wait-for-handle behavior if the handle is 0. For example, `wait 5 H 0` behaves like `wait 5`.
 - If two member functions have sub-functions with same name, and they call them with a variable (`var.sub.Func`), may be called wrong sub-function (of other member function).
 - QM in some cases stops working on run-time error/[end](#) in dialog/window procedures.
 - QM stops working on stack overflow exceptions.
 - In 2.3.4.5 fixed: on some XP and Vista computers cannot use some floating toolbar dialogs and function `Htm.FromXY`.
 - In 2.3.4.6 fixed: QM stops working when trying to open macro/function properties.
 - In 2.3.4.7 fixed: [lef/rig/mid/dou](#) flag 4 ignored. And several other bugs.
 - In 2.3.4.8 fixed: invalid signature of `quickmac.exe` (installer) of version 2.3.4.7.
 - And more.
6. **What can be incompatible with previous versions:**
 - Does not support Windows 2000.
 - Replacing the default QM floating toolbar by setting registry value "GinDi\QM2\User\Tools:Toolbar" now does not

3. What's new

work.

- As always, will need to rename your functions that have names of the new QM functions.

QM 2.4.2 (2014-11-22)

Exact version: 2.4.2.2.

Main new features

1. Improvements in Dialog Editor, scheduler support and some other features.
2. Supports Java accessible objects.
3. Bug fixes.

All new features

1. [Dialog Editor \(63\)](#): better supports sub-functions; copy/paste controls; capture alien controls; Tab-select controls; styles for more classes; SysLink controls; and more.
2. Creates and manages [scheduled tasks \(18\)](#) configured for Windows Vista and later.
3. Macros can be set to [run simultaneously \(20\)](#), like functions.
4. Menu editor. Creates menu definition for dialog menu bar or popup menu.
5. [Acc/acc](#) supports [accessible objects in Java applications \(84\)](#), including OpenOffice and LibreOffice.
6. New functions:
 - [DT_MouseWheelRedirect](#), [DT_SetMenuIcons](#).
 - [IXmlNode \(117\)](#): [DeleteChild](#), [DeleteAttribute](#).
 - [MenuPopup.Create](#). Creates from menu definition. [ShowMenu](#) also supports menu definition.
 - [Acc.JavaAction](#).
7. New in:
 - [RtOptions \(114\)](#): can change some initial run-time options as with [opt](#).
 - [ExcelSheet.Cell](#) can get comment and hyperlink.
 - QM grid control: added style to drag and drop rows.
 - [win \(77\)](#): new properties: [threadId](#), [processHandle](#).
 - [str.ConvertEncoding](#): supports charset names and can detect encoding from text.
 - Now functions can have more than 255 local variables.
8. Bug fixes:
 - Unicode mode is not set by default on user accounts where QM was not installed.
 - The 'Find help, functions, tools' tool does not index many words.
 - Sometimes error after changing type/class definition.
 - [Acc.GetChildObjects](#): clears the variable.
 - Chrome auto-enabling accessible objects does not work first time.
 - [run](#) returns an invalid value if [hwnd](#) used.
 - Fixed in 2.4.2.3. QM-created programs create My QM folder on computers where they run.
 - And more.

QM 2.4.1 (2014-06-20)

Exact version: 2.4.1.8.

Main new features

1. Better supports macro resources.
2. Records and displays images.
3. Sub-functions.
4. Several new functions. Improvements in some functions and other features.
5. Bug fixes.

All new features

1. Many QM file/image functions support [macro resources \(261\)](#).
2. Added dialog 'Resources' to manage macro resources.
3. Displays images in the code editor. [More info \(12\)](#).
4. Can display images in [output and tips \(245\)](#) controls.
5. Records screenshots (small images) and displays in the code editor. [More info \(12\)](#).
6. New functions:
 - [#sub \(182\)](#) - sub-functions.
 - [#region, #endregion \(180\)](#) - hide/collapse code blocks in the code editor.
 - [QM file management functions \(260\)](#).
 - [ExeOutputWindow, RedirectQmOutput \(114\)](#).
 - [ExeConsoleWrite, ExeConsoleRead, ExeConsoleRedirectQmOutput](#).
 - [ListDialog](#) (replaces [list](#)).
 - [FileGetAttributes, FileSetAttributes](#).
 - [Dir](#) class: [TimeModifiedUTC, TimeModifiedLocal, TimeCreatedUTC, TimeCreatedLocal, TimeAccessedUTC, TimeAccessedLocal](#).
 - [Acc.NotFound, Htm.NotFound](#).
 - [DT_SetAutoSizeControls](#).
 - [GetQmItemsInFolder \(114\), InitWindowsDll \(114\)](#).
 - [str.swap \(233\)](#).
7. New in:
 - You can explicitly set address of functions declared with [dll \(153\)](#)-, like `&Func=address`, and it can be any function, not just dll.
 - [dll \(153\)](#) function pointer parameters: Can declare with & (reference). Can pass variables without operator &.
 - Directives: [#if/#ifdef/#ifndef](#) can be nested in other [#if/#ifdef/#ifndef](#) or [#else](#) code block. Also now most directives can be tab-indented.
 - [mes, inp](#) and other standard dialogs: [links \(254\)](#).
 - [opt \(97\)](#): save/restore current [opt](#) and [spe](#) settings.
 - [getopt \(98\)](#): itemname.
 - [sel \(125\)](#): flags 4 (regular expressions) and 8 (null string match "").
 - [run \(64\)](#): flag 0x40000 - support more verbs, including "Properties".
 - C# and VB.NET functions: supports .NET 4.x.
 - [RtOptions \(114\)](#): [net_clr_version](#) (for C# and VB.NET functions).
 - [QmHelp](#) and [help tags \(245\)](#): can use QM help topic name instead of path.
 - [ArrangeWindows](#): action 8 - Flip 3D.
 - [SelectTab](#): can use tab name instead of index.
 - Trigger [QM show \(33\)](#): added option 'Run synchronously'. Can be used to show a password dialog.
8. New in QM window:
 - Menu [File \(6\)](#) -> Open/Close Item -> Open in primary/secondary/both editors.
 - Menu File -> Item+ -> Copy name/path.
 - Menu File -> File+ -> Find item in files.
 - Menu [Edit \(7\)](#) -> Lines -> Hide selected. Inserts [#region \(180\)](#) directive.
 - Menu Edit -> View -> Indentation. Shows indentation guides.
 - Menu Edit -> View -> Multiple selections. You can make multiple selections with Ctrl.
 - Menu [Run \(8\)](#) -> Compile Options -> Show unused variables.
 - Find: Replacing macro names and triggers.
 - Find: Better highlighting.
9. New in Dialog Editor:
 - Visible grid.
 - Right-drag to resize dialog.
 - Alt+drag to move/resize precisely.
10. Can create [console exe \(51\)](#).
11. Bug fixes:
 - QM 2.4.0 bug: does not get Windows Forms (.NET) control name. Affected functions: [child](#), [acc](#), [Acc.Find](#), [RecGetWindowName](#).

3. What's new

- QM 2.4.0 bug: records incorrect [mou](#) data when selected Variable.
- QM 2.4.0 bug: [newitem](#) always creates new folder if folder specified without path.
- QM 2.4.0 bug: scan "file.bmp" shows some text in QM output.
- Sometimes on Windows shutdown/restart/logoff does not save some settings and does not call global variable destructors.
- Exe exits on cancelled Windows shutdown/restart/logoff.
- Some `ERRC_` constants may be incorrect after reopening QM file.
- The 'end-macro' hotkey does not work while executing [key](#).
- [run](#) fails if used ITEMIDLIST string with more string arguments.
- [ArrangeWindows](#) 0 not always works.
- POP3 email functions: error when with flag 0x100 (progress dialog).
- [zip](#):- files extracted from some zip files are read-only.
- And more.

12. What can be incompatible with previous versions:

- If you have functions with same names as the new functions, will need to rename or delete them.

Changes since QM 2.4.1.0, shortly:

- 2.4.1.1. FileGetAttributes, FileSetAttributes, InitWindowsDll, `_qmfile.GetLoadedFiles`, `str.swap`, `#region/#endregion`, `dll` and `&`, directives, links in mes etc, opt save/restore, `getopt itemname`, run flag 0x40000, .NET 4.x, `RtOptions` `net_clr_version`, [ArrangeWindows](#) action 8, new QM menu items and Dialog Editor features, several bugs.
- 2.4.1.2. Fixed QM 2.4.1.1 bug in Dialog Editor.
- 2.4.1.3. sel regex/null flags. Fixed QM 2.4.1.1 bug in 'Find help, functions, tools', `str.stem` and `str.wrap`.
- 2.4.1.4. `#sub`. Fixed QM 2.4.1.1 bug: F1 does not work for some functions. Fixed more bugs.
- 2.4.1.5. Console exe, console functions. Fixed bugs: .NET control name, `zip-` readonly. And more.
- 2.4.1.6. Menu File -> File+ -> Find item in files. Trigger 'QM show' option 'synchronous'. Bug fixes.
- 2.4.1.7. Fixed: does not record images in 'Screen' mode.
- 2.4.1.8. Updated QM icon. Fixed some small bugs.

QM 2.4.0 (2014-02-05)

Exact version: 2.4.0.3.

Main new features

1. New QM file format.
2. Some changes in user interface.
3. New options in some functions.
4. Bug fixes.

All new features

1. [New QM file format, and related changes \(17\)](#).
2. Changes in user interface:
 - Moving multiple QM items: Right-drag to check. Drag all checked.
 - In the list of QM items added special folders "Deleted" (deleted items) and "Temp" (temporary items). Can drag to delete/undelete, or to make temporary or not.
 - The "Open items" list now is auto-ordered. Removed toolbar buttons "Previous" and "Next", use the "Open items" list instead.
 - Bookmarks replaced with [Tags \(5\)](#).
 - The Find dialog: auto-highlight and more.
 - Toolbar positions now are saved in current QM file, not in registry. See also: [Options -> Layout of toolbars \(12\)](#).
 - And more small changes. Search for "2.4.0" in this Help file.
3. New functions:
 - [FileExists \(114\)](#) (replaces `dir`).
 - `Dir` class: several new member functions.
 - `str` class: [SqlBlob](#).

3. What's new

4. New in:

- [str.setfile \(217\)](#), [IXml.ToFile \(117\)](#), [ICsv.ToFile \(116\)](#): safe/atomic saving and backup.
- [scan \(87\)](#): use image from macro resources.
- [wait \(89\)](#): Added HMA to wait for all handles. Uses alertable waiting.
- New options and changes in [newitem \(108\)](#), [Sqlite.Open](#), [FE_Dir](#), [GetFilesInFolder](#), QM grid control styles.

5. Removed features:

- Bookmarks+searches pane. Bookmarks replaced with Tags. Saved searches now are only in the Find dialog.
- [Command line \(18\)](#): M with L.
- Toolbar settings: /ini.
- Options -> Security -> Lock file.
- Options -> Files -> Always add these shared files.
- Menu File -> Item -> Sort.

6. Bug fixes:

- Menu File -> File... -> File Properties does not work.
- Some triggers don't work if there are 10 or more user sessions. Window, autotext, accessible object and shell menu triggers.
- Sometimes QM crashes in debug mode.
- Auto-enabling accessible objects in new Chrome versions.
- Make exe: in some cases incorrectly writes resources, then exe may crash.
- And more.


7. What can be incompatible with previous versions:

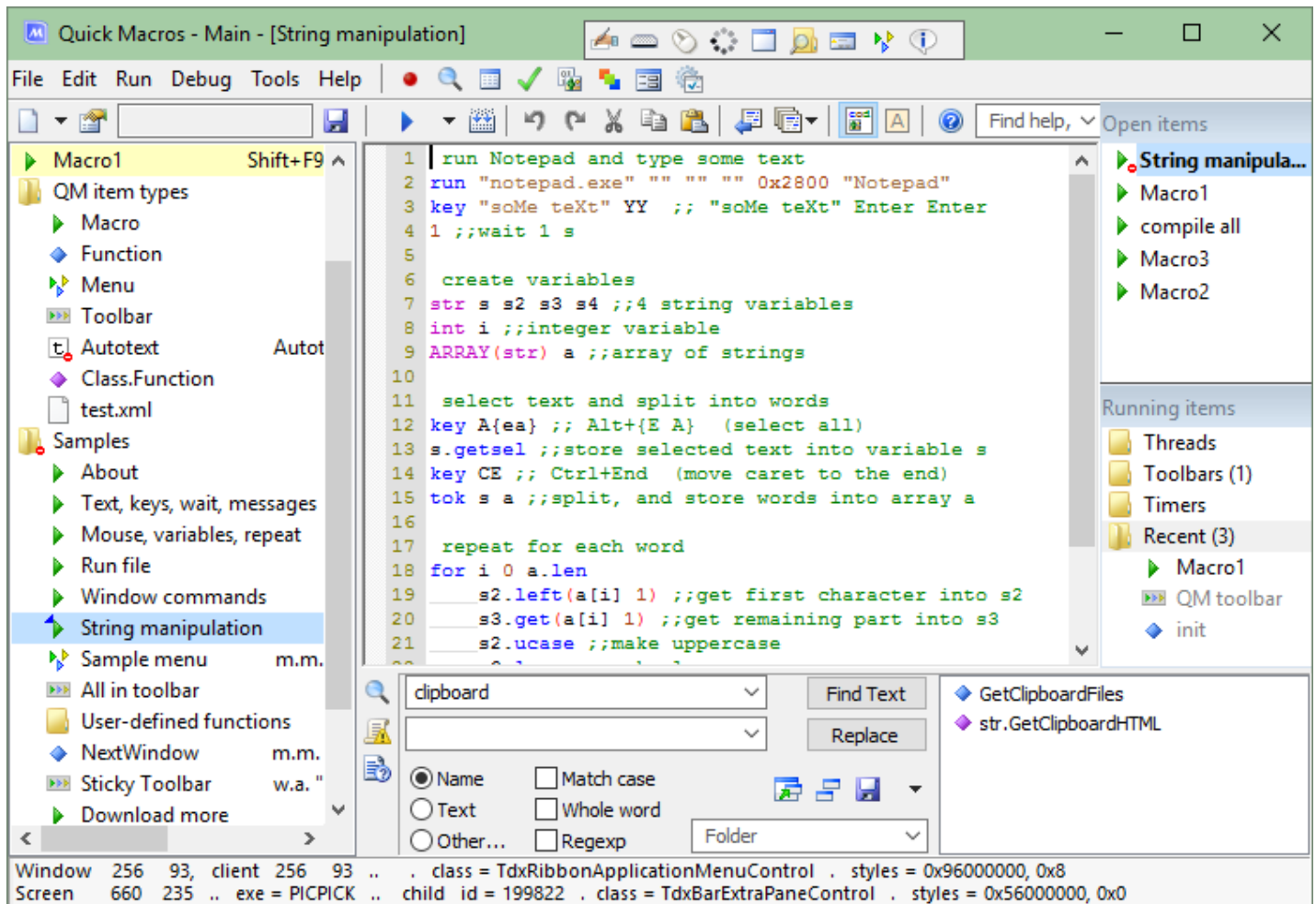
- May need to manage [QM files \(17\)](#) differently. See also: [network setup \(258\)](#), [backup \(15\)](#).
- If you have functions with same names as the new functions, will need to rename or delete them.

See also:

[What's new in versions 2.1.0 - 2.3.6 \(272\)](#)

Main window

Normally Quick Macros runs all the time. The main window is hidden most of the time. When you want to add or edit macros, click the QM tray icon . To hide the window, click the X button (Close).



Top: [Toolbars \(5\)](#), [File menu \(6\)](#), [Edit menu \(7\)](#), [Run and Tools menus \(8\)](#), [Find help, functions, tools](#)

Left: [List of items](#)

Center: [Code editor](#)

Right: [Active items](#)

Bottom: [Output](#), [Find](#), [Tips](#), [Status bar](#)

[Floating Toolbar](#)

[Tray icon](#)

[Keyboard shortcuts \(9\)](#)

List of items

Here you manage [QM items \(19\)](#) that are in currently open [file\(s\) \(17\)](#).

Click an item to open it for editing.

Middle click to close.

Right click to show the [File menu \(6\)](#) for that item. The Add commands will insert new item there.

Drag and drop to move item(s) to another place.

QM 2.4.0. Right-drag to select (check) multiple items for moving. Use Shift to add to selection, Ctrl to add/remove.

Ctrl+drag and drop to copy (clone) the item.

You can also drag and drop an item to: editor text (adds "run macro" statement), a custom toolbar (adds "run macro" button), an edit control.

QM 2.4.0. Alt+click an item to open it in the right editor. Ctrl+click to move to the primary editor. Ctrl+Alt+click to open in both editors.

QM 2.4.0. Deleted items are in the "Deleted" folder. You can: click a deleted item to see its text; undelete; delete from the folder; move items to/from the folder.

Item icons and colors

4. Main window

Each QM item type has its [icon \(19\)](#). You can change icons in the [Icons \(16\)](#) dialog.

Blue background - current item. Its text is displayed in the editor. Also there is a small triangle on the icon.

Yellow background - an open item. Read more below, in "Editor" and "Open Items" chapters.

Blue text - recently modified.

Red circle on icon - disabled trigger.

Small "S" on icon - [shared folder \(17\)](#).

Editor

Here is code (text) of currently selected item.

Right click to show the [Edit menu \(7\)](#).

Click the selection bar to select a line.

You can drag and drop the selected text.

To disable-enable a line, right click the selection bar.

To disable-enable more lines, select them (whole lines) and right click the selection bar, or press key Space or Shift+Space.

To insert or remove tabs at the beginning of multiple lines, select them (whole lines) and press key Tab or Shift+Tab.

You can open multiple items. Opening another item in the code editor does not close current item, just hides it. You can close an item to unload its text from the editor, clear its Undo history, free used memory and remove it from the Open Items list.

You can edit items in two editors. When you open an item, it is opened in the primary editor. Using the Active Items menu you can open it in the secondary editor at the right or bottom, or in both editors.

Active Items

The Open Items list can be used like tabs in a multitab editor or web browser. It shows items that are currently open. They are yellow in the list at the left. Right click an item to show the Active Items menu. Middle click to close the item. Names of edited items are blue.

In the Running Items pane you can see currently running [threads \(49\)](#), toolbars, timers and recently launched items. Right click an item to view a menu that allows you to end a thread, find a lost toolbar, reset toolbar settings, stop a timer. Items that are in private folders (in Folder Properties dialog checked "Private functions") are hidden or gray. To hide/unhide, right click a folder or empty space. Does not show threads and timers started by macros running in separate process.

Find help, functions, tools

Allows you to quickly find help, functions and tools in:

1. QM help file.
2. Floating toolbar dialogs.
3. Functions in public QM system folders.
4. Functions in specified QM user folders. To add folders, click the drop-down arrow and select Options in the popup menu.
5. Tips.
6. QM forum.
7. Internet (Google).

Type a search query and press Enter. It shows a menu of help topics, floating toolbar dialogs, functions and tips that contain the words. Also you can search in the forum and the Internet. The search query can be a question, or a task, or words in any sequence. Finds topics that contain ALL words. Also searches for similar words, eg "get", "gets", "getting". Ignores stop-words, like "a", "to", "how", "it", and nonword characters. Does not support phrase search, operators, etc. Examples:

copy files

how to move a window to a monitor

acc

dialog icon

attaching toolbars to windows

F1 and F2 here work in a similar way as in the code editor. For example, you can type a function or macro name and press F2 to open it or F1 to see its help. If it is a member function, type Classname.Membername.

Output, Find, Tips

In the Output pane you can see error messages, [out \(57\)](#) messages, and other notifications. Right click to view a [context menu \(8\)](#).

In the Find pane you can find macros, find and replace text, rename macros.

Can be used operators AND and OR. They must be in uppercase. The OR operator has higher priority.

To quickly clear a text field, middle click.

4. Main window

You can save current Find pane text and settings, and later load. Saves in current main [QM file \(17\)](#).

In the Tips pane you can learn how to work with Quick Macros.

When you type or click a function name or some other identifier in the code editor, and then press F1, its [help \(245\)](#) also may be displayed in the Tips pane.

Status bar

Here you can see mouse-related information.

First line: mouse position in window .. window name . window class . window style, exstyle.

Second line: mouse position in screen .. program .. child window id . child window class . child window style, exstyle.

Here also is displayed [short function information \(46\)](#) and other temporary information.

Floating toolbar

To enter some often used macro commands, you can use dialogs from the floating toolbar. If the toolbar is unavailable, click the "Check extensions" button in Options and follow instructions. Currently, you can use the dialogs only to enter commands, but not to edit.

Dialogs are available only for some commands. Actually there are thousands of commands and functions, although many of them can be useful only for programmers. You can find other commands in QM Help, or in [popup lists \(46\)](#), or using the "Find help, functions, tools" field, or on the Internet.

Many fields in the dialogs are optional. You can leave them empty. The required field always is at the top. For example, in the Run Program dialog, you can specify only the program. Various options initially are set to mostly used values, so you don't have to change them in most cases.

In some dialogs, you can see a Drag picture. You can drag it and drop on an object, and many dialog fields will be filled automatically (you may have to edit something in some cases).

In dialogs you can use variables. To use a variable in a dialog field where text is required, enclose it in parentheses (except when there is a checkbox or other control to specify that it is a variable). Parentheses are not necessary where numeric values are required.

Tray icon

The QM tray icon normally is blue. When QM is disabled (does not respond to triggers of keyboard, mouse and some other types), it is gray. When a macro (not a function or menu) is running, it is red.

Click to show QM window. If the tray icon is hidden (you can set this in Options), click QM shortcut (e.g., in the Start menu), or press Ctrl+Shift+Alt+Q.

Right click to show menu.

Middle click to disable or enable QM.

Ctrl+click to exit QM.

Toolbars in QM window

New macro - create new macro. Use the drop-down menu to create new [QM item \(19\)](#) of other type and use templates.

Properties - show the [Properties \(10\)](#) dialog, where you can set trigger and other properties of current item.

Trigger - [encoded \(264\)](#) trigger of current item. You can set trigger in the Properties dialog or directly in this field.

Save now - [save \(17\)](#)/apply changed text now. Changes also are saved/applied automatically: when you hide or exit QM, when you open another item or file, when a macro starts, etc.

Run - compile and execute current item.

Compile - check for errors and prepare to run. [Read more \(47\)](#).

Active items menu - a drop-down menu that contains commands to work with recently used QM items.

Images in code editor - display image [resources \(261\)](#) and files in the code editor below lines containing an image resource name or image file path enclosed in ". Displayed are [recorded screenshots \(12\)](#), images used with [scan](#), and other images.

Annotations - display annotations in the code editor. Annotations show some info about the used functions etc. This info also is in the status bar. Annotations are temporary, are applied for each document separately, and disappear when you edit text.

Find help, functions, tools - quickly find help, functions and tools. [Read more \(4\)](#).

Record - minimize QM and start [recording \(43\)](#). Alternatively, press Ctrl+Shift+Alt+R.

Find - open the [Find \(4\)](#) dialog.

My Macros - show the My Macros window. View all triggers in single list. Filter, sort, edit, run (use right-click menu). Other way to show this window - hotkey defined in Options (default Ctrl+Shift+Alt+M). When using the hotkey, it behaves slightly differently, and you can use this mode to view triggers that work with currently active program.

Tags - show the Tags dialog.

A tag is a word or phrase that can be assigned to a [QM item \(19\)](#) and later used to find QM items that have the tag. A QM item can have multiple tags. A tag can be assigned to multiple QM items.

In the dialog you can edit tags and assign them to QM items. In the right side - QM items that have the selected tag. To assign a tag to current QM item, check the tag or create new tag and check. To remove a tag from current QM item, uncheck or delete the tag. When you delete or rename a tag, the tag is deleted/renamed for all QM items that have it. Unassigned tags are removed. Tags of all currently loaded QM items (including shared and read-only) are saved in current main [QM file \(17\)](#).

Tags added in QM 2.4.0, replacing Bookmarks. Existing bookmarks converted to tags.

Resources - show the [Resources dialog \(261\)](#).

Icons - open the [Icons \(16\)](#) dialog, where you can find or create icons for menus and toolbars.

Dialog Editor - open [Dialog Editor \(63\)](#). If current macro contains a dialog definition, opens it, else creates new.

Options - show the [Options \(12\)](#) dialog.

Output, Find, Tips - see [here \(4\)](#).

Menu: File

To show this menu, you can use the menu bar or right click an item in the list of macros.

New (submenu) - create new [QM item \(19\)](#) (macro, toolbar, etc) or folder.

Disable/Enable - disable or enable trigger. A disabled macro still can be started using the Run button, [mac \(100\)](#), from menus and toolbars. You can disable single item, folder or file. To disable in some programs, use the [Properties \(10\)](#) dialog. See also [dis \(102\)](#).

Rename - rename the item. To rename multiple items, use the Find dialog.

Delete - move the item to the Deleted folder. By default, items in the folder disappear when closing file, but this can be changed in the Folder Properties dialog of the Deleted folder.

Properties, Folder properties - show the [Properties \(10\)](#) dialog for the item.

Close - close the item. It will unload its text from the editor, clear its Undo history, free used memory and remove it from the 'Open items' list. Other ways: middle-click the item in this or some other list (Open items, Find results, etc).

Collapse folders - close all expanded folders.

Copy name, Copy path - copy QM item name or path to the clipboard.

Check - select the item for moving. It allows to move multiple items. Other way: right-drag.

Move checked - move checked items to the right-click place or to the selected item's place. Other way: drag and drop.

Clone - create new identical item. Other way: Ctrl + drag and drop.

Print - print the item.

Open / New File - open an existing or create new [QM file \(17\)](#).

Recent files (submenu) - open a recently used file.

Import - add items from another QM file to current file. Or add another file as a shared file.

Export - save the selected item or folder as another QM file.

Save now - save/apply changed text. Same as the [Save button \(5\)](#) on the toolbar.

File properties - show the [File Properties \(11\)](#) dialog.

Find item in files - show dialog 'Find QM item in other files'. In the specified folder finds all QM files containing a QM item with the specified name. Then you can open the files in the File Viewer. This is useful when you need previous versions of your macros, auto-saved in files in the backup folder. Searches only in files that currently are not open in QM. Does not search in subfolders. Item name can be full or with [wildcard characters \(271\)](#); if empty, shows all files.

Close window - hide QM window. Other way: click the X (Close) button.

Exit program - exit Quick Macros. Other ways: 1. The tray menu. 2. Ctrl+click the tray icon. 3. The System menu. 4. In a macro: [shutdown -1](#).

Menu: Edit

To show this menu, you can use the menu bar or right click macro text. Some Edit menu commands are described in the [Toolbar \(5\)](#) topic. Some other menu items:

Copy for QM Forum - copy code to use in the QM forum. The format is not supported by other forums.

Copy BBCode - copy code to use in forums that support BBCode. Also can be used in the QM forum, but the above command is preferred.

Copy HTML - copy HTML.

HTML options - options for the HTML copy command. Check 'Without CSS' if you are using common CSS.

Copy CSS - copy CSS to be used as common CSS. QM creates it using settings in Options -> Editor. You can place it in a style sheet file, or in the same html file surrounded by <style> and </style>.

Copy escaped - copy text. Replace tabs and spaces at beginning of lines to commas and semicolons. Tabs and spaces would disappear if used in web pages or email messages.

List Members - show a [list \(46\)](#) of functions, variables, other identifiers.

Context Help - show help for the function or other identifier from the text cursor.

Go to Definition - show the place where the function or other identifier by the text cursor was [declared \(242\)](#). If it is not declared, QM tries to find and show it in a [reference file \(160\)](#).

Hide selected - create collapsed region from the selected text. It inserts [#region/#endregion \(180\)](#) lines at the beginning and end. This menu command also is in the right-click menu of the regions bar. QM makes the bar visible when displaying a #region line first time.

Indentation - draw vertical dotted lines (indentation guides) between indentation tab characters. It makes easier to see where big tab-indented code blocks begin and end.

Multiple selections - allow to make multiple selections. To make additional selections, select text with Ctrl (click or drag in text or selection bar). When you copy text to the clipboard, all selections are copied. When you type or paste text, the text goes to all selections.

Active Items (submenu) - commands to work with recently used QM items. Other ways to show this menu: 1. Toolbar button. 2. Right click an item in the Open Items list.

View Active Items - show/hide the Open Items and Running Items lists. You can read about the lists [here \(4\)](#).

See also: [toolbar buttons \(5\)](#)

Menu: Run, Debug, Tools, Output

Some menu commands are described in the [Toolbar \(5\)](#) topic. Others here.

Run

Show possible run-time errors - when compiling macros/functions, show possible run-time errors. Read more in [err \(129\)](#) help.

Show unused variables - when compiling macros/functions, show "unused variable" warning when a [local or thread variable \(142\)](#) is declared but not used.

- QM cannot always detect if a variable is actually used. In some cases there will be more warnings than should be. For example, some dialog variables may trigger this warning. Don't delete variable declarations if you are not sure that the variable is really unused.
- This warning is not shown for function parameters and for declarations with assignment like `int k=0` or `str s.getwintext(w)`.
- You can use `#opt nowarnings 1` to disable warnings in a function. It can be at the end of function.
- To avoid this warning when the variable is used only in code that is excluded from compiling (`#if` etc), move the variable declaration to the excluded code.

Show declarations from ref files - show declarations retrieved from [reference files \(160\)](#).

Show loaded dlls and type libraries - show names of [dlls \(153\)](#) and [type libraries \(164\)](#), when they are actually loaded.

Declarations and loaded files will be shown in QM output when compiling macros and sometimes when displaying in the code editor.

Tip: You can use [CompileAllItems \(114\)](#) to compile multiple functions.

View Active Items - show or hide 'Open items' and 'Running items' lists in QM window. In the 'Running items' list you can see currently running [threads \(49\)](#), toolbars, timers, and recently launched items. Right click an item if you want to end a thread, find a lost toolbar, stop a timer.

Auto Minimize QM - minimize QM when you click the Run button or menu item to run a macro. Only if it is a macro without option 'Run simultaneously'.

Disable triggers - when checked, QM does not respond to most trigger events (as set in Options).

- Other ways to disable-enable QM: the tray icon menu; middle-click the tray icon; Ctrl+Shift+Alt+D, [dis \(102\)](#).
- To disable triggers only in certain programs, use the [File Properties \(11\)](#) dialog.
- If you hold down F8 when QM starts, you can choose to start QM in safe mode. When QM runs in safe mode, all triggers are disabled. In safe mode, macros and functions will not run whatever way is used to launch them.

Make exe, Run exe - [create or run .exe file \(51\)](#) from current macro/function.

Debug

Run or control macro in [debug mode \(99\)](#). Set debug mode options.

Tools

Record menu - records menu command ([men \(80\)](#)) when you click a menu item in some window. Not all menus can be recorded. If does not record, it means that [men](#) cannot be used with that menu. When recording, the window does not receive menu commands, unless you click with Shift.

COM libraries - view registered COM type libraries and ActiveX controls. Insert type library declaration statement ([typelib \(164\)](#)). Register/unregister COM components.

Portable QM - install [portable QM \(259\)](#) in a removable drive.

Output

Open selected item - open QM item whose name is fully selected in the QM output.

8. Menu: Run, Debug, Tools, Output

Log -> Window events - when a window is activated, created, etc, display its handle, class and name in the Output. You can use this when it is difficult to make a window trigger.

Log -> Acc. trigger events - log accessible object events to get information required for accessible object triggers.

History - displays last 32 QM output events. You can see where and when was called [out](#). For macros running in separate process QM logs less info. Some right-click menu items: Locate - open the macro/function and show the statement that generated the output event; Caller - open the macro/function that called that function.

QM keyboard shortcuts

Most QM main window keyboard shortcuts are given in menus and tooltips. Here are the others:

Alt+Right	Set keyboard focus to the code editor.
Alt+Left	Set keyboard focus to the list of QM items.
Alt+Up	Set keyboard focus to the 'Find help, functions, tools' field in the toolbar.
Alt+Down	In some edit fields shows a popup list.
Enter	In some controls - focus the editor, like Alt+Right. Try Ctrl+Enter if don't need it.

Also you can use keys that are common in navigation: Applications (context menu), Up, Down, etc.

Default global hotkeys (Options -> Hotkeys):

Ctrl+Alt+Shift+Q	Show Quick Macros main window.
Ctrl+Alt+Shift+M	Show My Macros window.
Ctrl+Alt+Shift+D	Enable/disable triggers.
Pause	End macro. Modifier keys (Ctrl, Shift, Alt, Win) are not allowed because they would interfere with macro-pressed keys. If you press the key while a macro is running, the macro stops and the key is blocked (not passed to the active window). Otherwise the key is not blocked. The hotkey sent by key is ignored when using low-level keyboard hook (see Options -> Triggers).
Ctrl+Alt+Shift+R	Start or retry recording.
Ctrl+Alt+Shift+W	Quickly record single command, where <i>window</i> argument is window or control from mouse pointer. The hotkey works only when Quick Macros window is not hidden.

Properties

Here you can set a trigger and other properties of the current [QM item \(19\)](#).

Trigger

A trigger is an event that causes the item to run. Here you can set [hotkey \(30\)](#), [mouse \(31\)](#), [window \(32\)](#), [QM events \(33\)](#), [file \(34\)](#), [event log \(35\)](#), [process \(36\)](#), [accessible object \(37\)](#), [autotext \(29\)](#) or a [user-defined \(39\)](#) trigger. Here you can also set [command line \(18\)](#) triggers (scheduler, shortcut) or copy the command line string to use by other program that can launch QM using command line or SendMessage. Other ways to launch macros: [menus \(22\)](#), [toolbars \(23\)](#), [text \(29\)](#), some functions ([mac \(100\)](#), [tim \(92\)](#), [RunTextAsMacro](#), etc) and the Run button on the QM toolbar.

Programs

Program where this trigger works. Must be only filename, without path and .exe. Can also be several programs, comma-delimited. Example: NOTEPAD,CALC. You can see program name in QM status bar when mouse pointer is on a window of that program. Programs also can be set for all items in [folder or file \(11\)](#).

FF

Assign a [filter function \(40\)](#).

Schedule

Schedule this QM item to run at a specified time. QM uses Windows Task Scheduler, and you actually create its scheduled tasks. See also: [command line \(18\)](#), [unlock computer \(268\)](#). This feature is unavailable in [portable QM \(259\)](#).

Shortcut

Create a shortcut on the desktop, Start Menu or Quick Launch toolbar. When you click the shortcut, starts QM (if not already running), and launches this item. This feature is unavailable in [portable QM \(259\)](#).

Cmdline

Format [command line \(18\)](#) to launch this item, and copy it to the clipboard.

Multi-trigger

Create a shortcut-macro and open its Properties dialog.

A macro itself cannot have multiple triggers, but you can use shortcut-macros and assign triggers to them. A shortcut-macro is a macro that begins with space, slash and target macro name or [GUID \(249\)](#), like `/TargetMacroName`. When launched, actually runs the target macro. If the shortcut-macro has a trigger, the trigger will launch the target macro.

My Macros

See what triggers of selected type already exist (to avoid duplicate triggers).

Disable

Disable trigger. Disabled item still can be started using the Run button, [mac](#), from menus and toolbars.

Common properties

These properties are common to all item types (macros, menus, etc).

Template

If checked, current item will be added to the 'File\New\Templates' menu. If item name is "Macro", "Function", "Menu", "Toolbar", "Autotext" or "Member", it replaces default "File\New" menu item. If there are multiple templates where name begins with same word followed by space or underscore, they are added to a submenu.

Macro (20) properties

If a macro is running

Select what to do when this macro is launched while a macro (this or other) is running. By default multiple macros cannot run simultaneously like [QM items \(19\)](#) of other types (e.g., functions).

QM 2.4.2. Added option 'Run simultaneously'. In previous QM versions, to run simultaneously, would need to convert the macro to function. Now you can use this option instead. The macro will have some traits of functions, read more [here \(20\)](#).

Convert to function

Converts the macro to [function \(21\)](#). May change its name because it must be a valid identifier (no spaces etc).

Function (21) properties*Allow single instance*

If checked, this function will not run when you launch it while it (its [thread \(49\)](#)) is already running. This is applied only to normal threads, but not to special threads and functions called from code. Or you can add this at the beginning of function's code: `if (getopt (nthreads) > 1) ret;`

Use as filter function

If checked, this function can be used as [filter function \(40\)](#). Filter functions are used with triggers, to narrow trigger scope and achieve various other effects.

Properties common to macros and functions*Can run*

Set allowed launch methods. For example, to prevent accidentally launching the function, you can select "No, can only be called" or "Yes, except the Run button". You can also set to run some other macro or function instead. This option is saved in macro text (the first line), so you also can quickly edit without the Properties dialog. [Read more \(21\)](#).

Run in separate process

Read in [Make exe \(51\)](#) topic. To run a macro in separate process, QM creates and runs exe or qmm file.

Run as

UAC integrity level of the macro process. Read [Make exe \(51\)](#), [Vista/7/8/10 \(277\)](#).

Don't run in background; If computer locked, unlock

Read [Unlock computer \(268\)](#) topic.

Menu (22) properties

Here you can set [menu properties \(28\)](#).

Toolbar (23) properties

Here you can set [toolbar properties \(26\)](#).

Autotext (29) properties

Here you can set autotext list options.

Member function properties (157)**Folder properties, File properties (11)**

To set folder properties, right click a folder and select Folder Properties in the menu.

Folder properties, file properties

Folder properties

To show the Folder Properties dialog, right click that folder in the list of macros and select menu item "Folder properties".

Description - text that is shown in the QM file viewer when importing or opening. Also, in tooltips (until first empty line).

Scope of triggers - one or several programs, with which work or not work triggers (keyboard, mouse and window) of all items in the folder and subfolders. Several program names must be separated with comma. Example: NOTEPAD,CALC .

Read-only - items in this folder and subfolders cannot be modified (text, properties, etc), deleted, renamed, inserted new items, etc.

Disabled items - all items in the folder and subfolders are disabled.

Private functions - functions from the folder and subfolders are not displayed in [type-info popup lists \(46\)](#), unless you are editing an item that is in parent folder or its subfolders, or use Ctrl+Shift+. . These functions still can be used everywhere. Also these functions and other items are hidden or gray in "Running items" pane and Threads dialog.

Application - [obsolete](#).

This is obsolete and may be incompatible with some other QM features added later. Initially added to create separate scopes for variables, functions and some declarations. Now instead can be used classes, thread variables, sub-functions.

Adds these features:

1. Functions that are in this folder can be called only from functions that are in this folder.
2. The first (topmost) function always begins execution. It is like an application's main function.
3. Local variables declared in the main function can be used in other functions.
4. Type libraries and reference files declared in the main function can be used in other functions.

Note: Functions that are compiled at run time don't have these features. For example if a function is called using [call](#) with function's name as string.

Note: Don't call the main function recursively, because then its local variables are different every time, because of variable [scope rules \(142\)](#) (local variables have higher priority). For example, it happens if the main function is a dialog procedure that also shows the dialog.

QM 2.4.1: Fixed bug: in class member functions the 3 feature did not work.

File properties

All properties are same as folder properties, but applied are only "Description", "Scope", "Read-only" and "Disabled items". Other properties are applied to imported folder when this file is imported into another file.

Options: General

Tray icon - uncheck this to remove QM tray icon. To open QM window when the icon is removed, press Ctrl+Alt+Shift+Q. Or, click QM icon in Start Menu -> Programs -> Quick Macros 2, or in Windows Explorer (that is, run QM when it is already running). Also you can set to show the icon only when a macro is running. Then the icon does not respond to mouse clicks.

Run at startup - run QM when Windows starts. Administrators can change this setting for all users. If set for all users, QM is launched using the registry. If not set for all users, each user can separately change this setting, and QM is launched using the Startup folder. This feature is unavailable in [portable QM \(259\)](#).

Check for new QM version - check for new QM version when QM starts. If there is new version, shows it in QM output.

Unicode - if checked, QM interprets text in macros and QM user interface as [Unicode \(267\)](#) UTF-8. If unchecked - as ANSI, like in older QM versions (< 2.3.0) that did not support Unicode. After checking the checkbox and restarting QM, click the small arrow button to convert your macros to Unicode UTF-8.

Run as - [UAC integrity level \(277\)](#) of QM process. This feature is unavailable in [portable QM \(259\)](#) and on Windows XP.

Hybrid paste in menu/TB/autotext - use [hybrid paste \(58\)](#) in menu, toolbar and autotext list items where [paste/outp](#) omitted. For example, `:"text"` works like `:paste+ "text"`. If unchecked, it works like `:paste "text"`.

Erase category when item selected - when you select something from the list of [category \(159\)](#) items, erase category name.

Disable "lock foreground window" - allow background applications to activate windows. Normally, should be unchecked. Try to check this if [act](#), [run](#) or some other commands fail to activate window.

Show hidden Running items - in the Running items pane show all items. If unchecked, hides items that are in private folders (in Folder Properties dialog checked "Private functions"), eg QM extensions (System). To quickly hide/unhide, right click a folder or empty space in the Running items pane.

List with [+] boxes - set view and behavior of the main list of QM items. In [+] mode it's easier to edit item name - click two times.

On error - open macro containing error, and optionally show QM window when it is hidden.

Layout of toolbars - a name of an alternative layout (configuration) of custom toolbars on this computer/account. This can be useful if you use QM on multiple computers with the same QM file, and want to have different positions/sizes/styles of your toolbars on each or some computers. For example, you can use default layout (no name) on your main computer, layout L1 on computer B, layout L2 on computers C and D. All toolbar position/size/style data is saved in current QM file, but separately for each layout. QM saves this data for a toolbar when hiding it. Toolbars starting first time in a custom layout inherit this data from the default layout.

Record images - set to take screenshots (small images) when recording mouse clicks or using the Mouse dialog. QM saves the recorded images in [macro resources \(261\)](#) with names that begin with "~:", and inserts the names in macro text as comments. Auto-deletes unused screenshot resources, for example when you delete macro text containing "~:..." and then close the macro. To show/hide images in the code editor, use toolbar button 'Images in code editor'.

Enable Chrome acc when it starts - automatically enable accessible objects in Chrome web page area. It works only if Chrome process starts while QM is running. It can make large web pages load slower. See also [using accessible objects in web pages \(84\)](#).

Check extensions - check whether [QM extensions \(50\)](#) are working.

Export settings - export QM registry settings to a reg file. [Read more \(258\)](#).

Run-time options - shows how to change run-time options. In QM 2.3.4, global run-time options have been removed from the Options dialog. Now can be changed with [RtOptions \(114\)](#). For backward compatibility, options set in previous versions are still used.

Debug - some options that can help to debug QM or your macros. Select the 'dump exceptions' options to create crash dump files on every handled and unhandled exception, including QM internal exceptions and exceptions in DLLs and other components. Exceptions may be the reason of QM crashes or other weird behavior. You can attach the dmp files to the email when reporting a QM bug/crash/etc. You can delete the Debug folder when not needed. The options are not applied to

macros running in other processes.

Options: Triggers

A trigger is an event (e.g. hotkey) that causes the macro to run.

Active triggers - select what trigger types are enabled when QM is in "enabled" state (blue tray icon) and when in "disabled" state (gray icon).

- Triggers of "QM events" type are always enabled.
- If you uncheck "Autotext lists", all [autotext lists \(29\)](#) will not work, regardless of trigger.

Use low-level hook - usually makes triggers more reliable. Also, triggers will work in console windows on all Windows versions.

- Keyboard - used with [keyboard \(30\)](#) triggers. Allows you to steal hotkeys from Windows (Win+F, etc) and applications. However it may be incompatible with QM extensions that use direct input or raw input.
- Mouse - used with [mouse \(31\)](#) triggers. Uncheck if you notice that mouse pointer occasionally stops moving.
- Other - used with [window \(32\)](#) and [accessible object \(37\)](#) triggers. Normally should not be checked. Try to check if you notice incompatibility of some kind with some applications. If checked, "destroyed" triggers are unreliable.

Mouse sensitivity - adjust the required amplitude and speed of mouse movement triggers.

System - open the system Mouse Properties dialog.

- Recommended mouse pointer and hardware settings: average or high speed, no acceleration or low acceleration, average or high sample rate.

Autotext list: additional non-delimiter characters - additional non-delimiter characters used with [autotext lists \(29\)](#).

- Default characters are all alphanumeric characters.
- For example, if you want to add _ and #, enter _# in this field.

Options: Security

Encrypt macro - encrypt or decrypt current item's text. Decrypting requires the same password that was used when encrypting.

- If "All containing text" is checked, instead encrypts or decrypts all items that contain the text. The text in items must be a full comments line, ie a line that begins with a space or semicolon. Don't include the space/semicolon. When decrypting, decrypts only items that are encrypted using the same password.
- Note 1: If you encrypt a macro that contains a [dialog \(63\)](#) definition, the dialog will not work, unless the dialog definition is assigned to a variable which is passed to [ShowDialog](#).
- Note 2: When you click function name in code editor and press [F1 \(245\)](#), you can see its parameters and help section (comments above code), even if the function is encrypted.
- Note 3: By default, QM creates backup files. They may contain unencrypted (older) versions of your encrypted macros. The bad is that somebody can see the macros if you don't delete the files. The good is that sometimes there you can find unencrypted versions of your encrypted macros in case you have lost password. Usually QM eventually deletes the files. Look in [Options -> Files \(15\)](#).

Encrypt the password to use with a function - encrypt a password for using with a function that supports encrypted password. [Read more \(150\)](#).

See also: [file viewer](#), [shared files \(17\)](#), [disable folder or file \(11\)](#), [lock toolbar \(26\)](#).

Options: Files

Auto backup - create backup copies of currently used [QM files \(17\)](#) in the backup folder.

- Default folder is My Documents\My QMBackup.
- QM 2.4.0. Backups shared files too.
- QM 2.4.0. If you use some other software to backup QM files, note that currently open .qml files usually don't contain new changes. QM at first writes new changes to temporary .qml-wal files in the same folder. Let the backup software copy the files too. See also [_qmlfile.FullSave \(260\)](#).

Older files - keep older backup files of specified age or less.

- For example, if it is 15m 1h 4h 1d, there will be maximum 5 backup files for a QM file. The older four files are maximum 15 minutes, 1 hour, 4 hours and 1 days older (sometimes more older). Other auto-created backup files are auto-deleted.
- QM 2.3.5. In the older files list, you can append m, h or d to specify minutes, hours or days. If not specified, h assumed. In previous QM versions the numbers are always hours. Also there are more changes in QM 2.3.5. Different backup algorithm and file names.
- When you manually backup with the 'Backup now' button, the backup files have different names and are not auto-deleted.

My QM folder - folder where QM and QM extensions will store various files. This folder also contains the default main QM file. In macros, where file path is used, this folder can be specified as \$my qm\$ (e.g., "\$my qm\$\my file.txt"). Default path is \$personal\$\My QM (My QM folder in My Documents). If there are multiple user accounts, each user has its own My Documents folder.

Icons

The Icons dialog can be used to find or create icons for [toolbars, menus \(24\)](#), [QM items \(19\)](#) and shortcuts.

Left side - icon browser

Here you can find icons, assign them to menu items, etc.

Browse - open an icon file. You can select files and view their icons without closing the Open dialog. The small button shows menu of recent files. To add a file to the menu or move it to the top, select the file in the 'Open icon file' dialog and press OK.

Actions - shows menu of icon actions. Most actions use the current icon. The icon is displayed in the dialog, and its file name and index is in the title bar.

Set icon... - assign the icon to a menu/toolbar item, or to the current QM item, or to items of the same type as the current QM item, etc. For menu/toolbar items, inserts the icon file path into the editor text, in the line that contains the text cursor.

Reset - assign default icon.

Edit - open the icon in the icon editor at the right. If the icon is from a library (dll, exe, etc) or is not compatible (read below), opens it as new.

Edit in external editor - open the icon file in another program that is specified in Options -> Tools. If the icon is from a library, at first saves it to an ico file.

Save as ico file - if the icon is from a library, saves it to an ico file.

Copy path - store path of the icon to the clipboard.

Paste path - open icon file whose path is in the clipboard as text.

Set default save folder - choose a folder where Save As dialogs will start. It is not used by the Browse button.

Refresh icons - reload changed icons of QM items and common icons. To improve performance, icons are cached.

Imagelist Editor - creates imagelists that can be used with various controls in dialogs.

Right side - icon editor

Here you can create simple icons for menus, toolbars and QM items.

When you create or edit an icon, click "Save" or "Save As". Then you can use the Actions menu to assign the icon to a menu item, etc. QM does not automatically save icons or prompt to save when closing the editor.

The editor saves icons to *.ico files that contain one small icon (16x16 pixels, 16 or 256 colors). Usually icon files contain two or more icons of various size and color count. To prevent overwriting such icons, the editor opens them as new.

Tips

Right click the grid to undo. Ctrl+left click to pick color.

Drag the preview icon to capture an icon or to move the image in the editor. Drag with Ctrl to capture-add. While capturing, you can see top-left pixel's [color \(240\)](#).

The eraser color will be transparent.

When drawing, you can keep the Color dialog open and use it as alternative palette.

How QM uses custom icons

In QM 2.3.0.3 and later, you can set custom icons for QM items and item types. Also changed the way QM uses icons in menus, dialogs, etc.

In menus and toolbars: For items are used the same default icons that are displayed in QM. Also now you can set default icon for submenus.

[ShowDialog](#), [AddTrayIcon](#), [MainWindow](#): As default icon is used the icon of the function or macro that started the thread. In

exe (51) - exe icon.

Paths of icons of QM items are saved in the QM file together with items. Paths of icons of QM item types are saved in the registry.

QM macro-list files

QM stores multiple macros and other [QM items \(19\)](#) in a file with .qml extension. When starting, it loads the last used *main* file (initially it is Main.qml). Also loads System.qml (QM extensions) and optionally several other *shared files* (read below).

QM closes the files when exits or when you open another file. Then it ends running threads, closes toolbars, deletes variables and declarations.

All changes are saved automatically. QM item management changes (add/delete/move items, etc) are saved immediately. Text changes are saved with some delay, also when you click the Save button. Less important changes, such as open/expanded states, are saved when closing file or hiding QM.

What's new in QM 2.4.0

Changed file format. Now it is a [SQLite](#) database.

When opening an old version file, QM converts it to the new format. Older QM versions cannot open files of the new format. The file extension is the same. When converting, QM backups the old file.

While a file is open in QM, other QM instances cannot open it, unless all open it as read-only. Read more below about locally cached copies. See also [network setup \(258\)](#).

When you edit macros etc, QM at first writes new changes to a temporary file (.qml-wal). Transfers the changes to the true file (.qml) when closing it, also when hiding QM and when there are many new changes. See also [_qmfile.FullSave \(260\)](#). There may be more temporary files. See also [backup \(15\)](#). Note: the temporary files sometimes may grow to 4 MB or more, it is normal.

Now custom toolbars save their position, size and style in file, not in registry. All currently loaded toolbars save in current main file (the file that is currently open in QM), even those that are in currently loaded shared files. See also: [Options -> Layout of toolbars \(12\)](#).

Why file format has been changed:

- The old format was too limited and unreliable.
- Now much easier to add new QM features.
- Now faster when file is big and disk slow.
- QM does not have to load whole file into memory, and recreate from memory when saving.
- QM does not have to save some data in other files.
- You can use [QM file management functions \(260\)](#) or [Sqlite](#) class to manage macro resources, custom data, etc.
- You can open and edit QM files in a database manager program. Recommended - SQLite Expert. Exit QM, it locks the file.

Shared files

A *shared file* is another QM file loaded into a virtual folder in the main file. Such virtual folders, aka *shared folders*, have S letter on the folder icon. To add a shared folder, use menu File -> Import. If a shared folder is not at the root (is in another folder), QM does not load the file.

A shared folder is like a link to a shared file. Items that you see in the folder are saved in the shared file, not in the main file. But you can use, edit and manage them like they would be in the main file. QM automatically saves changes and makes backups (QM 2.4.0).

The System folder (System.qml file) is a special shared folder containing QM extensions. QM always adds it to every opened main file.

QM 2.4.0. There are two other special shared folders - Temp and Deleted. The Temp folder contains temporary items created by QM and macros. It is an in-memory database, not a real file, therefore all items in it disappear when closing the main file. The Deleted folder contains items that you delete. It is similar to the Recycle Bin in Windows. By default it is an in-memory database too, but you can right-click the folder and specify a file.

QM 2.4.0. If a shared file is on another computer, QM loads its locally cached copy in read-only mode. Then QM does not have to wait or fail when the file is unavailable. You can enable or disable this in the Folder Properties -> Shared File dialog.

Installed files

Quick Macros installs several .qml files:

\$my qm\$\Main.qml (Main.qml in My Documents\My QM folder) - your initial main file. This file is NOT replaced when upgrading QM.

\$qm\$\Installed Files\Main.qml - original Main.qml file of current QM version. This file is replaced when upgrading QM. Don't use it as main file.

\$qm\$\System.qml - [QM extensions \(50\)](#). This file is replaced when upgrading QM. Don't use it as main file.

Notes:

\$qm\$ - folder where QM is installed. Probably C:\Program Files (x86)\Quick Macros 2.

\$my qm\$ - your personal folder for QM files. By default, it is ...My Documents\My QM.

You can find more files in QM [forum](#).

How to transfer Quick Macros, along with macros and other QM data, to another computer

To transfer Quick Macros from computer A to computer B:

1. Install Quick Macros on B.
2. Make sure Quick Macros on B is not running. If running, exit (menu File -> Exit program).
3. On computer A, locate My QM folder. It is in My Documents, unless you changed it in Options.
4. Copy the My QM folder to My Documents on B. If My QM already exists there, at first delete it.
5. If your macro file on A is not in My QM, copy it to B too. You can see your macro file path in menu File -> Recent -> first entry.
6. Run Quick Macros on B.
7. If it did not open your file, open it using menu File -> Open/New File.

See also: [network setup \(258\)](#).

The File Viewer

When opening an unknown QM file, at first it is opened in the File Viewer. It allows you to safely preview file content. Then you can open, import, add as shared file, or cancel. Downloaded files may contain malicious, incompatible (duplicate functions, triggers, etc) or useless code. If you think the file is unsafe or useless, you can cancel. Or, you can import it with disabled triggers. Also, you can import only selected items.

The File Viewer also is used to preview the file when importing, to restore items from the backup folder, to view other files, to find an item in multiple files in a folder.

A file is considered unknown if it isn't in the recent files list (menu File -> Recent) and is not currently open in QM.

How to restore a corrupt file

It is possible to [corrupt](#) a QM file (SQLite database). Then QM fails to load the file or fails to open/save/delete/etc macros and other data stored in the file.

Usually the best way to fix it - replace the file with the newest backup file. You can find backup files in folder ...Documents\My QM\Backup, unless the path is changed in [Options -> Files \(15\)](#).

Or you can try to [repair](#) the file.

See also: [command line \(18\)](#), [File menu \(6\)](#), [file properties \(11\)](#), [network setup \(258\)](#), [backup \(15\)](#)

Command line parameters

[Command line](#)
[Shortcut](#)
[Task Scheduler](#)
[SendMessage](#)

Command line

Other programs can run QM macros using command line parameters. To create command line for current macro, click the Cmdline button in Properties.

Syntax of QM command line:

```
[V] [E] [S] [Q c] [T] [N "networkfile"] [L/I] "file" [M[S] "macro" ["command"]] [[C
command] | [A[(sep)] arguments]]
```

Here used gray symbols aren't part of command line:

- `[]` encloses optional parameters.
- `|` is logic "or".

Quotes are required. Parameters enclosed in quotes cannot contain quotes. Order of parameters is arbitrary, except C and A.

To pass command line parameters to QM, use qmcl.exe, not qm.exe. This small program quickly starts, relays the command line to QM and exits immediately. You can use qm.exe too, but some parameters (MS, N, T) are supported only by qmcl.exe.

V	Show QM window.
E	Exit QM when macro ends, if QM was not running. Tip: instead you can create exe from the macro and run it with a command line.
S	Indicates that QM started at Windows startup. QM uses it for trigger 'Windows started'.
Q c	Use character c instead of " to enclose strings. For example, instead of M "Macro" you can use Q ^ M ^Macro^.
L "file"	Open specified QM file (17) . <ul style="list-style-type: none"> • Use LI to import. • See also: SilentImport (114). • Opens in the File Viewer if LI or if the file is unknown (not in the recent files list in the File menu).
M "macro" "command"	Run macro. <ul style="list-style-type: none"> • Can be macro name or GUID (249). • You can pass some string (command) to the macro. It will be in variable _command (144). • Use MS to run synchronously. Qmcl.exe waits while macro is running and returns its return value (ret). • QM 2.4.0: Not supported with L and LI.
C command	Alternative way to pass command to macro. Here command isn't enclosed in quotes. It itself can contain quotes.
A arguments	Pass arguments to macro. Macro receives them through the function (152) statement. Use spaces to separate arguments. If an argument contains spaces, it must be enclosed in quotes or parentheses. An argument cannot contain quotes (unless it is enclosed in parentheses) and parentheses (unless it is enclosed in quotes). Numeric integer arguments can be numbers, named constants, global variables and expressions with operators. Expressions with operators are evaluated from left to right regardless of operator priority.
A(sep) arguments	Alternative way to pass arguments. Here sep is some string that is used to separate arguments. Arguments can contain any characters, including "()".
T	Used with scheduled tasks.
N "networkfile"	Wait for a file (networkfile) that is on another computer on network (258) and may be still unavailable. Qmcl.exe waits for the file (max 5 minutes) and launches QM when the file becomes accessible. If QM is already running, or is started while waiting, qmcl.exe exits and does not pass command line to QM.

Examples:

```

M "Macro"
M "Macro" "some string"
M "Macro" C some string
M "Macro" A "some string" 1 55.5
M "Macro" A(,,) some string,,1,,55.5

```

Or whole command line can be a macro or a file. If the file is not .qml, adds it as [file link \(19\)](#) and opens in QM (QM 2.3.0). Special folders are not supported. Can be several paths separated by spaces. Examples:

```

"Macro10"
"c:\program files\qm\list3.qml"
c:\Users\G\Desktop\test.txt

```

Disabled items don't run from command line trigger.

QM command line can be used with desktop shortcuts, Task Scheduler and other programs that can launch programs with a command line.

Shortcut

To create a shortcut to launch current macro, click the Shortcut button in the Properties dialog or in the Icons dialog. You can optionally assign a hot key to the shortcut. The hotkey will work even when QM is not running (QM will start).

Task Scheduler

Quick Macros does not have its own scheduler. Instead, it uses Windows Task Scheduler service. It allows to run a macro even when QM is not running (QM will start). To create or edit a scheduled task that runs current macro, click the Schedule button in the Properties dialog.

Although this is an external trigger, QM displays schedule times in the list of macros.

QM 2.4.2. Creates and manages scheduled tasks configured for Windows Vista and later. Also supports previously created XP tasks.

See also: [unlock computer \(268\)](#)

SendMessage

Other programs or scripts can launch QM macros more efficiently, by sending a message to the QM main window. Syntax:

```
SendMessage(hwndQM, WM_SETTEXT, mode, commandline)
```

commandline - QM command line string. Don't forget to include quotes.

mode:

1	run macro or other item asynchronously. QM launches macro and returns control to the calling application.
2	run function or macro synchronously. SendMessage returns function's return value. This mode forces to run even if item is disabled, or QM command line triggers are disabled.
3	similar as 2, but function runs in QM main thread. This is faster, because new thread is not created. Can also be used by a macro to call a function that must run in QM main thread. Be careful when creating macros that run in QM thread. Don't use wait commands and other long-running code, because QM may stop responding.

Don't use PostMessage or wParam=0. Can use SendMessageTimeout. Can use FindWindow("QM_Editor", 0) to get QM window handle.

Example in QM: `SendMessage (hwndqm WM_SETTEXT 3 "M 'func' ")`

QM items

You can create items of these types:

► **Macros** are used to execute macro commands and functions. By default, only a single macro can run at a time. Red tray icon indicates that a macro is running. You can stop it manually: press the Pause key.

QM 2.4.2. Added option 'If a macro is running, run simultaneously'. Read [here \(20\)](#).

◆ **Functions** can be called from code (from macros, functions etc). Functions also can be started like macros (not called from code).

Macros and functions can contain the same code. The main difference between macros and functions is for what purposes they usually are used:

- Macros are used to automate something. Especially if they interact with windows (send keys, etc). They are safer than functions (single instance, red tray icon, stop with Pause). Often used just to test some code.
- Functions are mostly used for:
 - To execute the same code in multiple places (call function containing the code).
 - When a function is required, for example for callback functions.
 - For background tasks and other tasks that run long time, unattended, like applications or services. Usually such tasks don't interact with windows and don't interfere with user actions and with each other. Macros with option 'Run simultaneously' also can be used for this.

Default speed ([spe \(90\)](#)) of functions and member functions is 0. Default speed of macros and items of other types is 100, and can be changed with [RtOptions \(114\)](#).

See also: [sub-functions \(182\)](#).

◆ **Member functions (157)** are functions that belong to a class. Same as functions, but are called differently. Cannot have a trigger or start a thread.

► **Pop-up menus** are used to launch macros, files, execute other commands.

► **Toolbars** are used to launch macros, files, execute other commands. Toolbars can be associated with windows (run automatically when window appears, and disappear when window closed), or free (once launched, exist all the time, on top of other windows).

▢ **Autotext lists** (aka TS menus) are used to execute commands when you type certain text. Can be used for text replacement/autocomplete/autocorrect.

You can create [folders \(11\)](#) to store items.

An item has a name, can have a trigger (an event that launches it), text (code) and several other properties.

To add new macro, click the 'New macro' button on the toolbar. To add item of any type, click the small arrow beside it and select from the menu. Or right click somewhere in the list and select from the New submenu. Then you can enter name in the small edit field in the list. To set trigger and other properties, use the Properties dialog (click the Properties button on the toolbar). To delete an item, right click it in the list of macros and click Delete in the menu, or click and press the Delete key.


QM 2.3.0. Added new item type - **file link**. File links are used to open text files in QM for viewing and editing. A file link is like a shortcut to the file. File contents is not stored in the [QM file \(17\)](#). When you delete or rename the item, it does not affect the file. To create a file link, drag and drop a file from Windows Explorer to the list of macros in QM. Or open the file with QM, for example drag&drop on QM icon. Or use [newitem \(108\)](#). An item of this type can have a trigger or can be launched using the Run button. However by default it does nothing. To run the file, or do something else with it, create a function that does it and assign trigger '[QM events -> file link run](#)' ([33](#)).

QM 2.3.0. QM items can have custom [icons \(16\)](#).

Macro

A macro is a list of commands, such as "type text", "run file", and other statements. Read [Syntax \(44\)](#) and [Help and type-info \(46\)](#) topics.

You can enter macro commands directly in the editor. You can also [record \(43\)](#) keys and mouse events, or use dialogs from the floating toolbar.

When you launch a macro, QM compiles and executes it. The tray icon  indicates that a macro is running. Macro stops when last command was executed, or some command tells to do so, or when an error occurs. To stop a running macro manually, press the hotkey that is set in Options (default is Pause).

By default, only a single macro can run at a time. To run multiple threads simultaneously, set 'Run simultaneously' option, or use a [function \(21\)](#) instead.

QM 2.4.2. Added option 'Run simultaneously'. In previous QM versions, to run simultaneously, would need to convert the macro to function. Now you can instead use this option in the Properties dialog. The macro will have some traits of functions:

- Can start even if a macro is running. Other macros can start while it is running.
- When it runs, the QM tray icon does not change.
- The thread (running macro) cannot be ended with the 'End macro' hotkey (Pause by default) or menu command. To end thread you can use the 'Running items' pane or the 'Threads' dialog (look in the QM tray icon menu).
- When ending thread, does not release modifier keys left pressed by code like `key+ C`.
- The 'Auto minimize QM' setting is not applied.

Function

See also: [function tips \(150\)](#), [about functions \(149\)](#), [declaration \(parameters etc\) \(152\)](#), [programming in QM \(45\)](#), [class member functions \(157\)](#), [sub-functions \(182\)](#)

A user-defined function is a [macro \(20\)](#) that can be called from other macros.

Differences between macros and functions:

1. Functions can be called from macros and functions. It allows you to reuse the same code in multiple places: use function's name instead of all the code. A function can [receive arguments \(152\)](#) and [return some value \(130\)](#). In code, function names have [this color](#).
2. Like macros, functions also can be started by the user or a trigger. Function can run simultaneously with other functions or macro.
3. When a function runs, the QM tray icon does not become red.
4. You cannot stop a running function by pressing Pause key. Use the Threads dialog (in the tray icon menu) or the Running Items list (menu Run -> View Active Items). Also, you can use special code, like `ifk(C) break` (if Ctrl pressed, exit [for](#) or [rep](#) loop).
5. Initial macro speed is 100. Initial function speed is 0 (no autodelays). It can be changed with [spe \(90\)](#).

By default, functions can be launched by the user or a trigger, like macros. Often you want to prevent launching a function accidentally, because the function is designed to be called from code. Use the Properties dialog, which inserts special line at the beginning of function's text. If the line begins with space and slash (/), the function runs only if it is called as function (from macro, other function, or as callback function). If you start it using the Run button, trigger, [mac](#), etc, it does not run. Example (beginning of function's text):

```
/
function ...
...
```

If function (or macro, or member function) begins with space, slash and name of other item, then, when you press the Run button, runs that item. This is useful when debugging a function that must be called from code. Example (beginning of function's text):

```
/test
function ...
...
```

When you launch this function, runs macro "test", that may call this function. If backslash (\) is used, it only prevents starting the function when you press the Run button, but the function can be started using e.g. [mac](#) or trigger.

QM [compiles \(47\)](#) functions and macros on demand.

It is possible to change (edit and save) function's code at run time, but changes are not applied while that function (or a function it called) is running. For example, if you edit/save a running function that repeatedly executes code using [rep](#), the changes are applied only when you run the function next time. But if you edit/save a running function that contains a dialog procedure, the changes are applied immediately, because the function is called repeatedly and usually quickly returns.

Pop-up menu

Pop-up menus are used to launch macros, files, and to execute any other commands. To create new menu, click menu File -> New -> New Menu.

The following information applies to pop-up menus and [toolbars \(23\)](#) (syntax is almost identical). For example, "menu item" in most cases also means "toolbar button".

Menu items

Each line in menu text creates one menu item. Syntax:

```
[label ]:statements[ * icon]
```

label - menu item's text. Optional. Can contain [escape sequences \(137\)](#).

statements - one or more commands. If several commands are used, use semicolon to separate.

icon - icon file.

There are also two simpler forms:

1. Run macro:

```
Macro[ * icon]
```

2. Run file:

```
[label ]"file"[ * icon]
```

Here **Macro** is macro name, which also is menu item's label; **file** is file name (full path or filename) or [ITEMIDLIST string \(246\)](#).

Lines that begin with space or semicolon can be used for comments.

Menu items can be added using dialogs from the floating toolbar. Also, you can drag and drop macros, files and Internet links.

Menu example:

```
My Macro :mac "My Macro"
IE :run "iexplore.exe"
Select All :key Ca * text.ico
Arial italic :spe; men 57696 "WordPad"; key "Arial" TPDY
:mes "Label is optional"
-
comments
My Macro
IE "iexplore.exe"
```

Separators

- horizontal separator (for toolbars, it can be vertical or horizontal).
- digits wide separator (for toolbars only). Digits is separator width.
- | vertical separator (for menus).

Submenus

Submenus are created using > and < characters, as in the example below. Line that begins with > opens submenu. Line containing only < ends the submenu. Submenus in toolbars are not supported. Example:

```
Macro1
>Submenu
  Macro2
  Macro3
<
Macro4
```

Expanding file folders (QM 2.2.0)

In menu Properties you can set to expand file folders, ie automatically create submenus of files. Use the following syntax to create an expandable folder item:

```
Label "folder" ["folder2" ...] [ * icon]
```

Label usually is folder name, although can be any text. Folder can be folder path or [ITEMIDLIST string \(246\)](#). If several folders are specified (max 10), they are merged.

Example menu:

```
/expandfolders
Start "$start menu$" "$common start menu$"
Desktop ":: "
My Computer ":: 14001F50E04FD020EA3A6910A2D808002B30309D"
C:\ "C:\"
```

If the menu contains only single expandable folder item and no other items, the folder is expanded in the main popup menu, otherwise it creates submenu.

In Include/Exclude fields (in Properties), you can specify one or more patterns of files to be included or/and excluded. If both fields are empty, included are all files. Read [more \(28\)](#).

Special characters in menu item labels

To create keyboard shortcut (underlined letter) for a menu item, insert & before the letter. Example: `L&abel`. Use && for simple &. Also, you can select an item with arrow keys and execute with Enter. Use Esc to close the menu or current submenu.

QM 2.2.0. To append hotkey name or some other text aligned at the right, insert tab. It can be literal tab, or several tabs, or [9]. Example: `Label[9]Ctrl+Shift+U`.

How to show a menu from code

To show a menu from code (e.g. from a macro, another menu or toolbar) use [mac \(100\)](#). If used as function (like `variable=mac("menu")`), `mac` waits until the menu is closed. It then returns 1-based index of selected line, or 0. Note that lines not necessary match visible menu items, because menu can have comments, separators, etc. If used like `mac "menu"`, it does not wait.

QM 2.2.0. For expanded folder items, `mac` returns -1. To get last clicked item info, use function [GetLastSelectedItem \(114\)](#) after `mac`. The function also can be used to get info of other items.

QM 2.2.0. In Properties you can set to not run the command/file/macro when a menu item is clicked. The macro for example could get file path using [GetLastSelectedItem \(114\)](#) and do some other action.

QM 2.2.0. While `mac` waits until the menu is closed, it processes messages. In previous versions, `mac` did not process messages, and therefore showing menus from dialogs did not work well.

If menu is launched with a command (e.g., `mac "Menu" "command"`), the command is interpreted as menu item label, and, instead of showing menu, that item is executed.

QM 2.2.0. Menu position can be specified:

```
mac "Menu" "" x y flags
```

Here flags are [TrackPopupMenuEx \(256\)](#) flags, e.g. `TPM_RIGHTALIGN`. To use default position, x and y can be "".

See also: [menu bar triggers \(33\)](#)

Sub-functions

[Sub-functions \(182\)](#) can be used in menus, as well as in QM items of other types. The first `#sub` directive ends menu text. Use sub-functions for menu items that have multiple code lines and you don't want to create separate macros for them.

Example menu:

```
A :out sub.Add(3 1)
B :sub.Sub1
```

```
#sub Add
function# a b
out __FUNCTION__
ret a+b
```

```
#sub Sub1 m
```

With m attribute, this sub-function is used as menu item text, not called as function.

```
out __FUNCTION__
```

Like in all functions, default speed ([spe \(90\)](#)) in sub-functions is 0. In sub-functions with m attribute it is like in macros and menu items, default 100.

The right-click menu (QM 2.2.0)

When you right-click a menu item, pops up another small menu. You can use it to quickly open the menu in QM, open the target macro, or open the folder of the target file. To right-click an item that opens a submenu, at first press Esc to close the submenu.

See also: [menu and toolbar icons \(24\)](#), [menu options \(28\)](#), [hybrid paste \(58\)](#), [DynamicMenu](#), [ShowMenu](#), [ListDialog](#), [ShowDropDownList \(118\)](#).

Toolbar

Toolbars are used to launch macros, files, and to execute any other commands. Toolbar syntax is the same as of [pop-up menu \(22\)](#), except that toolbars cannot have submenus.

A toolbar can be free or attached to a window. If a toolbar is attached to a window, that window becomes owner of the toolbar. The toolbar always is on top of that window. It is hidden when the window is minimized or hidden. It is destroyed when the window is destroyed. It follows the window (optionally) when the window is moved or resized. A toolbar can have multiple instances attached to multiple windows. A window can have several different attached toolbars.

A toolbar can be attached to a window using two ways:

1. Assign a [window trigger \(32\)](#).
2. Launch the toolbar using [mac \(100\)](#) with window handle as command. Example: `mac "Toolbar9" win("Calc")`

If toolbar is launched using some other way, it isn't attached to a window. Such toolbar is always on top.

Macro launched from a toolbar receives toolbar window handle in the `_command` variable. Use `val(_command)` to get it.

When you use [mac \(100\)](#), it returns toolbar window handle (even if the toolbar was already running).

Tips

Move toolbar: drag with the mouse right button.

Move or delete a button: Shift+drag.

Add a button to run macro, file or open Internet link: drag and drop. Ctrl can be used for shortcuts.

If you drag and drop a file onto a program icon, you can choose to open the file in that program.

Right click toolbar to view context menu.

Don't show text of some buttons: insert tab at the beginning of line. You still will see tooltips.

See also: [Quick Start \(2\)](#), [toolbar right-click menu \(25\)](#) [toolbar properties \(26\)](#) [extended toolbars \(27\)](#) [DynamicToolbar GetToolbarOwner \(114\)](#)

Menu and toolbar icons

Use the [Icons \(16\)](#) dialog to find or create icons and assign them to menu items and toolbar buttons.

You can specify icon file after *. Also can be specified 0-based icon index. If icon is not specified, is used default icon. Use ** to prevent adding an icon. To prevent adding icons to all menu items, check 'No Icons' in Properties.

Examples:

```
Macro5 :mac "Macro5" * Shell32.dll * 5
Macro5 :mac "Macro5" * Shell32.dll,5
Select All :key Ca * C:\Icons\Select All.ico
:int i=2*2; paste(i * 10) * iconinmyqmfolder.ico
Macro5 :mac "Macro5" * $desktop$\ic1.ico
Macro5 :mac "Macro5" * "resource:<Macro5>ic1.ico"
Macro5 :mac "Macro5" **
```

QM 2.3.0. Icon index also can be specified using syntax iconfile,iconindex (example 2).

QM 2.4.1. Supports [macro resources \(261\)](#). If resource or file (full path) is enclosed in ", displays the icon in the code editor.

With menus created at run time ([DynamicMenu](#)), instead of icon file can be used icon handle. To retrieve icon handle, use [GetFileIcon \(114\)](#) or [GetWindowIcon \(114\)](#) or Windows API functions. [Example](#). Icon handles can be used with toolbars too. The icons must not be destroyed until the toolbar is closed.

Shows dynamically created menu of windows.

```
ARRAY(int) aw; GetMainWindows aw
ARRAY(__Hicon) ai.create(aw.len)
str s t
int i
for i 0 aw.len
  ai[i]=GetWindowIcon(aw[i])
  t.getwintext(aw[i])
  t.findreplace(" : " " [91]58]") ;;escape :
  t.escape(1) ;;escape " etc
  s+F"{t} :act {aw[i]}; err * {ai[i]}[]"
  out s
i=DynamicMenu(s "" 1)
if(i) outw aw[i-1]
```

Toolbar right-click menu

This popup menu is shown when you right-click a custom [toolbar \(23\)](#). It allows you to set some options for the toolbar. [More options \(26\)](#) can be set in the Properties dialog.

Edit toolbar - show QM window and open the toolbar.

Edit macro - if the button is used to run a macro, show QM window and open the macro.

Open file location - open folder that contains button's program/file.

Options

Show text - show text labels.

Sizing border - show border. If not checked, you cannot resize the toolbar.

1-pixel border (QM 2.2.0) - show 1-pixel border. The border can be used to resize the toolbar if *Sizing Border* is checked.

Auto shrink - different toolbar size when the mouse pointer is inside and outside the toolbar.

Follow owner window - follow the owner window when it moved or resized.

Follow screen size (free toolbars) - move the toolbar if need when screen size (resolution) changed.

Activate owner window on click - clicking the toolbar activates the owner window.

Tooltips always - show tooltips even when *Show text* is checked.

Vertical toolbar - check this if you use the toolbar as vertical toolbar. It replaces vertical separators with horizontal separators, etc. Note that it does not change toolbar's width and height. To do it, drag toolbar's borders.

Equal-size buttons - equal-size buttons when *Show Text* is checked.

3D buttons (QM 2.2.1) - display old-style 3D buttons. Incompatible with hot item colors. On Vista and later, incompatible with custom font.

Open file on drop (QM 2.3.0) - when a file dropped, don't show menu. If the button runs a program, opens the file in the program. To show menu, drag and drop with the right mouse button.

Hide if full-screen window (QM 2.3.3) - hide the toolbar when a full-screen window is active. If it is a window-attached toolbar, hides when its owner window is full-screen. Else hides when a full-screen window is active in the same monitor.

Other programs cannot hide - don't allow other programs to hide the toolbar or move it somewhere. This option is used only with free toolbars (not window-attached toolbars). For example, a program that creates multiple desktops will not hide the toolbar on inactive desktops. Note that free QM toolbars are always visible on all Windows 10 virtual desktops, regardless of this option.

On top of topmost windows (QM 2.3.3) - periodically set the toolbar on top of always-on-top windows (in the Z order). If not checked, the toolbar can be behind another always-on-top window (e.g. taskbar). Alternatively you can make a toolbar visible by triggering it again. This option is used only with free toolbars (not window-attached toolbars).

Quick icons - load icons quickly. When closing the toolbar, icons are saved to a single file (cache), and, when opening the toolbar next time, icons are quickly loaded from that file. This is useful with toolbars that have large number of icons and therefore are opened slowly. If unchecked, icons everytime are extracted from associated files. It is slow process (especially if an antivirus program scans each loaded executable module) and requires more memory.

Refresh icons - refresh outdated icons. Useful when *Quick icons* is checked.

Coordinates

If checked *Follow owner* or *Follow screen size*, toolbar coordinates are relative to the selected corner of the owner window or screen. If checked *Auto shrink*, the expanding direction depends on the selected corner.

Auto select - auto-select the nearest corner when moving the toolbar with the mouse right button.

Notes

QM 2.4.0. These settings, also toolbar position and size, are saved in the main [QM file \(17\)](#) when closing the toolbar. Toolbars that are in currently loaded shared files also save in the main file. Older QM versions used the registry.

If you have lost a toolbar (it is running but invisible), check View Active Items in the Run menu, right click a toolbar in the Running Items list, and click Move Here. If you click Reset, it closes the toolbar and resets its size, position and right-click menu settings to the default values.

Toolbar properties

[Toolbars \(23\)](#) have two sets of properties - dynamic and static.

Dynamic properties can be changed while the toolbar is running. Position - move the toolbar (right-drag). Size - resize the toolbar (drag an edge). Some other properties can be changed in the [right-click menu \(25\)](#). Dynamic properties are automatically saved in current main [QM file \(17\)](#).

Static properties can be changed in the Properties dialog. They are saved in toolbar text, first line, and also can be edited directly there. They are described below. If toolbar position, size or right-click menu settings are specified, the specified static properties are used instead of the automatically saved dynamic properties.

Syntax

```
/option1 [parameters] [/option2 [parameters] ...]
```

Space or semicolon, and one or more options. Each option begins with / and can have 1-2 parameters. If some parameter is not used, it can be omitted or "". Numeric parameters can be simple integer values, named constants, global variables, functions, {code} and expressions with operators. Expression with spaces or / must be enclosed in parentheses.

Example (first line of toolbar text):

```
/col 0x80C0FF /hcol "" 0xFF /style WS_CAPTION|WS_SYSMENU|WS_MINIMIZEBOX 0
```

A numeric parameter can be or contain code enclosed in { }. The code is executed as function, when creating the toolbar, in main QM thread. The code should return a value ([ret](#)). In code and user-defined functions, toolbar owner window handle is `val(_command)`.

Examples:

```
/mov 200 {ret GetClientTop(val(_command))-20}  
/mov 200 TB_ClientY-20
```

Use `mov0`, `siz0`, `ssiz0` and `set0` to apply these options only first time and after Reset.

Available options

1. Set toolbar position.

```
/mov x y
```

Here **x** and **y** are toolbar position, in pixels. If the parameters are (or begin with) `xm` and `ym`, toolbar will be under the mouse pointer. Coordinates are relative to the owner window or screen top-left or other corner (depends on the Coordinates setting; see also [/client](#)). Use `mov0` to apply only first time.

Examples:

```
/mov 200 0  
/mov xm ym+100  
/mov 200 -23 /client
```

2. Set toolbar dimensions.

```
/siz width height
```

Use `siz0` to apply only first time.

3. Set toolbar dimensions for shrinked state.

```
/ssiz width height
```

Use `ssiz0` to apply only first time.

4. Set icon size.

```
/isiz width height
```

Default size is 16x16. If **cy** is omitted, it will be equal to **cx**. If **cx** is more than **cy**, text is placed below icon.

5. Set maximum and/or minimum button size.

```
/bsiz maxwidth minwidth
```

QM 2.2.1: Works with all styles. Added minwidth.

6. Set initial settings of the [right-click menu \(25\)](#). In the Properties dialog it is "Style flags".

```
/set flags mask
```

flags (247):

1	Follow owner window when it is moved or resized.
2	Show button text.
4	Activate owner window on click.
8	Auto shrink.
16	Equal-width buttons.
32	No sizing border.
64	Show tooltips always, even if button text is shown.
128	Use as vertical toolbar. Line breaks will be horizontal.
0x000	Coordinates: top-left.
0x100	Coordinates: top-right.
0x200	Coordinates: bottom-left.
0x300	Coordinates: bottom-right.
0x400	Coordinates: auto select.
0x1000	Show in all virtual desktops.
0x2000	Quick icons. Uses an icon cache.
0x4000 (QM 2.2.0)	1-pixel border.
0x8000 (QM 2.2.1)	3D buttons.
0x10000 (QM 2.3.1)	When a file dropped, open it with the button program. Don't show a menu.
0x20000 (QM 2.3.3)	Hide if a full-screen window is active.
0x40000 (QM 2.3.3)	Super on-top. Make on top of other always-on-top windows, eg taskbar.

mask - use to change only some flags. Will be changed only those bits of **flags** that are 1 in **mask**. First time are applied all flags regardless of **mask**.

Example:

```
/set 1|2|8|0x400
```

Use `set0` to apply only first time.

7. Set toolbar window style and extended style.

```
/style st exst
```

Here **st** and **exst** are style and extended style of the toolbar window. Can be used style constant names, like in the example. Reference - [MSDN \(256\)](#).

Example:

```
/style (WS_CAPTION | WS_SYSMENU)
```

8. Set background and text color.

```
/col color textcolor
```

Here **color** and **textcolor** are [colors \(240\)](#) in 0xBBGGRR format. Value 1 to 31 will set one of system colors.

9. Set hot item background and text color.

```
/hcol color textcolor
```

If you set hot item background color, Windows themes (visual styles) are not applied, and the toolbar may look differently.

Does not work when 3D buttons.

10. Set transparency.

```
/transp opacity color
```

opacity - value between 0 (makes the toolbar completely transparent) and 255 (completely opaque).

color - sets the specified color completely transparent. Completely transparent areas are transparent to the mouse too. Optional.

On Vista and later, hot button may look distorted. It is because its color is gradient, and some lines match the transparent color and therefore are transparent. Try to change background color with /col.

11. Set background image. Can be bmp, gif, jpg. QM 2.3.4: also can be png.

```
/bmp "file"
```

Example:

```
/bmp "$my qm$\images b1.bmp"
```

12. Glass. QM 2.2.0.

```
/glass
```

On Windows Vista and 7, if Windows Aero theme is enabled, the toolbar will be translucent, like title bars of windows. Incompatible with /bmp, transparent color and hot color (these are ignored if /glass is used).

13. Set font. QM 2.2.0.

```
/font XXXX
```

Font can be set in Properties.

On Vista and later, does not work when 3D buttons.

14. Lock toolbar.

```
/lock flags
```

flags (247):

1	don't show the right-click menu.
2	disable moving.
4	disable resizing.
8	disable modifying. This disables drag&drop features, and three first items in the right-click menu. Even if this option is not used, a toolbar cannot be modified if it is encrypted or in a read-only folder or file.
16	disable the Close item in the right-click menu.
128	lock only if the toolbar is in a shared file ([s] folder).

If **flags** is not used, are applied all flags except 128.

15. Toolbar position specified in mov/mov0 is relative to the client area of the owner window, or to the work area if no owner. QM 2.4.3.

```
/client
```

16. Set [hook function \(27\)](#).

Extended toolbars

The following toolbar option sets hook procedure:

```
/hook functionname
```

The specified user-defined function (hook procedure) receives all messages that are sent to the toolbar's main window (not owner), starting from WM_CREATE. This allows extending QM toolbars. You, for example, can create more controls.

A template is available in menu -> File -> New -> Templates.

Toolbar hook procedure must begin with `function# hWnd message wParam lParam`.

If the hook procedure returns a nonzero value, the message is not further processed in the default window procedure. Some messages (see below) also are not processed in the default toolbar window procedure. To return 0 without further processing, declare the function as long (`function% ...`), and return 0x100000000.

To change toolbar behavior, return a nonzero value on these messages:

WM_SIZE - disable resizing the child toolbar control.

WM_EXITSIZEMOVE - disable deactivating after resize/move.

WM_MOUSEACTIVATE - disable deactivating on click.

WM_COMMAND - don't run the macro/file/command associated with the button.

Other messages that will not be processed: WM_GETMINMAXINFO, WM_NCHITTEST, WM_WINDOWPOSCHANGING, WM_SYSCOMMAND, WM_ERASEBKGD.

If the hook procedure executes `end`, it is no longer called.

QM 2.2.0. WM_INITDIALOG is sent when the toolbar is fully initialized, before showing. wParam is child toolbar control handle, lParam is 0, should return 0. In previous versions, WM_INITDIALOG was not sent.

QM 2.2.0.11. Toolbar controls are created without WS_EX_NOPARENTNOTIFY style, and therefore toolbar hook procedure receives WM_PARENTNOTIFY messages. For example, it can set font when the toolbar control is created (buttons are still not added at that time).

Toolbar hook procedure runs in QM main thread. Be careful when creating it, because code that runs in QM thread can easily make QM unstable. Don't use wait commands. Don't use `atend`. Avoid [thread variables \(49\)](#). To have variables with current toolbar scope, use `SetProp`, `GetProp` and `RemoveProp` functions. Example:

```
/MyToolbar
function# hWnd message wParam lParam

type MYTOOLBARVARS a str'b ARRAY (POINT) c ;;declare type to hold multiple variables (example)
MYTOOLBARVARS* v=+GetProp(hWnd "v") ;;retrieve (required)

sel message
  _case WM_INITDIALOG
    v._new; SetProp(hWnd "v" v) ;;create (required)
    v.a=1; out v.a ;;use (example)
  _
  _case WM_DESTROY
    v.a=3; out v.a ;;use (example)
    v._delete; RemoveProp(hWnd "v") ;;destroy (required)
  _
  _case WM_SETCURSOR
    v.a=2; out v.a ;;use (example)
```

There are several sample toolbars in the [forum](#).

Menu properties

You can set [menu \(22\)](#) properties in the Properties dialog. They are placed in the first line of menu text, so you can also edit them directly.

Syntax

```
/option1 [parameters] [/option2 [parameters] ...]
```

Space or semicolon, and one or more options. Each option begins with / and can have 1-2 parameters. If some parameter is not used, it can be omitted or "".

Some options are explained more in the [menu \(22\)](#) topic.

Available options

1. Set menu position.

```
/pos position
```

position - one of the following words: *mouse* (default), *text*, *center*.

2. Don't show icons.

```
/noicons
```

3. When an item is clicked, close the menu but don't run the associated command. Can be useful if the menu will be launched from code ([mac](#)).

```
/dontrun
```

4. Disable Edit items in the right-click menu.

```
/noedit
```

5. Expand file folders. [Read more \(22\)](#).

```
/expandfolders flags level
```

flags (247):

1	Don't include shell objects that are not file system files/folders. For example, Control Panel folder and most objects in it are not file system objects.
2	Don't include folders.
4	Show hidden files/folders.
8	Expand not only folders but also shortcuts to folders. Expanded are only shortcuts that are in expanded folders.
16	Include only folders. Incompatible with flag 2.
32	Add 'This Folder' item at the top of a submenu. It opens the folder from which is created the submenu.
128	Display paths in the output when the menu is shown. Temporarily select it when you need to know paths of menu items. Especially useful when you need to include or exclude a non-file-system object (e.g. Recycle Bin). While this flag is set, /folder is not applied.

level limits the number of submenus. Examples: 1 - folder; 2 - folder\subfolder; 3 - folder\subfolder\subfolder.

6. Include/exclude patterns for /expandfolders. Applied only to files.

```
/filter include exclude
```

You can specify one or more files to be included or/and excluded. Use | as separator. Use [wildcard characters \(271\)](#).

If include/exclude contains \ or : characters, evaluated is full path, else only filename.

Non-file-system objects are evaluated by path, which usually is one or more class id strings. To see paths, temporarily select "Show paths" in Properties (it is flag 128).

Examples of **include** or **exclude**: "*.txt" (txt files), "*.exe|*.com|*.bat" (files of these 3 types), "jun*" (files that begin with "jun"), "C:*" (don't include files in C:\, but include files in folders), "?:*" (the same for all drives).

7. Include/exclude patterns for /expandfolders. The same as above, but applied only to folders and drives.

28. Menu properties

`/filterfolders include exclude`

Examples of **include** or **exclude**: "C:\\" (drive C:), "A:\|D:\\" (drives A: and D:), "::{645FF040-5081-101B-9F08-00AA002F954E}" (Recycle Bin).

8. Set background image for menu items. Can be bmp, gif, jpg. QM 2.3.4: also can be png. The image is tiled, and therefore can be smaller than a menu item.

`/bmp file`

9. Set background image for highlighted menu item. The image is tiled.

`/bmpsel file`

10. Set text color for normal and selected items.

`/tcol color colorsel`

Here **color** and **colorsel** are colors (240) in 0xBBGGRR format.

11. QM 2.3.0. When menu items exceed screen height, use multiple columns.

`/multicolumn`

Expanded file folders are always displayed in multiple columns regardless of this option.

Autotext list

Autotext lists are used to execute commands when you type certain text. Typical usage - text replacement/autocomplete/autocorrect. Also known as text-sensitive menu or TS menu.

An autotext list has no visible interface. When you type text somewhere, and the text matches one of list items, executes its command.

Triggers

An autotext list works only if it has a trigger.

Text trigger

Added in QM 2.3.3.

Assume you have this autotext list with 2 items (the "/b/i/c/p3" sets options):

```
/b/i/c/p3
sun : ' "Sunday"
mon : ' "Monday"
```

When you type "sun" and a delimiter character (for example space), it replaces the text with "Sunday " (erases "sun " and types "Sunday "). Or you can press Ctrl instead of a delimiter character. When you type "mon ", replaces with "Monday ".

This trigger does not require an event (hotkey etc) before typing text. It just makes the autotext list active. Other QM item types cannot have this trigger.

To assign this trigger, use the [Properties \(10\)](#) dialog. You can set programs where the autotext list works, and a filter function ([read more below](#)). Also you can set prefix text; for example, if prefix text is ##, for the above example you would have to type "##sun " or "##mon ".

Other triggers

Use if you need a hotkey or other trigger before typing text.

Assume you have this autotext list with 2 items (the "/b/i/c" sets options):

```
/b/i/c
sun : ' "Sunday"
mon : ' "Monday"
```

Assume its trigger is key F12. When you press F12 and type "sun", it replaces the text with "Sunday". When you press F12 and type "mon", replaces with "Monday".

Assume its trigger is key `, and "Eat" is unchecked. When you type "` sun", it replaces the text with "Sunday". When you type "`mon", replaces with "Monday".

You should not use alphanumeric keys for trigger. Instead use text trigger. If using alpha keys and option /i (case insensitive), also set third state for Shift checkbox in Trigger tab.

Notes

QM 2.3.3. An autotext list cannot work if it doesn't have a trigger. If you want to use [mac](#) or other way to activate it, at first assign a trigger. It can be any trigger except text trigger. Also, an autotext list will not work if it or QM is disabled.

QM 2.3.3. All autotext lists don't work if disabled in Options -> Triggers, regardless of trigger type.

List items

Autotext list text is a list of items, each in separate line. Item format:

```
text : statements
```

text - text that you must type to execute **statements**.

- Can be used any characters, including spaces, tabs and line breaks.

- Can be used [escape sequences \(137\)](#).

statements - any commands that you can use in macros.

- To separate multiple statements, use semicolons.

Separator between **text** and **statements** is single space and colon (`:`). For example, if there is line ``mail : "name@abc.com"` (two spaces after "mail"), the command is executed when you type "mail " (with space).

Lines that don't have `:` and don't begin with `/` are ignored and can be used for comments.

QM 2.3.3. Lines that begin with space are comments. If need space, use escape sequence [32].

Options

Lines that begin with `/` are used to set options for following items (until next such line). To set options, you can use the Properties dialog. Options:

/s	Select the typed text. Uses Shift+Left keys.
/b	Erase the typed text. Uses Backspace key. <ul style="list-style-type: none"> • You can use either /s or /b, not both.
/i	Case insensitive.
/c	If first character of the typed text is uppercase, capitalize first character of the first paste (58) or key (56) . <ul style="list-style-type: none"> • It works with <code>key "text"</code> and <code>key (variable)</code>, but not with <code>key text</code>. Not with <code>str.setsel</code>. • QM 2.3.3. It works in whole thread. previously didn't work in functions called from the autotext list. • QM 2.3.3. If all typed text is uppercase, capitalizes all text of the first paste or key command. • QM 2.3.3. This also adds "case insensitive" option. • /s or /b also must be set. If not, this option works only if all typed text is uppercase.
/m	QM 2.3.3. With confirmation. Shows a popup list (read below) containing single item.
/p1	QM 2.3.3. Need a delimiter character (postfix) to execute an item. <ul style="list-style-type: none"> • For example, item "test" will be executed when you type "test ", or "test," or "test.", etc. • Delimiter characters are all characters except alphanumeric and those specified in Options -> Triggers.
/p2	QM 2.3.3. Need to press Ctrl to execute an item.
/p3	QM 2.3.3. Need a delimiter or Ctrl to execute an item.

This example sets "select text", "case insensitive" and "capitalize" options for items that follow:

```
/s/i/c
```

You can also add and remove options for each item separately. To remove, use uppercase characters. In this example, `/C` tells that "capitalize" option must not be used:

```
/C/ pw1 : "password1"
```

If you need `/` character at the beginning of item's text, use `//`.

More options

It is possible to add some more options with a [filter function \(40\)](#). Filter functions allow you to define conditions where the autotext list (all or some items) will work. You can define a window, define a custom set of postfix characters, etc.

To get user-typed text and other info, you can use function [TriggerInfoAutotext](#) (QM 2.3.5) or [TriggerInfoTsMenu](#) (QM 2.3.3). The function can be used in autotext list filter functions and item code.

QM 2.3.5. You can use [FFT_Autotext_PostfixCustom](#) as a filter function example or template for autotext lists.

Multiple matching items

QM 2.3.3. If multiple items match typed text, shows a popup list.

The displayed popup list item text is extracted from list item code or comments. Example:

```
text : '"This will be popup list item text"
text : '"zzzz" ;;This will be popup list item text
```

Keyboard shortcuts:

- Select item: Enter, Tab, 1-9.
- Cancel: Esc.
- Other: Down/Up arrow, Home, End, Page Down/Up, F1.

To show this for a single item, use option /m.

Notes

To prevent executing an autotext list item, while typing its text (not before) do one of the following:

- Click the mouse in the same window.
- Press Ctrl or some other key that does not type text. Except Shift, Alt, Win, Backspace.

An autotext list item will not be executed if previously typed character is non-delimiter (alphanumeric + specified in Options -> Triggers), unless trigger is a key with "Eat". To execute anyway, before typing do one of the above (click, Ctrl, etc). QM 2.3.5: previously typed character can be any if item text begins with a delimiter and trigger is autotext.

You can associate sounds with autotext list events: Options -> Sounds.

QM 2.3.3. Works in console windows.

QM 2.3.3. Removed 5 s timeout.

QM 2.3.3. Default [opt keymark \(97\)](#) is 1. It makes more reliable. In called functions it is 0.

Usually in autotext list items you use an "insert text" command. To insert text, can be used keyboard ([key \(56\)](#)) or clipboard ([paste \(58\)](#)). For short text it's better to use keyboard, because it does not erase clipboard, however it can be too slow with quite long text. For items where code is simple text in double quotes (like `abc : "text"`), QM uses clipboard or keyboard, depending on [hybrid paste \(58\)](#) setting in Options -> General.

May interfere with other running keyboard/autotext software, or with similar features of the target program, eg Microsoft Word autocorrect.

Triggers: keyboard (hotkey)

Launches the macro when you press the specified hotkey. To assign this trigger, use the [Properties \(10\)](#) dialog or the [Trigger field \(264\)](#).

If the Shift button in the Properties dialog is in indeterminate state, the trigger will work with Shift pressed or released.

If "Eat" is not checked, the trigger keys are passed to the window that would normally receive them. Triggers with a next key always eat the first key if there are macros that have the same first key and "Eat" checked.

By default, QM does not wait until you release trigger keys, but releases them itself. In some cases it can cause unexpected effects. For example, if trigger contains Ctrl and Alt, and macro sends Delete, can be generated Ctrl+Alt+Delete. If "When released" is checked, macro starts only when all keys and mouse buttons are released.

Two non-disabled macros should not have same hotkey trigger, unless both are specific to a program or some other condition (see [filter functions \(40\)](#)). On conflict, will run the macro that was above in the list of macros when QM was started or current file opened, or that is older.

Some hotkeys are used by Windows, other programs, shortcuts. For example, Alt+Tab, Win+R, Win+Shift+M. If you want to use these hotkeys, check "Use low-level keyboard hook" in [Options \(13\)](#). Low level hooks also make triggers to work in console windows in Windows Vista. However, low-level keyboard hooks stop working if some macro uses DirectInput or raw input (like Keyboard Detector) to get keyboard input.

Hotkey triggers don't work if there is no active window. It is rare situation, because some window (e.g. desktop or taskbar) usually is active.

Other type of hotkey - hotkey for shortcut (shortcut itself can be on the desktop or in the Start menu). Such hotkeys work even when QM is not running, also they always work in console windows. To create a shortcut with a hotkey, press the Shortcut button in the Properties dialog, click in the Hotkey field, press the hotkey, and click OK.

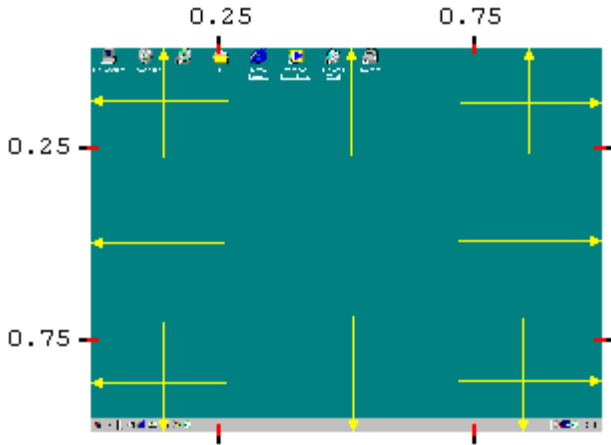
See also: [autotext \(29\)](#)

Triggers: mouse

Launches the macro on mouse click, certain movements, etc. To assign this trigger, use the [Properties \(10\)](#) dialog.

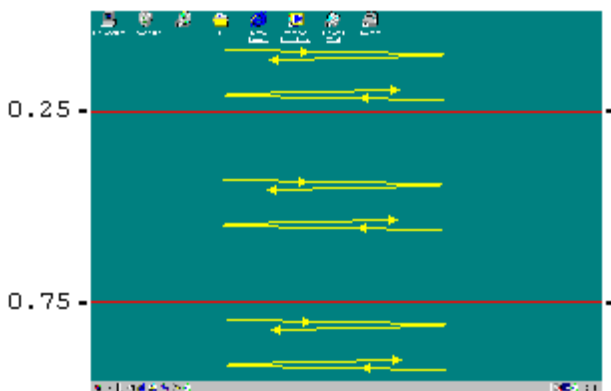
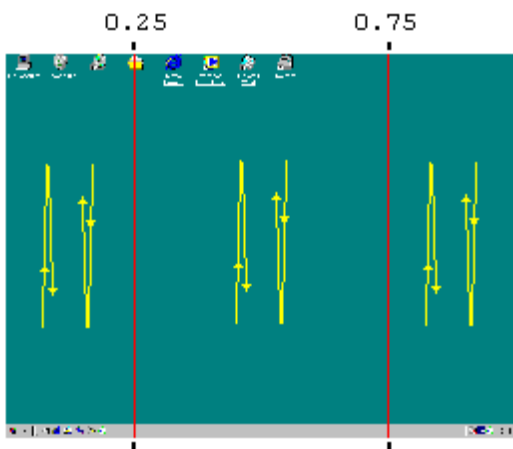
Mouse trigger types




1. Fast mouse movement (or double movement) to the screen edge. There are 12 active areas:



You can set required mouse speed in [Options \(13\)](#). If "Slow" is checked, trigger works even when you touch screen edge without speed.

2. Fast mouse movements (or double movements) in one of 3 vertical areas (move up-down or down-up) and 3 horizontal areas (move toleft-toright or toright-toleft):



Mouse movements must be sharp (like , but not like  or ) , not too slow and not too fast (about 0.2 s). You can set required amplitude in [Options \(13\)](#).

3. Mouse left, right, middle, X1 or X2 button click or double click.

You can make a trigger specific to certain parts of window (caption, minimize button, etc). To make it specific to controls or other conditions, use [filter functions \(40\)](#).

If "Eat" is checked, mouse click messages are not passed to the application.

4. Mouse wheel rotation forward or backward.

Notes

Mouse triggers can be used with modifier keys Ctrl, Shift and Alt.

Be careful when unchecking "When released". If it is unchecked, the macro starts while modifier keys and mouse button are still pressed. QM cannot reliably "eat" them. If the macro sends keys ([key](#), [paste/otp](#), [str.setsel](#), [str.getsel](#)) or mouse buttons, they may be modified by trigger's modifier keys.

Two non-disabled macros should not have same mouse trigger, unless both are specific to a program, monitor, hit-code or other condition (see [filter functions \(40\)](#)). On conflict, will run the macro that was above in the list of macros when QM was started or current file opened, or that is older.

If mouse triggers don't work in some windows (e.g. console windows in Windows Vista), or in some parts of some windows, check "Use low-level mouse hook" in [Options \(13\)](#). Also, when using low-level hook, mouse movement triggers work better when the program that receives mouse input is busy.

QM 2.2.0: Hit-test code sometimes can be different than in previous versions, although in most cases the same.

Triggers: window

Launches the macro when the specified window appears, disappears, is activated, etc.

To assign this trigger, use the Properties dialog. You can use the Drag tool to fill window name/class fields. While dragging, you can right-click to send window to bottom, or Ctrl+Shift+right-click to send child window to bottom.

You should always specify window class. If it isn't specified, QM ignores some windows, e.g. windows without caption.

You can also specify text/class of a control (child window) of the window. Accessible objects (e.g., objects in web pages) are not supported.

You can also specify [window style \(275\)](#) - numeric value, or p (popup), or m (not popup).

Window name and control text can be partial. Must match case. If empty, matches any.

If "Use *" is checked, window name and control text must be full or with [wildcard characters \(196\)](#) * and ?. For example, to tell that window name must end with " - Notepad", check "Use *" and enter "* - Notepad". To tell that window name must be exactly "Notepad", check "Use *" and enter "Notepad". String "" matches windows with no name.

In the window name field, you can use [regular expression \(198\)](#). First character must be \$. For example, \$Word.*\.(?i)rtf\$ matches window name that contains "Word" and ends with ".rtf" or ".RTF" (the first \$ tells that you use regular expression, .* means any number of any characters, \. means literal ., (?i) sets "case insensitive" option, the last \$ means end of window name). Regular expression can be used to identify more than one window. Example: \$(Notepad|Calc). When using regular expression, "Use *" is not applied to window name.

Class names must be full. Can contain wildcard characters, regardless of the "Use *" option (QM 2.3.4). If empty, matches any.

To launch macro when window is created (appears first time), you should select "Created & active". If window is shown but not activated, select "Created & visible". To launch macro each time when window is activated, select "Activated". You can use "Activated" with toolbars (this does not create more instances of the same toolbar on the same window). To launch macro each time when window name is changed and matches the specified name, select "Name changed".

If "When idle" is checked, the macro starts only when the window is not busy and is ready to receive user input. However, the macro may not run with some windows, for example if the window is busy (uses CPU) all the time.

Several items can have same "Window" trigger. For example, a window can have several toolbars.

A window-triggered macro receives that window handle through `_command` variable. To get it, you can use `val(_command)` or [TriggerWindow](#).

The "destroyed" trigger is unreliable with 64-bit windows. Also if checked "Other" in Options -> Triggers -> Low level hooks. Instead use a function that gets window handle and waits until it is destroyed. Example:

```
trigger: window created
int hwnd=TriggerWindow
wait 0 -WC hwnd
out "window destroyed"
```

See also: [Properties \(10\)](#)

Triggers: QM events

Launches the macro on certain Quick Macros events. To assign this trigger, use the Properties dialog.

Startup

These events occur when opening [QM file \(17\)](#), for example when QM started. If "Run synchronously" is checked, QM waits until the function ends, therefore it should be fast to avoid QM startup delay. The `_command` variable is "1" on file change, "2" on QM startup, "3" on Windows startup (QM started with command line parameter S). These events can occur in this sequence: file loaded, QM started, Windows started.

If exists a user-defined function named `init2`, it is called when opening QM file, before any other functions/macros, after initializing QM extensions. Don't need a trigger for it.

Exit

These events occur when closing QM file, for example before QM exits. The function runs synchronously and should be fast. The `_command` variable is "1" on file change, "2" on QM exit, "3" on Windows shutdown/restart/logoff. These events can occur in this sequence: Windows exits, QM exits, file unloading.

Tray icon

These events occur when you click the QM tray icon. These triggers disable default QM actions (show QM on left click, exit QM on Ctrl+left click, show menu on right click, disable/enable on middle click).

Trigger management

Used by [user-defined triggers \(39\)](#).

Add to a menu

QM items that have this trigger are added to the QM menu bar, to the New menu, or to the tray menu.

QM 2.2.0. Menu bar triggers launch the macro with two arguments, which are x and y coordinates where [popup menu \(22\)](#) should be displayed.

QM 2.2.0. For macros and menus that have menu bar trigger, you can include & character in macro name. Then next character in menu bar will be underlined, and you can use the keyboard to show the menu.

File link run (QM 2.3.0)

Run when a QM item of type [file link \(19\)](#) is launched. To run only for certain file types or files, specify a filename pattern. Can be used wildcard characters * (matches 0 or more characters) and ? (matches any single character). When you assign the trigger using the Properties dialog, it inserts code that provides file link item [id \(107\)](#) and its file path.

End thread (QM 2.3.0)

Run when the user tries to end a thread in the Threads dialog or in the Running Items list. If the function returns 0, QM does not end the thread. The function, for example, can display a warning message box, or just silently deny the user's request, or correctly end the thread in a special way.

QM show/hide (QM 2.3.3)

QM show: Run when QM window is shown. For example, on tray icon click.

QM 2.4.1. Added option 'Run synchronously'. The function runs before showing QM window. If it returns not 0, QM window remains hidden. [Example with password](#).

```
if(!inpp("p" "Password to show window" "Quick Macros")) ret 1
```

QM hide: Run when QM window is hidden. For example, on X button click. Not on QM exit.

Recording ended (QM 2.3.3)

Run when recording ends, except on Cancel. Must be function. Receives recorded macro string, and can change it.

Triggers: file

Launches the macro when a file in the specified folder is added, removed, renamed or modified. To assign this trigger, use the Properties dialog.

QM watches for changes in the specified folder. The folder can be local or network folder or drive.

By default, QM watches for changes on all files within the folder. You can use "Include files" and/or "Exclude files" fields to filter files you need. You can specify one or more filenames or filename patterns with [wildcard characters * and ? \(271\)](#). To specify several filenames or patterns, use | separator. For example, use "test.txt" to watch for changes on file "test.txt"; use "*.txt|.ini" to watch for changes on all text and ini files. Filenames can optionally include path relative to the watched folder, e.g. "subfolder/test.txt". A filename is compared as containing relative path if it contains \.

Sometimes Include/Exclude fields may not work as expected because QM may receive short (DOS) filename (e.g. "C:\PROGRA~1\SOMEFI~1.TXT" instead of "C:\Program Files\Some File.txt"). This depends on what filenames uses the program that works with the file. Before evaluating include/exclude fields, QM converts short filename to long filename on Added and Modified events (QM 2.2.0). It is impossible to convert on Remove and Name Changed events because the file already does not exist. To make the trigger independent from this, in Include/Exclude fields use only extension (e.g. "*.txt"), or part of filename (e.g. "some*.txt"), or both long and short versions ("some file.txt|somefi~1.txt").

In the list box, select what changes will cause the macro to run:

Added - a file is created, moved or copied into the watched folder.

Removed - a file is deleted or moved out.

Name changed - a file is renamed.

Modified - any changes on file times, attributes, size and security attributes. For example, if file content changes, the modified time is changed. Note: the last access time precision is very low, and therefore the macro will not be launched each time if you only select Time Accessed.

QM automatically starts/stops watching network and removable drives when they become available/unavailable.

Duplicate "Modified" events are joined to single event.

The macro can receive trigger information through the [function \(152\)](#) statement:

```
\
function event $name [$newname]
```

event - 1 on added, 2 on removed, 4 on name changed, 8 on modified.

name - file name, including path.

newname - on name changed, this is new name, including path. On other events, **newname** is empty.

Triggers: event log

Launches the macro when an event is added to the Windows Event Log. To assign this trigger, use the Properties dialog.

Windows and applications record important events to the Windows Event Log. You can see these events in the Event Viewer. For example, Windows logs when an application is shut down due to an error, when an Internet connection is connected/disconnected, when services are started/stopped/paused, various other errors and information.

There are several logs - System, Application, Security, and possibly more. Event source typically is the application or service that logs the event. Event id is a unique number (within that source) that identifies the event. Message is a text information. Message typically is composed of constant text that is associated with that event id, and inserted strings that are specific to a particular event. If "Retrieve full message" is unchecked, is retrieved only these inserted strings, separated by spaces. This is much faster, because to retrieve whole message QM have to load the dll that contains the text. Since you can specify event id, you don't have to retrieve whole message to identify the event. In the Message field can be used wildcard characters * and ?.

The macro can receive trigger information through the [function \(152\)](#) statement that is inserted with your confirmation when you close the Properties dialog.

Event log triggers don't work on user accounts with limited rights because access to event logs is denied.

Triggers: process

Launches the macro when the specified process (program) starts or ends. To assign this trigger, use the Properties dialog. Unavailable in [portable QM \(259\)](#).

If "Already running" is selected, launches the macro if the process is already running when QM starts or a process trigger is added, removed or modified.

The macro can receive trigger information through the [function \(152\)](#) statement that is inserted with your confirmation when you close the Properties dialog.

Triggers: accessible object

Launches the macro when the specified [accessible object \(84\)](#) generates an event. To assign this trigger, use the [Properties \(10\)](#) dialog.

Accessible objects are various user interface objects in a window, such as buttons, text fields, list items, links, menus. This trigger type also supports objects that are not window parts: top-level windows, caret (text cursor), cursor (mouse pointer), sound and alert. There are numerous events that can be used to launch the macro. Examples:

- When an object is created or destroyed.
- When an object receives focus.
- When an object's state or location changes.
- When any of an object's properties change.

To discover events and object properties, use accessible object event logging. Right click the output pane and check Log -> Acc. events in the menu. In the logging options dialog you can specify certain filters, to exclude some types of events and objects.

For each event, in the output is displayed indexed string that has this format:

- i. event, idObject, idChild
 - ow: object window class, id and text
 - pw: top-level parent window class and name
 - ao: accessible object properties

Properties that cannot be retrieved are not displayed. For example, if the object window is [top-level window \(275\)](#), it does not have parent window, and pw line is not displayed. In the Properties dialog, leave corresponding fields empty.

In the Properties dialog, select an event in the list at left, and enter other properties in the right. All properties except idObject are optional, but should be specified at least object window class (or/and id) and parent window class (and name, if class isn't unique). Accessible object properties and child window text are slower to retrieve, and therefore should be used only if really necessary. Note that idChild is not child window id, but is used to identify the accessible object. If it is some nonzero value, consider checking "Any nonzero", because often it is nonconstant.

Class names must be full or with wildcard characters (QM 2.3.4). Other strings can be partial. If Use * is checked, must be full or with [wildcard characters \(196\)](#). Empty fields are not evaluated. To match empty string, use single * character in that field.

There are quite many fields in the Properties dialog. It is easy to make a mistake when filling them. If the trigger does not work (macro does not run) when expected, temporarily check Debug and click OK. Then try to reproduce the conditions that should cause the macro to run. In the output pane will be displayed the first property that doesn't match when an event of that type occurs and these properties match: idObject, idChild, ow class and pw class.

The macro can receive trigger information through the [function](#) statement that is inserted with your confirmation when you close the Properties dialog. **hwnd** is object window handle. If it is a child window, you can use `int h=GetAncestor(hwnd, 2)` to get its top-level parent window handle. To get the accessible object, you can use ObjectFromEvent function of [Acc](#) class.

Before logging, save all your work in all programs, because some events of some objects may crash the program to which the object belongs. When an exception occurs while logging, QM tries to handle it so that the program would not crash, but it is not always possible. Particularly, exceptions occur when QM tries to get accessible object properties. This is another reason why you should not use accessible object properties in triggers where the object can be identified not using them. To make logging safe, check "don't get ao properties".

Events "destroy" and "hide" are unreliable with 64-bit windows.

This trigger type is based on Microsoft Active Accessibility and WinEvents. You can read more in [MSDN library \(256\)](#) (search for WinEvents). To read about event types, search MSDN for "Event Constants".

Example

Let's make trigger "any web page in Internet Explorer is opened and fully loaded". Open Internet Explorer (IE). Then switch to QM, right click the output pane and select Log -> Acc. events in the popup menu. In the dialog, click OK. Switch to IE and open a page, for example, Google page. Then switch to QM and stop logging (click the same menu item). Try to find an event that could be used for this trigger. Probably the best event is this (or similar; may depend on Internet Explorer version):

```
.... CREATE, WINDOW, 0  
ow: class="Internet Explorer_Server", id=0  
pw: class="IEFrame", name="Google - Microsoft Internet Explorer"  
ao: role=PANE, state=0x100040, name="Google", value="http://www.google.com/"
```

Open the Properties dialog, select Accessible object, CREATE. In the right, enter object window class (Internet Explorer_Server) and top-level parent window class (IEFrame). As parent window name, optionally enter Internet Explorer. Click Next and select PANE in the Role combobox. Leave Name and Value fields empty, unless you want to make the trigger specific to this page only.

Click OK. QM will ask whether to insert code to receive trigger properties. Let's say, you need page URL. Then click Yes. It adds two lines of code at the beginning of the macro. Uncomment the second line (delete space at the beginning). The a variable can be used to access and manipulate the PANE accessible object. Its Value property contains the URL of the page. Insert the code to get the Value property. You can use the Accessible Object Actions dialog to insert it.

```
function hwnd idObject idChild  
Acc a.ObjectFromEvent(hwnd idObject idChild)  
str value=a.Value  
out value
```

Triggers: Shell menu

Adds the macro to the shell context menu. It is the popup menu that appears when you right-click a file in a folder window or desktop. To assign this trigger, use the Properties dialog. Unavailable in [portable QM \(259\)](#).

The macro runs when you click the menu item. It receives trigger information through the [function \(152\)](#) statement:

```
\
function $files
```

files - file path. If multiple files selected - list of paths.

Label

Menu item label. To add the item to a submenu, use path like Submenu\Macro. The submenu will be created automatically, or you can create it (read more below).

Single & character makes following character underlined. Use && for &.

QM 2.3.5. Single-hyphen (-) label creates separator.

Other fields are optional.

Include, File types

For what file types to show the menu item. By default, the macro is added to the menu for all files except folders. To add it only for certain file types, enter one or more extensions separated by space, eg txt doc rtf. For files with no extension, use <noextension>.

QM 2.3.5. Works with drives too, like with folders. If need only drives, use ?:\ in Folders field. If need only folders, use ?:\ and check Not.

QM 2.3.5. You can set to display the menu item when user right clicks in desktop or folder background, not on a file. The macro will receive full path of the folder.

Folders

Multiline list of folders where the menu item will be added or Not added. The menu item will be in subfolders too.

QM 2.3.5. Can be used [wildcard characters \(271\)](#). Then compared is whole path of the selected file, not just its parent folder.

Image file

An icon file for the menu item, eg \$my qm\$\copy.ico. Can be ico, bmp or other file. You can specify icon index, eg icons.dll,4. Don't use many icons from executable files, because they are extracted slowly.

Description

Text to display in status bar.

Submenu

If checked, it will be menu item that opens a submenu. The macro can be empty. Use this if you want to assign file types, icon or other properties to the submenu, because there is no way to assign properties to auto-created submenus.

Notes

The number of menu items you can add is limited. On WinXP it is ~150.

QM 2.3.5. Trigger string now begins with \$sm. Previously was ^Shell_menu. QM automatically converts it at startup. Previously this trigger type was implemented as user-defined trigger.

QM 2.3.5. Removed alphabetical sorting. Now menu items are ordered like in QM as they are when QM starts.

QM 2.3.5. Now icons don't disable menu theme.

Triggers: user-defined

Quick Macros can be extended with new trigger types. This topic gives information for programmers about creating user-defined triggers. These triggers can be implemented completely in user-defined functions, or partially in dlls or COM components. They can be added to the Properties dialog, so that other QM users can simply download, import and use the triggers like QM intrinsic triggers.

There are various Windows API functions that allow receiving notifications about various events. Information about these functions can be found in [MSDN Library \(256\)](#). Other way to create a trigger - periodically check some condition, say every 1 second, using [rep \(126\)](#) and [wait \(88\)](#). Also, on the Internet you can find [COM components \(162\)](#) that can be used to receive various system notifications in form of [COM events \(169\)](#). Many notifications are provided by WMI.

The main trigger function should start when QM file is loaded and run continuously. It, for example, can have "QM file loaded" trigger ("Run synchronously" must be unchecked), or started from the trigger management function (read below) using [mac](#).

When your trigger engine is about to start a macro, it should call the [dis \(102\)](#) function to ensure that the macro and user-defined triggers are enabled. Use, for example, this code: `if(dis(macro) or dis&16) ret`.

To implement some triggers, must be used dlls, because the function that receives notifications is called in context of other processes. But most notifications can be received using only QM user-defined functions. QM language has all power to create any trigger that does not require to be implemented in a dll or driver.

Integrating with QM

Your trigger will be easier to use if you integrate it with QM. To integrate your trigger engine with QM, create a user-defined function ("trigger management function"), and assign it "Trigger management" trigger. QM will call the function to communicate with your trigger engine.

At first, about the trigger string. The format of the trigger string that is displayed in the Trigger field on the toolbar is:

```
^Trigger_type trigger data
```

Here **Trigger_type** is the name of the trigger management function, and **trigger data** is some trigger-defined string. The trigger data is optional and is used only by the trigger engine.

The trigger management function must begin with:

```
function# unused1 unused2 action UDTRIGGER&p
```

The first two parameters are always 0.

action - numeric value that defines the purpose for which the function is called.

p - variable of **UDTRIGGER** type that is used to exchange various information. Its members that are used with different actions are described below.

QM calls the trigger management function when:

1. When this trigger type is selected in the Properties dialog.

action - 1.

p.iid - [QM item id \(107\)](#).

p.hwnd - handle of the "Trigger" property page of the Properties dialog.

p.tdata - old trigger data string. It is empty (s.lpstr=0) if the macro does not have a trigger of this type. It is "" if the trigger does not have a trigger data.

The function should create and initialize trigger's property page, and return its window handle. The property page must be modeless dialog, child window of **p.hwnd**. Use `ShowDialog(x x x p.hwnd 1 WS_CHILD)`. If the trigger engine does not provide a property page, the function must just return 0. Then QM will provide default property page to enter trigger data string.

2. When the Properties dialog is being closed (OK) while this trigger type is selected. Is called only if the trigger engine provides a property page.

action - 2.

p.iid - QM item id.

p.hwnd - handle of the property page that was returned in action 1.

p.tdata - empty.

The function must collect data from controls in the property page, format new trigger data string (if any) and store it into **p.tdata**. Should not apply the changes at this moment. Must return 0 if trigger is set, 1 if trigger is not set, or 2 to not close the Properties dialog.

3. When new trigger string must be validated. Is called after a trigger is added or changed.

action - 3.

p.iid - QM item id.

p.hwnd - 0.

p.tdata - new trigger data string.

The function must validate the new trigger data string. It can modify the string if needed. Should not apply the changes at this moment. Must return 1 if the string is valid, or 0 otherwise.

4. When all triggers of this type must be applied. Is called when file is opened, a trigger is added/changed/deleted, macro(s) deleted, and in other cases.

action - 4.

p.iids - array of ids of all macros that have a trigger of this trigger type.

p.niids - number of elements in the array.

The function must remove old triggers and apply triggers to the macros whose ids are in the **p.iids** array. For example, if the trigger engine uses internal trigger tables, it must [re]create the tables. To get trigger strings, use [str.getmacro \(219\)](#) or [qmitem \(107\)](#).

5. When QM needs to display icon of this trigger type.

action - 5.

The function should return icon handle or 0. The icon is destroyed by QM.

6. When the user requests help for this trigger type.

action - 6.

The function should provide help. For example, it can display a message box, or open a web page. Should return 0.

7, 8. When QM or an item is enabled or disabled.

action - 7 if enabled, 8 if disabled.

p.iid - QM item id or 0.

This notification is rarely useful. The function can just return 0. Return 1 if trigger tables have to be recreated; then QM will call this function (later) with action=4.

Only actions 3 and 4 must be implemented. For other actions, the function can just return 0.

The function is always called in QM main thread. If it starts the trigger engine, it must create new thread for the engine (use [mac](#)). Never use QM thread for user-defined triggers, because it can make QM unstable.

The function is not called when it itself is changed in some way (disabled, enabled, deleted, trigger added, removed or changed).

If the function is disabled, it is not called, and does not appear in the Properties dialog. But triggers of that type are not automatically disabled.

You can find a sample user-defined trigger in the [forum](#).

Filter functions

Warning: Incorrectly used filter functions can decrease system performance and stability.

Filter functions usually are used to narrow trigger scope. A filter function is called when QM is about to start the macro to which the filter function is assigned. Depending on filter function's return value, QM starts macro, or not, or starts other macro. You can make the macro specific to a certain window, window part, control, mouse pointer position, time, whatever. Filter functions are especially useful with triggers that "eat" trigger event ("Eat" is checked in Properties).

A macro that has a trigger (key, mouse, window or autotext) can have a filter function. To assign a filter function, use the Filter Function dialog, which appears when you click the FF button in the Properties dialog. To use an existing filter function, click "Use" and select one from the drop-down list. To create new, click "New" (or click "Copy" and select an existing filter function) and optionally type the name for the new filter function. Alternatively, you can specify a filter function directly in the [Trigger \(264\)](#) field on the toolbar.

A filter function is an usual user-defined function. You must write code for it. It must begin with:

```
/
function# iid FILTER&f
```

iid - [identifier \(107\)](#) of the macro that is about to run (a macro to which this filter function is assigned).

f - variable of type FILTER:

```
type FILTER hwnd hwnd2 x y hit !itype !ttype !tkey !tmod !tkey2 []!tmon []!tht
```

hwnd - [handle \(274\)](#) of [top-level \(275\)](#) window.

- For window triggers, it is the trigger-window.
- For mouse click triggers - the clicked window.
- For other triggers - the active window.

hwnd2 - handle of child window or top-level window.

- For key triggers, it is the focused window (can be 0).
- For mouse triggers - window from mouse.
- For other triggers - 0.

x, y - mouse coordinates. Used only with mouse triggers.

hit - hit-test code of mouse trigger. Hit-test codes documented in [WM_NCHITTEST](#) page.

Other members provide some info about the macro. The same as in [QMITEM](#).

Return values for key, mouse and window triggers:

Return value	Action
iid	run the macro. Other macros that have same trigger will not run.
id or name of some other macro	run other macro. For example, the filter function can select a macro from a list of macros that should run from one main trigger depending on other conditions (active window, mouse pointer position, etc).
0	don't run. Other macros that have same trigger also will not run.
-1	don't run, but eat the trigger event (if "Eat" is checked in the Properties dialog). Usually -1 is used when the filter function itself starts other macro (mac (100)).
-2	don't run. Other macros that have same trigger possibly will run.

Return values for [autotext \(29\)](#) triggers:

Return value	Action
iid	run the autotext list item. Other autotext lists that have same trigger will not run.
0 or -1	don't run. Other autotext lists that have same trigger also will not run.
-2	don't run. Other autotext lists that have same trigger possibly will run.

Possible various variants of usage of filter functions. Often a filter function is specific to a single macro, but it also can be assigned to several macros that have different triggers. If several macros must have the same trigger, but run e.g. in different windows, assign the trigger and filter function to one of them (or to the filter function itself); the filter function evaluates

conditions and depending on results allows starting different macros.

In **iid** and **f** variables the function receives information about the trigger. To evaluate it, you can use functions [sel \(125\)](#), [wintest \(77\)](#), [getwintext](#), [getwinclass \(224\)](#), [GetWinId](#), [GetWinStyle](#), [GetWinXY](#), [child](#), [childtest \(83\)](#), [id \(82\)](#), [qmitem \(107\)](#), [getmacro \(219\)](#) and other. Information that the function receives in **f** has different meaning for different trigger types (keyboard, mouse, etc). If a filter function is created for one trigger type, often it cannot be used with other trigger types.

While filter function runs, other application waits and cannot process user input. Therefore filter functions must be as fast as possible. They must only evaluate conditions, also can start other macros ([mac](#)), but itself MUST NOT CONTAIN MACRO COMMANDS (such as mouse, window, wait, dialogs, etc). Max allowed time is 1 second. You can use [perf \(91\)](#) to measure the time.

Filter function runs while the target application is processing some input event. For this reason, some code that communicates with that window (e.g., [acc\(mouse\)](#)) in some cases may not work.

All filter functions of key, mouse and window triggers run in one special [thread \(49\)](#). All filter functions of text triggers run in QM main thread.

If a filter function is disabled, macros to which it is assigned run like without a filter function.

Several filter function templates are available.

When you edit a filter function, to apply the changes click Save or Compile button.

[Examples \(41\)](#)

Filter function examples

Example1

Assume that you have macro "Macro" that has trigger Cb (Ctrl+B), and you want that it would work only in Quick Macros window.

Open the Properties dialog and click the FF button. Click Copy and select "FF_Window" from the drop-down list. It is a template filter function. Click OK, and OK again. New function "FF_Macro" is created. Select it. Initially, text is:

```
/
function# iid FILTER&f

if(wintest(f.hwnd "WindowName" "WindowClass")) ret iid
```

Replace "WindowName" with "Quick Macros", and "WindowClass" with "QM_Editor" (you can see window name and class in the QM status bar while the mouse pointer is over that window). If you want to use only window name or only window class (it is recommended to use only class, unless it is not unique), delete other string, leaving empty quotes. After editing, the filter function may look:

```
/
function# iid FILTER&f

if(wintest(f.hwnd "" "QM_Editor")) ret iid
```

Now, your filter function allows starting "Macro" when you press "Ctrl+B" in Quick Macros window, but discards it in other windows.

Example2

Assume that you have two macros ("M1" and "M2"), and you want that "M1" would run when you press F12 in "Notepad" window, and "M2" when you press the same key in "Internet Explorer".

Instead of assigning F12 trigger to each macro, we will create new filter function and assign F12 trigger to it. The macros should not have a trigger.

At first, create new filter function: menu File -> New -> Templates -> Filter Functions -> FF_Window_Dispatcher. Then open the Properties dialog and assign F12 trigger. Then open the Filter Function dialog, click "Use" and select the new function itself. Click OK, and OK again. Now, the new function has hotkey F12 and filter function itself assigned to it. Function's text is:

```
/
function# iid FILTER&f

sel wintest(f.hwnd "Window1[]Window2[]Window3" "" "" 16)
_case 1 ret "Macro1"
_case 2 ret "Macro2"
_case 3 ret "Macro3"
```

After editing, it should be:

```
/
function# iid FILTER&f

sel wintest(f.hwnd "Notepad[]Internet Explorer" "" "" 16)
_case 1 ret "M1"
_case 2 ret "M2"
```

Tutorial: creating macros, menus and toolbars

[Macro that types text when I press Ctrl+F12](#)

[Macro that runs or activates Notepad when I quickly move the mouse to the right screen edge](#)

[Macro that automatically closes Abc window](#)

[Macro that fills User Id and Password fields in a web page in Internet Explorer](#)

[Menu that appears when I move the mouse up-down two times](#)

[Toolbar for Notepad window](#)

[An "auto-hide" toolbar at the top edge of the screen](#)

See also: [video tutorials](#)

Most of the time, the [Quick Macros window \(4\)](#) is hidden. To show it, click the tray icon .

To enter some frequently used commands, you can use dialogs from the floating toolbar.



You can also [record \(43\)](#) keys and mouse: click the Record button on the [toolbar \(5\)](#), or press Ctrl+Shift+Alt+R.

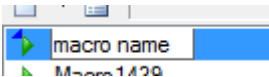
Other [commands](#) you have to type directly in the code editor.

To find commands, dialogs and help, use the 'Find help, functions, tools' field on the toolbar. [More info \(4\)](#).

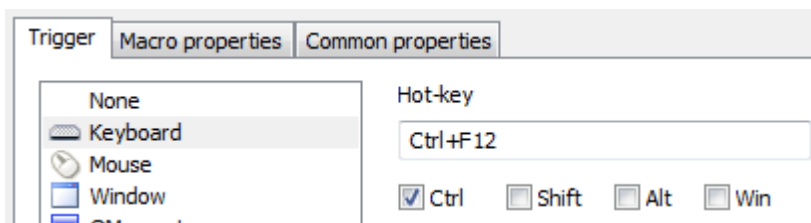


Macro that types text when I press Ctrl+F12

1. *Create new [macro \(20\)](#).* Click the New Macro button on the toolbar. You can type a name for the new macro in the small temporary edit field.



2. *Define trigger (event that causes the macro to run).* Click the Properties button on the toolbar, and then click Key in the [Properties \(10\)](#) dialog. Click the Hotkey field and press Ctrl and F12 at the same time. Then click OK.



3. *Write macro commands.* Only single command is required for this example. Click the Text button on the floating toolbar. Type your text in the Text dialog. Click OK. The macro will look like this:

```
key "First line of your text[]Second line"
```

4. *Test.* Run Notepad and press Ctrl+F12. The macro will type the text.

Macro that runs or activates Notepad when I quickly move the mouse to the right screen edge

1. *Create new macro.*

2. *Define trigger.* In the Properties dialog, select [Mouse \(31\)](#), Right edge, Top (for example).

3. *Write macro commands.* Only single command is required for this example. Click the Files, Web button on the floating toolbar, and then select Run Program from the menu. Will appear the Run Program dialog. In the Path field, type notepad.exe,

or click the Browse button and find the program file or shortcut. Then select or type window name (for example, Notepad) in the Window field. Window name can be partial. Must match case. Then click OK. The macro will look like this:

```
run "notepad.exe" "" "" "" 1 "Notepad"
```

4. *Test.* Quickly horizontally move the mouse to the top quarter of the screen right edge. The macro will launch Notepad, or activate it if already running. Launching macros with mouse movements may require some training.



Macro that automatically closes Abc window

1. *Create new macro.*
2. *Define trigger.* In the Properties dialog, select [Window \(32\)](#). Specify window name and class (use the finder tool, or enter manually). Window name can be partial. Must match case. Click OK, Yes. It inserts one line of code that gets window handle.
3. *Write macro commands.* There are several ways to close a window: the [clo](#) command, press the Esc key, press OK or other button, click it with the mouse. Let's use [clo](#). To enter it, you can use the Window dialog from the floating toolbar. The macro will look like this:

```
int hwnd=TriggerWindow
clo hwnd
```

4. *Test.* Open the Abc window. When it appears, this macro runs automatically and closes it.

Macro that fills User Id and Password fields in a web page in Internet Explorer

1. *Create new macro.* Name it "Id and Password".
2. *Define trigger.* Let's make this macro without a trigger, and run it from a menu. About how to create a menu, read the next example.
3. *Write macro commands.* It is easy if the id field has focus. If not, the macro must select it. We could use the mouse, but the field will not necessary be at the same location every time. To make the macro independent from the id field location, we can use html element functions. In the floating toolbar, locate the Find Html Element dialog. In the dialog, drag the "Drag" picture and drop onto the id field in the web page. When you click the Test button, it must find the id field (you must see black blinking rectangle around it). Click OK. Then open the Html Element Actions dialog, select Set Focus from the list of available actions, and click OK. Then, use the Text dialog to enter id and password, and the Keys dialogs to press Tab (select the password field) and Enter. The macro will look like this example:

```
Htm el=htm("INPUT" "id" "" "Internet Explorer" 0 0 0x221)
el.SetFocus
key "myid"
key T
key "mypassword"
key Y
```


4. *Add the macro to a menu.* Open the menu (or create new). Drag and drop the "Id and Password" macro from the list to the menu text. It will insert:

```
Id and Password :mac "Id and Password"
```

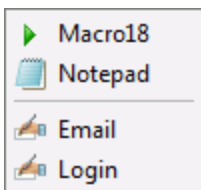
5. *Test.* Open the web page, run the menu and click "Id and Password" menu item.

Tip: To enter id and password, you also can use this dialog: floating toolbar -> Dialogs, Other -> Enter Password.

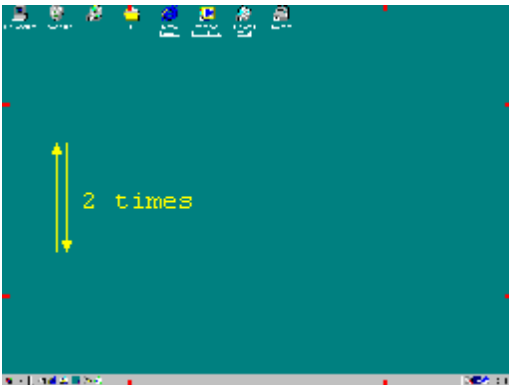
Menu that appears when I move the mouse up-down two times

1. *Create new (22)menu.* Click the small arrow beside the New button, then click New Menu and type a name.
2. *Define trigger.* In the Properties dialog, click Mouse, and, for example, Up-down, Left. Check the Double checkbox.
3. *Add menu items.* Each line in menu text adds one menu item to the menu. To add a menu item that runs a macro, drag and drop the macro from the list to the menu text in the editor. Or, just type macro name. To add an item that runs a file, use the Run File dialog from the floating toolbar. Or, drag the file from the desktop or Windows Explorer. To add items that execute other commands, use other dialogs from the floating toolbar. To add several commands in a single line, separate them with semicolon. To add a separator, type - or |. You can edit item labels (text before ). Menu text and menu itself may look like in this example:

```
Macro18 :mac "Macro18"
Notepad :run "notepad.exe"
-
Email :key "my@ema.il" * text.ico
Login :key "abcd"; key T ; key "1234"; Y * text.ico
```



4. *Assign icons.* Click the Icons button on the toolbar. In the Icons dialog, select an icon from a file (left side), or create and save new icon (right side). In menu text, click the item (line) to which you want to add the icon. In the Icons dialog, click Actions -> Set icon of menu/toolbar item. Then select and assign icons to other items.
5. *Test.* Quickly vertically move the mouse up-down two times in the left side of the screen. [Launching macros by mouse movements \(31\)](#) may require some training.



Toolbar for Notepad window

1. *Create new (23)toolbar.* Click the small arrow beside the New button, then click New Toolbar, and type a name.
2. *Define trigger.* In the Properties dialog, select [Window \(32\)](#), Active. Drag the "Drag" tool and drop onto the Notepad window. Delete document name (e.g., if the window name is "Untitled - Notepad", leave only " - Notepad").
3. *Add buttons.* All the same as with popup menus (see the above example).

Now, when you open Notepad, you can see this toolbar attached. You can move the toolbar with the right mouse button, resize with the left button. Right click to view the context menu. Shift+drag to move or delete buttons. Drag and drop a macro, file or Internet link to add it to the toolbar (you can drop onto the editor text or directly onto the toolbar).

An "auto-hide" toolbar at the top edge of the screen

1. *Create new toolbar.* Click menu File -> New -> Templates -> Toolbar Top or Bottom.
2. *Define trigger.* In the Properties dialog, select Mouse, Top edge, Middle. Check the Slow checkbox. Or, you can assign it "QM file loaded" trigger to run at startup.
3. *Add buttons.* See the above example.

The toolbar will appear when you move the mouse to the left part of screen top edge.

Recording

You can record keys, mouse clicks, mouse movements.

To start recording, click the Record button on the toolbar, or press Ctrl+Shift+Alt+R. To end recording, click Insert or ... button in QM Recording window.

Recording options in the 'QM recording' window

The check boxes:

- *Keys* - record keyboard events ([key](#)).
- *Mouse* - record mouse clicks ([lef](#), [rig](#), [mid](#), [dou](#)).
- *Drag* - record mouse movements while a mouse button is pressed ([mou](#)).
- *Move* - record mouse movements while mouse buttons are released ([mou](#)). Mouse movements before the first and after the last other event are not included.
- *"text"* - convert text key codes to characters. For example, when you type "New York", it records ""New York". If unchecked, records key codes: 'SnewVSyork (here S is Shift, V is Spacebar). However in some cases may record incorrect characters, because the active window interprets the keys differently.
- *Images* - when recording mouse clicks, take screenshots (small images) to display in the code editor. [More info \(12\)](#).

Mouse coordinates relative to - select what coordinates to record - relative to window, control (child window) or screen. If window, in most cases records client area coordinates, as it is more reliable.

Speed:

- *Fast* - don't record pauses. The macro will run reliably in most cases, but sometimes you should insert [wait \(88\)](#) or change macro speed ([spe \(90\)](#)).
- *Slow* - same as *Fast*, but adds [opt slowkeys and opt slowmouse \(97\)](#). The macro will run slower.
- *Real* - record pauses, except between most keys.
- *Variable* - record pauses (same as *Real*), multiplied by a variable F. You can change F value to change macro speed. For example, to run macro 4 times faster, set F=0.25.

The *Insert* and ... buttons:

- *Insert* - insert the recorded code in current macro, at current position.
- *Replace all text* - replace current macro text with the recorded code.
- *Create new macro* - create new macro with the recorded code.
- *Replace text of macro "temp"* - open or create macro named "temp", and replace its text with the recorded code. Can be used to record temporary macros.

How to quickly record single command

You can use the following ways to insert a single command at any time. Don't need to start recording mode.

- Press Ctrl+Shift+Alt+W. It is a global hotkey, can be changed in Options. You can record [win](#), [child/id](#), [wait](#), [act](#) or [lef](#). It records window/control/coordinates from mouse.
- To record menu commands ([men](#)), click menu Tools -> Record Menu, and then click the menu item you want to record.
- To insert file path or Internet link ([run](#), [web](#)), drag it from desktop, Start Menu, folder window or web browser.
- To insert "run macro" command ([mac](#)), drag that QM item (macro, menu, etc) from the list.

Syntax of QM macro code

QM macro text is a sequence of statements. Usually, a statement is a single line of code that performs some operation or declares some identifier. Examples:

```
lef+ 10 200 "Notepad"
lef
int a b c
a = b * 100
Func b 1
a = Func(b 1)
```

The first and second statements are *macro commands*. They consist of the following *parts*:

- **lef** is command *keyword*.
- **+** is *option*.
- 10, 200 and "Notepad" are *arguments*. An argument is the actual value for a parameter of the command/function.

The third statement declares *(140)variables* a, b and c.

The fourth statement assigns *expression* b*100 to variable a.

In the fifth statement, is called *function* **Func** with arguments b and 1.

In the last statement, is called function **Func**, and its return value is assigned to variable a.

10, 200, 100 and 1 are *numeric (137)constants*.

"Notepad" is *string constant*.

= and * are *(133)operators*.

Parts of statements must be separated with spaces (except command/option). In statements with operators (example 4), spaces may be omitted. To separate arguments, also can be used commas (,).

The list of arguments must be enclosed (example 6), but if function name begins the statement (examples 1 and 5), it is not necessary. With enclosed arguments, the () must be immediately after function name and option character (if any).

Some characters at the beginning of a line have special meaning:

Space, ;, /, or \ disables the line. This can be used for comments. To easier disable/enable a line (or several selected lines), right click the selection bar. Comments also can follow a statement. For such comments use ; ; .

Tabs or commas are used with flow-control statements (if, rep, etc). Example:

```
if a is less than 10, left-click, else exit
if a<10
lef
_a + 1 ;;increment a
else ret
```

A single line can contain several statements separated by semicolon (;). Semicolon is optional after a statement that begins with command/function name and has enclosed arguments (or empty parentheses). Semicolon also is optional after **else**, **err** and **case**. Examples:

```
lef+ 10 100 Notepad; lef; int a; a = b + 100; Func(a b); b = Func(a b)
rep() if(b>a) b=Func(a b); else break
```

Syntax descriptions used in this Help file

Gray symbols have the following meaning:

[a]	a is optional.
a b	a or b.
a&b	a and/or b.

<code>(a b) (c d)</code>	a b or c d .
<code>(space)a</code>	before a must be space or semicolon.
<code>(tab)a</code>	before a must be tab or comma.
<code>...</code>	more parameters or statements.
<code>int a</code>	a is function; it returns a value of type int.

Syntax description example:

```
lef [+|-] [x y] [window] [client]
```

In the example, all parameters are enclosed in `[]`. It means that you can use 0, 1, 2, 3 or 4 arguments with the command.

The `[+|-]` means that you can optionally use option character + or -.

The `lef` keyword in the example is at the beginning. It means that the `lef` command does not return a value and cannot be used as function.

Real code examples based on the above syntax description:

```
lef 10 200 "Notepad"
lef
lef+ 10 200
```

Programming in QM

[Predefined commands and functions](#)

[Constants](#)

[Variables](#)

[Operators](#)

[Functions](#)

[If-else](#)

[Go to](#)

[Repeat](#)

[String manipulation](#)

[Errors](#)

[Threads](#)

As you can read in the [syntax \(44\)](#) topic, QM macro code is a sequence of various statements - macro commands, functions, calculations, [declarations \(242\)](#), flow-control and everything you need for programming. A single line of code can contain a single statement or several statements separated by semicolons. For comments, use space at the beginning of line, or two semicolons after statements. Tabs at the beginning of a line are used with flow-control statements ([if](#), [rep](#) and other). Example:

```
if a is less than 10, left-click, else exit
if(a<10)
  _lef 5 a "Some Window"; wait 1
  _a + 1 ;;increment a
else ret
```

Predefined commands and functions

Quick Macros has about 200 built-in [keywords \(52\)](#). They include macro commands, functions, member functions, flow control statements, declaration statements, intrinsic types, compiler directives and operators. You can also use [dll functions \(153\)](#), [COM functions \(162\)](#), and [user-defined functions \(21\)](#).

QM offers several [features \(46\)](#) to simplify programming. You can use [F1 \(245\)](#) to get help for functions and other identifiers. Basic syntax or definition of these identifiers also is displayed in QM status bar. To browse and insert available identifiers, you can use lists of members. Some commands may be entered through dialogs, or recorded.

Constants

[Constants \(137\)](#) are simple numeric or string values, such as `45` (integer constant), `0x1A` (integer constant in hexadecimal format), `10.57` (floating-point constant), `"text"` (string constant). String constants are enclosed in double quotes. To insert a double quote in a string, use two single quotes (' '). For new line, use `[]`. Constants can be assigned to variables, used as arguments of macro commands and functions, and in expressions with operators. You can also define [named constants \(139\)](#).

Variables

[Variables \(140\)](#) are used to temporarily store data (numeric, text, etc) at run time. The data can be changed during various operations. Like constants, variables can be assigned to variables, used as arguments of macro commands and functions, and in expressions with operators.

Before using a variable, you must [declare \(141\)](#) it (except [predefined variables \(144\)](#)). Declaration includes type and [name \(257\)](#). Mostly used types are `int` (for numeric integer values), `str` (for strings) and `double` (for numeric floating-point values). In the following example, we declare [local \(142\)](#) variable `i` of type `int`, local variable `S3` of type `str`, and global variable `g_var` of type `double`. Then we assign some values:

```
int i
str S3
double+ g_var
i=5
S3="text"
g_var=15.477
```

You can also use [pointers \(147\)](#) and [arrays \(146\)](#). In the following example, we declare array `a` of `str` variables, allocate 10 elements, then store string "abc" into element with index 0:

```
ARRAY(str) a.create(10)
a[0]="abc"
```

You can [define new types \(154\)](#). User-defined types usually have several member variables that may have different types. For example, type **POINT** has members x and y. In the following example, we declare variable p of type **POINT**, and assign 10 to its member y:

```
POINT p
p.y=10
```

Operators

An [operator \(133\)](#) is a special symbol used to perform assignment, arithmetic, comparison or other operation. Several examples with comments:

```
i = 5 ;;assign 5 to variable i
i + 2 ;;add 2 to variable i
a = b + 10 ;;calculate sum of b and 5 and assign it to variable a
ave = i + j / 2 ;;add j to i, divide by 2, then assign result to variable ave
f = i - (10 * j) + func(a b) * -10 ;;multiply 10 and j, extract result from i, call function Func and add its
return value, multiply by -10, then assign result to variable f
str s = "notepad" ;;declare variable s and assign string "notepad"
s + ".exe" ;;append ".exe"
if(i<10 and s="notepad.exe") i = j/10 ;;if i is less than 10, and s is equal to "notepad.exe", then calculate j /
10 and assign result to variable i
i = Func2(j s (i + 5) f) ;;call function Func2 and assign its return value to variable i; pass four arguments; the
third argument is sum of i and 5
lef a-10 100 ;;left click; x is a-10, y is 100
```

In QM, operators belong to 3 priority classes: first are calculated arithmetic and bitwise operators, then comparison operators (=, !, <, etc), and lastly logical operators ([and](#), [or](#)). Operators that belong to the same priority class are calculated from left to right. Parentheses can be used to change this order.

See also: [unary operators \(134\)](#) (- and other), [member access operator \(155\)](#) . and [array element access operator \(146\)](#) [].

Functions

See also: [functions \(149\)](#), [user-defined functions \(21\)](#), [sub-functions \(182\)](#), [function tips \(150\)](#).

A function is a named piece of code executed as a unit. It can receive several values (arguments) and return some value. Beside the predefined functions, you can create your own (user-defined) functions. Usually, you create a function when you want to have a piece of code that can be executed more than once, in any macro. Instead of placing the same code in each macro, you place it in a function, and then call the function by name from any macro. Like constants and variables, functions can be assigned to variables, used as arguments of macro commands and other functions, and in expressions with operators. Examples:

```
Func a "text" 100 ;;call function Func with three arguments
a = Func2(10) ;;call function Func2 with one argument, and assign its return value to variable a
Func3(a Func4(b c)) ;;call function Func4 with arguments b and c, then call function Func3 with two arguments
(second argument is return value of Func4)
d = e + Func5(b c) / 10 ;;use function Func5 in expression with operators
```

Some variable types have member functions. A member function is called with a variable of that type. Example:

```
str s s2="notepad"
s.from(s2 ".exe")
if(s.end(".exe")) out "s ends with '.exe'"
```

To create new [user-defined function \(21\)](#), select New Function from the File/New menu and type function's [name \(257\)](#) in the temporary edit field. To define function's return type and parameters, use the [function \(152\)](#) statement at the beginning. To return a value, use the [ret \(130\)](#) statement. By default, a function can be called from code or launched as a macro. To allow it only to be called from code, the first line should be space and /, as in the following example:

```

/
function'int str'a str&b [int'c]
statements
ret 1

```

This function returns a value of type int. It has three parameters (local variables that receive the passed values (arguments) when the function is called). Variables a and c receive copies of passed values. Variable b is reference to the str variable that is passed. When this function modifies b, it actually modifies that variable. The last argument is optional, that is, caller can omit it, in which case c is 0. The function executes statements and returns 1. If it returns not through `ret`, or `ret` is without a value, the return value is 0.

If-else

Often you need to execute or skip several statements depending on some condition. Example:

```

if a>0
  _out "a is greater than zero"
  a-1
else if a=0
  _out "a is equal to zero"
else
  _out "a is less than zero"

```

The [if \(123\)](#) statement evaluates some expression (in this case, `a>0`), and, if the expression is true (nonzero), allows to execute the following tab-indented statements. The `if` statement optionally can be combined with the `else` statement, which allows to execute the following tab-indented statements if the expression is false (zero). As you see in the example, `else if` can be used to evaluate more expressions. The statements also can be in the same line:

```

if(a>0) _out "a is greater than zero"; a-1
else if(a=0) _out "a is equal to zero"
else _out "a is less than zero"

```

When need to compare a variable with multiple constant values, it is more convenient to use [sel \(125\)](#).

Go to

The [goto \(122\)](#) statement is used to jump to some other place in code. Example:

```

goto g1
statements
g1
statements

```

In the example, statements after `goto` are skipped, and execution continues from the statement that follows the label `g1`. Note that the label is preceded by single space (same as comments).

Repeat

The [rep \(126\)](#) statement is used to repeatedly execute the following tab-indented statements. It optionally can include number of times to repeat. To exit the loop, use the `break` statement. The statements also can be in the same line. Examples:

```

rep(100) i-1

```

In this example, `i-1` is executed 100 times. In the following example, `_out i` is executed while i is less than 100, that is, also 100 times:

```

i=0
rep
  if(i>=100) break
  _out i
  i+1

```

You can also use [for \(127\)](#) (repeat with counter variable) and [foreach \(128\)](#). Example:

```

for i 0 100

```

```
out i
```

This example does the same job as the second example with [rep](#). Initially, variable `i` is set to 0. Then, `out i` is repeatedly executed, and after each loop `i` is incremented. This process stops when `i` becomes equal to 100.

String manipulation

QM has a set of functions to process [strings \(183\)](#) (text). To store and manipulate strings, use variables of type `str`. In the following example, we declare variable `s` and assign string "Cat".

```
str s = "Cat" ;;now variable s is "Cat" (string "Cat" is stored in variable s).
```

There are two groups of string functions. The first group - global functions - are used to get certain information about a string (length, numeric value, find substring, etc) or a character. Examples:

```
i=val(s) ;;i = numeric value of string s (or 0, if s does not begin with digits).
i=find(s "substring") ;;i = index of first character of "substring" in string s (or -1, if "substring" is not found in s).
if (IsCharLower(s[0])) s[0]=CharUpper(+s[0]) ;;if first character of string s is lowercase, convert it to uppercase.
```

The second group - `str` member functions. Most of them are used to build or modify string. The calling syntax slightly differs: function's name is preceded by variable's name and dot. Examples:

```
s.from("word1" " ", " " "word2") ;;build string variable s from three strings (join). Now s is "word1, word2". Here s
is variable of type str.
s.ucase ;;make s uppercase. Now s is "WORD1, WORD2".
str ss.get(s 7 5) ;;declare str variable ss and get part of s. Now ss is "WORD2".
```

With `str` variables, you can use operators `+` and `-` to append and prepend:

```
s+".txt" ;;append ".txt"
s-"C:\Doc\" ;;prepend "C:\Doc\"
```

To compare `str` variables, use operators `=` (equal, case sensitive), `~` (equal, case insensitive) and `!` (not equal, case sensitive). To compare part of string, use `str` member functions `beg`, `mid` and `other`. Examples:

```
if (s~"Monday") ... ;;if s is "Monday" (case insensitive) ...
if (s.endi(".exe")) ... ;;if s ends with ".exe" (case insensitive) ...
```

Errors

When you launch a macro, QM [compiles \(47\)](#) it, which includes checking for syntax [errors \(48\)](#). On error, QM highlights the error place and gives error description. Macro with errors is not executed.

At run time, in certain conditions, a macro command may fail. Then QM generates run-time error, and the macro ends. The [err \(129\)](#) statement allows you to continue execution after a run-time error in the preceding statement. Examples:

```
clo "Notepad"; err

5 "Notepad"
err
run "notepad.exe"
```

In the first example, `err` allows to continue if an error occurs. In the second example, on error is executed `run "notepad.exe"` statement, which would not be executed otherwise.

Threads

A [thread \(49\)](#) is like a subprogram within a running program. In QM it is a running macro or function, including all functions that it calls. Multiple [function \(21\)](#) threads can run simultaneously, including several instances of the same function. Multiple [macro \(20\)](#) threads can run simultaneously if the macro has option 'Run simultaneously'.

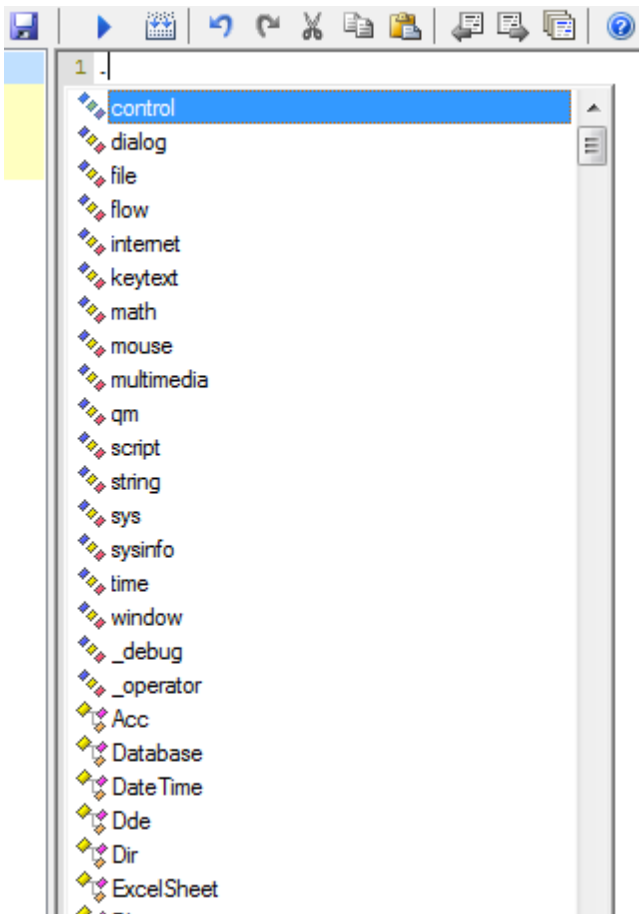
Help and type info

To enter some often used macro commands, you can use dialogs from the floating toolbar. If the toolbar is unavailable, click the "Check extensions" button in Options and follow instructions.



However, Quick Macros does not provide dialogs for all commands and functions. You will have to type them directly in the editor. You can find all intrinsic commands and functions through the [reference \(52\)](#) topic. There are several features that can help in finding commands and getting help.

When you type dot (.) somewhere in text (for example, at the beginning of a line), appears list of functions and other identifiers that you can use. Various type kinds (functions, types, constants, variables, etc) have different [icons \(242\)](#). To insert an identifier, double click it. Or begin to type, and press Tab or Enter to complete.



At the top of the list are [categories \(159\)](#) - collections of related functions. Type `.` after a category name to view the functions. At the bottom of the list are various libraries. Type `.` after a library name to view its contents.

Press:

- Ctrl+, to view only types.
- Ctrl+. for global functions.
- Ctrl+/- for constants and variables.
- Ctrl+; for local/thread variables, also member variables and functions of current class.

Also there are many other functions (member functions) that are displayed when you type `.` after a variable of a certain type, for example str, Acc, Database, ARRAY, BSTR, a COM interface. To use such functions, declare a variable of that type, and call functions with that variable. Example:

```
Ftp f
f.Connect("ftp.myserver.com" "user" "passw")
f.DirSet("public_html")
```

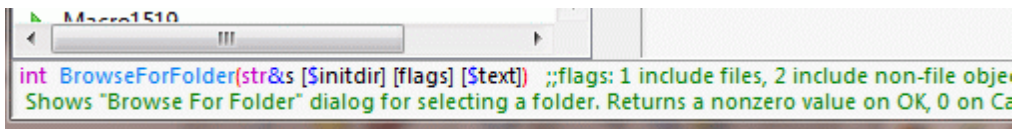
In the lists, names of hidden and restricted members are gray.

If you want to hide a user-defined function in the list, place it in a [private folder \(11\)](#). The list also does not show anything that begins with `__` (two underscores). To show hidden items, press `Ctrl+Shift+.` or use menu `Edit -> Members -> Show Hidden`.

To show the list when you already partially typed an identifier, choose the Completion command (`Ctrl+Space`). In most cases it shows only identifiers that begin with the same letter. `Ctrl+.` and other commands also can be used for this.

When the text insertion point is on an identifier (function, type, etc, except variables), QM status bar shows some info for it. You can `Ctrl+click` to view it in QM output.

- Optional parameters are [enclosed].
- `|` means OR.
- Some parameter names are preceded by type name or character, like in [function declaration \(152\)](#). For example, string parameters are preceded by `$` or `~`. If it is `~`, you can pass a string or a number. If type not specified, it is `int` (integer number).
- If a parameter supports values of several types, shows ``` or the mostly used type.
- Short description and other additional info is displayed as comments (green). Read [here \(245\)](#) how to add function info to the status bar.



The status bar info for intrinsic and user-defined functions is always available. For other user-defined identifiers, it is available only if the identifier is already [declared \(242\)](#). That is, if the macro or function with the declaration is already [compiled \(47\)](#), or the identifier is from a [type library \(164\)](#) or [reference file \(160\)](#). QM also automatically declares identifiers from type libraries and reference files when typing `typelib.identifier`. QM also may look in reference files to show info in QM status bar.

To show help for a function or other identifier, type or click it and press [F1 \(245\)](#).

Compiling and debugging

Compiling

Before a macro runs first time, or after editing, QM at first compiles it. Also compiles when you click menu Run -> Compile, or when you create [exe \(51\)](#). The compiled code is not native machine code, but it is executed much faster than a non-compiled script.

If there are syntax errors, the compiler [stops \(176\)](#), shows error message in QM output, and highlights that statement or part of statement.

Tips for testing and debugging macros

Debugging - finding and fixing errors.

Useful functions:

- [out \(57\)](#) displays variables etc in QM output.
- [mes \(62\)](#) message box.
- [deb \(99\)](#) starts debug mode.
- [bee \(96\)](#) plays short sound.
- [LogFile](#) writes to a log file.
- [OutWinMsg \(114\)](#) displays Windows message name.
- [Statement \(114\)](#) gets a statement (code line) and/or id of the function or one of its callers.
- [GetCallStack \(114\)](#) gets or displays name+statement of the function and its callers.

To quickly disable/enable a single line or several selected lines, right click the selection bar.

To handle run-time errors, use [err \(129\)](#).

Debugging and error handling functions and variables are in `_debug` [category \(159\)](#).

See also: [common errors \(48\)](#)

Errors

See also: [compiling and debugging \(47\)](#), [err \(129\)](#), [#err \(175\)](#).

Compile-time errors

Compile-time errors are generated when [compiling \(47\)](#) macros that contain syntax errors. Macros are compiled before they start. They don't start if there are syntax errors. The first syntax error is displayed in the QM output like "Error in Macro: error description". Compile-time errors are not generated in [exe \(51\)](#), because in exe the macro is already compiled.

Some compile-time errors and possible reasons/solutions:

- *unknown identifier*:
 1. Forgot to [declare a variable \(141\)](#) or other identifier. [Declaration \(242\)](#) must go before usage.
 2. Character case mismatch. For example, if a function is MyFunc, error if you use Myfunc.
 3. Forgot to enclose in `""` a string argument (window title, file path, etc).
 4. The function/type/etc is not available in this QM version. It may happen if the macro author is not you. Upgrade QM.
- *unexpected identifier*:
 1. This type of identifier cannot be used here.
- *unexpected character*:
 1. Names of variables and other identifiers must consist of alphanumeric characters and underscore, and cannot begin with a digit.
 2. Incorrect syntax.
- *missing parts*:
 1. Forgot some argument or other parts of the command/function/etc.
- *too many parts*:
 1. Forgot to enclose in `""` a string argument.
 2. Forgot `;` after statement that is followed by more statements in the same line.
 3. If a function argument is quite complex expression (e.g., `a.r[i].right-a.r[i].left`), enclose it in parentheses.
- *missing (after function name, or ; after statement*:
 1. The same reasons as above.
 2. The (must immediately follow function name. Correct: `Func(...)`. Error: `Func (...`.
- *else without if*:
 1. Indentation (number of tabs) of `else` must match indentation of its `if`.
- *case without sel*:
 1. Indentation (number of tabs) of `case` must be one more than indentation of its `sel`.
- *expected numeric (or string) expression*:
 1. Used a string where there must be a numeric value, or vice versa.
 2. Forgot some argument.
- *expected int, integer constant, etc*:
 1. Similar reasons as above.
- *expected int*, etc*:
 1. Must be address of a variable (e.g., `Function(&var)`, but not `Function(var)`). Variable type also must match expected. For type casting, use [operator + \(134\)](#).
- *... already exists or is declared*:
 1. Declaring a variable or other identifier while it is already somewhere declared differently.
 2. Declaring a local variable more than once in the same function.
- *expected x (to y) arguments ...*:
 1. Forgot some arguments. When calling a function, the number of arguments must match the number of parameters in function's declaration ([function](#), [dll](#), etc).
- *syntax*:
 1. other error.
- *Exception while compiling*:
 1. An internal QM error. Try to restart QM.
- If QM reports a false error (possibly a bug), try to restart QM.

Run-time errors

Run-time errors are generated while macro is executed. On a run-time error the macro ends. In the QM output is displayed error description, like "Error (RT) in Macro: error description".

There are three groups of run-time errors:

1. Errors that are generated by QM ("Window not found", "File not found", etc). Such errors can be handled with the [err \(129\)](#)

statement. There are also several fatal errors that cannot be handled, such as a noncompiled function containing syntax errors.

2. Errors that are generated using the [end \(131\)](#) statement. Such errors can be handled with [err](#).

3. Exceptions (errors generated by the operating system or components). In most cases it is result of incorrect programming (invalid pointer, division by 0, endless recursion, etc). Exceptions can be handled with [err](#), but you should avoid them if possible.

Some run-time errors and possible reasons/solutions:

- *Window not found:*
 1. Window name must match case.
 2. Window is hidden: use `opt hidden 1` before, or use class name.
 3. Part of window name (e.g., document name) now is different than when recording. Use partial name.
- *Menu item or button not found:*
 1. Underlined characters must be preceded by &. To view underlined characters, expand menu with the keyboard (Alt+...).
 2. Nonstandard menu.
 3. If button class name does not have "Button" or "Btn", don't use `but name [window]`. Instead use [id](#) or [child](#) function.
 4. Button cannot be found using its text because has "owner-draw" style. Try function [id](#), or [child\(x y\)](#), or [scan](#).
- *0x80020003, Member not found* (with accessible objects):
 1. The accessible object does not support this feature.
 2. The window is inactive. Use [act](#) to activate it.
- *object not found*, and other:
 1. The macro runs too fast. Insert [wait \(88\)](#) or change macro speed ([spe \(90\)](#)).
- *cannot paste:*
 1. The focused control is not editable.
 2. The focused control does not support Ctrl+V. Some controls use Shift+Insert instead.
 3. The focused control does not support this clipboard format.
 4. The window is disabled or hung.
 5. The window has higher [integrity level \(277\)](#). Eg, QM runs as User, and the target program runs as Administrator.

Threads

A thread is a running macro or function. All functions it calls run in the same thread.

A thread is the basic unit to which the operating system allocates CPU time. It executes code. Each thread runs simultaneously with other threads. Every process (running program) has one or more threads. To create effect of several simultaneously running threads, operating system divides CPU time into time slices (~20 ms). It allocates a CPU time slice to each thread that need it. The currently executing thread is suspended when its time slice elapses, allowing another thread to run. If there are multiple CPU, threads may run simultaneously on different CPU. You can read more in [wikipedia](#).

QM creates new thread for each macro or function started by the user, a trigger, [mac \(100\)](#) or [tim \(92\)](#). It also creates new thread for menu/toolbar/autotext items. Multiple threads can run simultaneously, except macros without option 'Run simultaneously'. Multiple threads of the same function etc also can run simultaneously.

All currently running threads, except special threads, are displayed in the Running Items list. To see the list, check View Active Items in the Run menu. Private threads are hidden by default. To see threads without opening QM window, use QM tray menu.

See also: [EnumQmThreads \(114\)](#), [IsThreadRunning \(114\)](#), [EndThread](#).

Special threads

There are two threads that run all the time:

1. QM main thread. Functions that run in it: global variable constructors/destructors, toolbar hook functions, functions executed during compilation and some other functions. Also, a macro or other application can send [message \(18\)](#) to run a function in the main thread.
2. Thread of [filter functions \(40\)](#). All filter functions of key, mouse and window triggers run in this thread.

[Thread variables \(142\)](#) of these threads are destroyed when unloading [QM file \(17\)](#), before destroying global variables.

Threads also can be explicitly or implicitly created by some dll functions, for example [CreateThread](#). Avoid such functions if possible, because then QM cannot destroy thread variables and free some other memory allocated for that thread.

In all these special threads don't use [atend \(103\)](#).

Notes

Before a thread ends:

1. QM 2.3.4. All top-level windows of that thread receive [WM_QM_ENDTHREAD](#) message. They should destroy self ([DestroyWindow](#)) and free their allocated data to avoid memory leaks.
2. Called functions registered with [atend \(103\)](#).
3. Destroyed local variables of thread main function.
4. Destroyed thread variables.

Quick Macros extensions

If you have some programming experience, you can do with QM most of things that you can do with C++ or other programming languages. You can create and call functions, use dll and COM functions (Windows API, C run-time library and other), create dialogs, create new trigger types.

You can share useful functions and macros (import, export), for example in Quick Macros [forum](#). If you share your macros/functions, remember that they may not work on older QM versions. You should provide version info, e.g. "Tested with QM 2.3.5, Windows 7.". Macros compiled to [exe files \(51\)](#) don't require QM and therefore will always work.

Many QM features are implemented as QM extensions. They are in System.qml [file \(17\)](#) (the System folder in the list of macros). Many functions, declarations, dialog editor, floating toolbar etc are there.

Many other functions are in Quick Macros [forum](#). If you cannot find a function you need, you can request it in the forum.

Make exe

By default, macros run in QM, but you also can create standalone programs (exe files) from them. Exe files run like macros, but don't need QM. It is especially useful if you distribute them. Distributing exe also is better because it does not depend on QM version, whereas macros that run in QM may not run on older QM versions.

To make exe from current macro or function, click menu Run -> Make Exe.

The "make exe" feature also is used to [run macros in separate process](#). QM creates a .qmm (compiled macro) or .exe file and executes it in separate process. Therefore in most cases here terms "exe" or "program" mean ".exe program or a macro that runs in separate process".

[Options](#)

[Notes](#)

[Resources](#)

[Features that are unavailable in exe](#)

[Features that are different in exe](#)

[Running on Windows Vista/7/8/10](#)

[Properties -> Run in separate process](#)

[License](#)

Options

Main - the macro or function that runs first when the program runs.

Make - the program file to create. Can be .exe or .qmm (used with the "run in separate process" feature).

Icon - program's icon. It also will be used in dialogs etc. Can be ico, exe, dll, ocx. Icon index can be specified like shell32.dll, 18.

Manifest - [manifest file](#).

It is optional but should always be used. You can use the default file default.exe.manifest. Manifest enables visual styles (dialogs will look nicer). For Windows Vista/7/8/10, manifest specifies whether the exe must run with higher privileges. Read more [below](#).

.res - an optional .res file containing resources to add to exe [resources](#).

Auto add files... - add files used in exe macro/functions to exe [resources](#). Adds only files where a resource id specified.

Example: `scan ":10 $my qm$\file.bmp"`.

Version - file version and other optional info. Valid version formats: 1, 1.2, 1.2.3, 1.2.3.4. The list in the 'Version info' dialog can include anything.

End hotkey - a hotkey that can be used to end the program. [Notes](#).

Only a single program can use the hotkey. The hotkey will not work if another instance of the program or another program already uses that hotkey. For example, Windows Explorer may use it for a shortcut. Don't use hotkeys with modifiers (Ctrl, Shift, Alt) if the macro uses keyboard/mouse commands, because then user-pressed modifier keys will be mixed with macro-pressed keys/buttons and the result may be dangerous.

[Run functions when creating exe](#)

When creating exe, QM can call your functions that automate related tasks. The functions receive full path of the program in the `_command` variable.

Before - a function to call before starting to create exe (when you click OK). QM waits while it is running. Error if it returns 0. The function, for example, can end the program if it is running (you can use function [MakeExeCloseRunning](#)).

After - function to call after the exe is successfully created. QM waits while it is running. Error if it returns 0. The function, for example, can add the exe file to a zip file. Example:

```

str s1 s2

exe file
s1=_command

other files
s2=
  $qm$\a.ico
  $qm$\b.bmp
  $qm$\c.dll
s1.from(s1 "[]" s2)

add all to zip
zip "$my qm$\myexe10.zip" s1
ret 1

```

On Run - function to run when the program is launched from QM. If used, QM does not run the program. The function, for example, can run the program with a command line. Example for .exe files: `run _command "/command line"`. Example for .qmm files: `run "qmmacro.exe" F""{ _command} ' ' /command line"`. To run with command line or pass function arguments, also can be used `mac`.

Debug

Show added items - display in the output all added functions, items that are added as text, ActiveX control classes (read below), and image files (read below).

Run immediately - run the program or the "Run on Run" function automatically when the program is created.

Output to QM - if QM is running, display all output of the program (`out`, RT error, etc) in QM. If unchecked, `out` does nothing, unless redirected (`RedirectQmOutput (114)`, `ExeOutputWindow`, `ExeConsoleRedirectQmOutput`). If unchecked or redirected, RT error is shown in a timed message box.

Go to error - if QM is running, show RT errors of the program in QM (open the function and show the place). This works correctly only if the function is not modified and not renamed since the program was created last time.

Note: the last two options are active even if the exe runs on other computers where QM is installed and running.

Console (QM 2.4.1) - create console exe. A console process has a console window or uses console of parent process (eg cmd.exe). Also, cmd.exe does not wait until a non-console process ends. [Console exe code example](#).

```

SetConsoleTitle "QM test console"
ExeConsoleRedirectQmOutput ;;redirect out etc to ExeConsoleWrite

ARRAY(str) a; int i
ExeParseCommandLine _command a
for(i 0 a.len) out a[i]

ExeConsoleWrite "Type something and press Enter"
str s
ExeConsoleRead s
ExeConsoleWrite F"s='{s}'"
2

```

These macro/function properties also are applied when creating exe: "allow only single instance" (function) and "don't run/wait/terminate" (macro). Read more below.

Notes

Where are saved Make Exe dialog settings

When creating an exe first time, you can choose where to save its settings - in current macro or folder. Choose to save in folder if exe will have more functions that are in the folder; then later these settings will be used when you make (update) exe

51. Make exe

while any of the functions is open. The folder name should be the same as the program name (e.g. "X Editor"); it is used in exe, for example in error message boxes. You can find exe settings in Folder Properties dialog.

How to run exe

You can use whatever way that you can use to run programs (double click in Windows Explorer, etc).

If you click the Run button, the macro runs in QM, as usually. To run exe when you click the Run button, check "[Run in separate process](#)" in Properties.

Source code

When creating exe, QM compiles the main function and all other used functions, including threads created using [mac](#). Compiled code is added to exe. Source code is not added, unless need to get certain text from it at run time (read below). In some cases need to use [#exe \(179\)](#) to explicitly add some functions and other data. To see what is added, check 'Show added items'.

Full text (source code) of a macro/function is added to exe if it contains a dialog definition used with ShowDialog, a menu definition used with ShowDialog or DT_CreateMenu, or an other-language script used with CsExec, CsFunc, CsScript.AddCode, CsScript.Compile, CsScript.Exec, WshExec, VbsExec, VbsAddCode, JsExec, JsAddCode, PsCmd, __Tcc.Compile. To avoid adding full source code, store dialog definitions etc in separate macros or in variables. [Example](#).

This source code will be added to exe because need to extract the dialog definition from it.

```
BEGIN DIALOG
0 "" 0x90C80AC8 0x0 0 0 224 136 "Dialog"
1 Button 0x54030001 0x4 116 116 48 14 "OK"
2 Button 0x54030000 0x4 168 116 48 14 "Cancel"
END DIALOG

if(!ShowDialog("This Macro" 0 0)) ret
```

This source code will not be added to exe because the dialog definition is in a variable. Will be added only the dialog definition string.

```
str dd=
BEGIN DIALOG
0 "" 0x90C80AC8 0x0 0 0 224 136 "Dialog"
1 Button 0x54030001 0x4 116 116 48 14 "OK"
2 Button 0x54030000 0x4 168 116 48 14 "Cancel"
END DIALOG

if(!ShowDialog(dd 0 0)) ret
```

Exe cannot be decompiled to see the source code that is not added to exe as text, but it is possible to see strings that were in source code. You can encrypt (in Options -> Security) items that contain passwords or other confidential information in strings.

Other files

External files (e.g. dll) are not added to exe automatically. You can add them together with exe file to a zip file or setup program, and on other computers extract everything to the same folder. You can also add files to exe as resources, like function [ExeQmGridDll](#) does (you can see its code). Read more [below](#).

You may need these files. Take them from QM folder.

- qmgrid.dll. Required if exe uses QM [grid control \(119\)](#). To add it to exe, you can use function [ExeQmGridDll](#).
- qmtc32.dll and qmtc64.dll. Required if exe uses [WindowText](#) class.
- sqlite3.dll. Required if exe uses [Sqlite](#) class.
- mailbee.dll. Required if exe uses QM email functions ([SendMail](#), [ReceiveMail](#), [SmtMail](#), [Pop3Mail](#)).
- arservicesmgr.dll. Required if exe uses functions from Services type library.
- tcc folder. Required if exe uses [__Tcc](#) class (C compiler).

If exe runs on your computer, it finds these files in QM folder. You don't have to copy them to your exe folder. The same when it runs on other computers where QM is installed (should be same version). Particularly, if a dll whose [path \(153\)](#) begins with "\$qm\$" isn't in exe's folder, exe also searches in QM folder, if QM is installed.

QM 2.3.4. Although mailbee.dll and arservicesmgr.dll are COM components, don't need to register them.

Creating Setup program

If you want to create a setup program for your exe, you'll need a software that does it. It adds all used files (exe, dll files, Help file, etc) to a single compressed setup program, on user's computer runs setup wizard, creates folders, extracts files, creates shortcuts, creates uninstaller, and more.

Recommended free software - [Inno Setup](#). The QM setup program also is created with it. Download and install the QuickStart Pack unicode version. To start creating your setup script, use Example1.iss as an example.

Errors

Compile-time errors ("unknown identifier", etc) will not occur in exe, because exe contains already compiled code. Declarations, type libraries and other references are not used in exe (all required information is added to exe when creating it). However, like any other program, the program will fail if a required [dll function \(153\)](#) or COM component does not exist on the computer. To check whether a dll function exists, you can use [version variables \(144\)](#) and delay-loading (`(dll- ...)`). To handle "No such interface supported" errors you can use [err](#).

When an unhandled run-time error ("window not found", etc) occurs, the program exits. The error is shown in QM output or message box, depending on 'Output to QM' checkbox in 'Make Exe' dialog (see above). See also: [ExeOutputWindow](#), [RedirectQmOutput \(114\)](#).

Global variables and settings

If you use global variables that are initialized at QM startup (for example in your [init2](#) function), you should add the initialization code to the exe, or these variables in exe will be empty. For example, call the initialization function at the beginning of the main function (you can use `#if EXE` to skip it when running in QM). Don't need it for [predefined global variables \(144\)](#) and for global variables that are initialized by constructor. If you change global run-time settings with [RtOptions \(114\)](#), do it in exe too.

Single instance

If the main function has "Allow only single instance" option checked in Properties, only single instance of the program can run at a time. If it is a macro, macro properties "don't run/wait/terminate" are applied, which also means that only single instance of the program can run at a time. In both cases, limited is only the number of instances of the same program. Don't use "Allow only single instance" if you want to have more control, e.g. relay the command line to already running instance. You can use a [mutex](#) instead. Note that [getopt](#) nthreads will not work. [Mutex example](#)

at the beginning of exe main function:

```
SetLastError 0
Handle m=CreateMutex(0 1 "mutex_my_exe")
if(!m) mes- "Error"
if(GetLastError=ERROR_ALREADY_EXISTS) ;;another instance of your exe is running
possibly execute some code here
ret ;;exit exe
```

Command line arguments

If the program is launched with command line arguments, it is in a special variable [_command \(144\)](#). To parse, you can use function [ExeParseCommandLine](#).

Exit code

On success, it is the return value of the main function. It should be 0 or positive. On failure, it is: -1 if the main function ended due to a run-time error; -2 if the program didn't start because another instance is already running; -3 if terminated from outside (e.g. using hotkey); -4 if could not start (e.g., a dll function not found).

Threads

The main function runs not in the primary thread of the exe process. Like in QM, the primary thread is used to manage threads and provide other services. The main function is executed in a separate thread.

More threads can be created using [mac](#). The exe process ends when the main function's thread ends. To wait until other threads will end, the main thread can use [WaitForThreads](#).

Functions [OnScreenDisplay](#), [MsgBoxAsync](#), [Play](#) and [Speak](#) run asynchronously, in separate thread. As well as [mac](#). Your main thread can use [wait 0 H](#) or [WaitForThreads](#) to wait for them. Some of them have a flag to run synchronously.

51. Make exe

You can use `shutdown -6` to end a thread in the exe process. It will not end threads in QM and other processes. You can use `shutdown -1` to end the exe process from any thread.

Like in QM, constructors and destructors of global variables run in the primary thread.

Triggers

QM triggers cannot be used to launch exe (unless QM is running and you use its triggers).

To launch exe using a hotkey, can be used a shortcut on desktop or in start menu (you can assign a hotkey in shortcut Properties dialog). To avoid possible problems when Ctrl/Shift/Alt are mixed with keys pressed by exe, exe created from macro waits until these keys are released before starting to execute commands. Exe created from function does not, but you can insert this code at the beginning:

```
rep() if(GetMod) 0.02; else break
```

How to get exe file path

Use `_qmdir (144)` or `ExeFullPath`.

Exe file size

QM cannot create small exe files. Minimal size is about 400 KB.

Speed

Programs created with QM run at the same speed as macros and functions in QM.

QM does not create native machine code. In some cases QM code runs significantly slower than native machine code (e.g. C++ programs).

Can QM create dll files?

QM can create only exe files.

You can use class `__Tcc` to create dlls in C language.

Supported Windows versions

Programs created with QM can run on Windows XP, Vista, 7, 8, 10. Read about Vista/7/8/10 [below](#).

Resources

See also: [macro resources \(261\)](#). Note that exe resources is not the same as macro resources, but macro resources are automatically converted to exe resources when creating exe.

Resources are files and other data added to exe file. Can be added bitmaps (pictures), icons, cursors, menus, accelerators, strings, version info, binary data (e.g. files), etc. If you use files in exe macros/functions, you can either distribute them together with the exe (for example in a zip file) or add them to exe as resources.

If "Auto add files ..." is checked, QM searches the source code of all macros/functions that are added to exe (except as text) for strings that look like `":resourceid filepath"` and adds the files to exe as resources. For example, to automatically add bitmap file used with `scan`, use code like `scan ":10 file.bmp"`, and check "Auto add files ...". The table shows what resources are added depending on file type (filename extension).

File type	Resource type	How to load explicitly	How to free explicitly	Comments
bmp	Bitmap.	LoadPictureFile (114) . Returns bitmap handle.	DeleteObject	
ico	Icon.	GetFileIcon (114) . Returns icon handle.	DestroyIcon	
cur	QM 2.3.0. Cursor.	GetFileIcon (114) with flag 4. Returns cursor handle.	DestroyCursor	
exe, dll, ocx, icl	QM 2.3.0. Icon.	GetFileIcon (114) . Returns icon handle.	DestroyIcon	filepath must contain icon index or negative resource id. Examples: <code>":10 shell32.dll,15"</code> , <code>":10 shell32.dll,-140"</code> . Otherwise whole file will be added as RT_RCDATA.

51. Make exe

gif, jpg, png	QM 2.3.0. RT_RCDATA.	LoadPictureFile (114) . Loads as bitmap; returns bitmap handle. QM 2.3.4: supports png.	DeleteObject	Can be used in dialog static picture controls. Assign ":\resourceid filepath" to the variable before calling ShowDialog . QM 2.3.4: supports png.
ani	QM 2.4.1. RT_ANICURSOR.	GetFileIcon (114) with flag 4. Returns cursor handle.	DestroyCursor	In older QM - RT_RCDATA.
Other	QM 2.3.0. RT_RCDATA.	str.getfile (217) or ExeGetResourceData . Loads raw data.	Don't need.	

resourceid must be a number between 1 and 0xFFFF. Filename extension must consist of 1 - 4 alphanumeric characters. The string must not be in comments or #if-excluded code or F"string".

QM functions that support the ":\resourceid filepath" syntax: [ShowDialog](#) (title bar icon, icons, bitmaps, imagelists), [scan \(87\)](#) (bmp, icon), [GetFileIcon \(114\)](#) (icon, cursor), [LoadPictureFile \(114\)](#) (bmp, gif, jpg, png), [str.getfile \(217\)](#), [IXml.FromFile](#), [ICsv.FromFile](#), [CsScript.Load](#), [RichEditLoad](#), [bee \(96\)](#) (QM 2.3.4), [__ImageListLoad](#), [__ImageList.Load](#), [__ImageList.Create](#), [ShowDropDownList](#), and functions that use them.

QM 2.3.0. When using ":\resourceid filepath", QM always gets data from the file when the macro runs in QM. In previous versions, QM would try to get the data from exe if it exists.

QM 2.3.1. Supports multiline strings. [Example](#)

```
str multiline=
:206 $qm$\keyboard.ico
:207 $qm$\mouse.ico
```

Other way to add files to exe is [#exe addfile \(179\)](#).

To create resources of other types, you can use a resource editor. QM does not have it, but you can download one. Using the resource editor, add all that you need and save to a file with .res extension. Specify the file in the Make Exe dialog. When creating exe, QM will add resources from the file to the exe. To load these resources, you can use syntax ":\resourceid" (e.g. ":\300") with the functions listed above.

Whenever you edit files that you have added to exe, you should make exe again, otherwise its resources will not be updated.

Resources also can be used with Windows API functions ([LoadImage](#), [FindResource](#), and other). To get resource module handle, use [GetExeResHandle \(114\)](#). Example: `int hi=LoadImage(GetExeResHandle +133 IMAGE_ICON 0 0 0)`. Such code will run in QM as well as in exe.

You should not use dialog resources. You can create dialogs in QM. With dialog resources you could not use [ShowDialog](#) and all additional features that QM provides (control variables, etc). Menus also can be created in QM.

By default, exe contains these resources: icon 1 (if icon file specified), manifest 1 (if manifest file specified) and version info 1 (if version or other version info specified).

QM 2.3.0. Exe icon resource id changed from 133 to 1.

The <default> exe icon is the main function's icon. Icons of other QM items are not automatically added to exe. For example, in exe all dialogs will have the same icon (exe icon). In QM they may have different icons. To have different icons in exe, specify the icon files explicitly when calling [ShowDialog](#), using the ":\resourceid filepath" syntax.

Features that are unavailable in exe

Some functions and other QM features are unavailable in exe. Most of them are related to QM items and interface, and therefore have no sense in exe. Others use QM hooks, services, etc, that would cost too much in exe.

Unavailable functions:

- [deb](#), [dis](#), [newitem](#), [str.setmacro](#).
- Some [QM dll functions \(114\)](#).
- Functions from folder ":\System\Functions\Qm\unavailable in exe".

Unavailable only some features:

- [mac](#) can start only functions. Cannot open items (+).

- [qmitem](#) can be used only to get item id.
- [str.getmacro](#) can only get text and name.
- [wait](#) x KF (wait for key-down and eat).
- [_error.line](#) (will be empty).
- flag 3 with [end](#) (will behave as without flag 3).
- [shutdown](#) -2, -3, -4, -5 (will fail at run time).
- [str.encrypt](#) algorithm 16 (will fail at run time).
- [EnsureLoggedOn](#) cannot unlock (will always return 0 if locked).

QM detects unavailable functions when creating exe. Also detects unavailable features used with [mac](#), [qmitem](#), [str.getmacro](#), [wait](#).

A special constant [EXE \(144\)](#) can be used with [#if \(172\)](#) to include/exclude code that is not supported in exe. It is 1 if compiling .exe, 2 if .qmm, 0 if the macro runs in QM.

Other unavailable features:

- Menus, toolbars, autotext lists. Only macros, functions and member functions can be added to exe. Other items can be added only as text (to use with [str.getmacro](#), etc).
- QM user interface (editor, output, recording, tray icon, etc). To add tray icon, you can use [AddTrayIcon](#). To see output - [ExeOutputWindow](#) or [RedirectQmOutput \(114\)](#).
- Source code.
- Triggers.
- QM command line parameters.
- Sounds, as well as any other settings in the Options dialog.
- Automatic registration of COM components. If you distribute COM components with the program, you can use [RegisterComComponent \(114\)](#) to install them (admin privileges required). It is possible to [use COM components without registration \(163\)](#).
- Does not load rich edit dll, except when a rich edit control is used in a dialog or with [CreateControl](#). You can load it: [LoadLibrary\("Riched20"\)](#). Similar with [QM_Grid control \(119\)](#).
- Although [net](#) in exe can be used to run macros in QM, exe itself cannot be controlled. Ie exe works as TCP client but not as server.
- Cannot unlock locked computer (Properties -> Common -> Unlock ...).

Features that are different in exe

Some [predefined variables \(144\)](#) have different values when the macro runs in exe.

[Predefined constants \(144\)](#) (as well as all constants) have values specific to the computer where the exe was created, not where it runs.

Special folder \$qm\$ is exe file path, not qm.exe path.

Special folder \$my qm\$ probably does not exist on computers where the program will run.

Some functions are implemented differently in exe, and therefore may behave slightly differently in some cases. It includes:

- [run](#) flags 0x10000 and 0x20000 (run as admin) work differently on Vista/7/8/10.
- [web](#) may work differently on Vista/7/8/10.
- [key](#) may be slower and in some cases less reliable.

Running on Windows Vista/7/8/10

Windows Vista/7/8/10 has a security feature - *User Account Control (UAC)*. With UAC, even if you are working on an administrator user account, most programs have only standard user's privileges. It creates various problems for many programs, including QM. They cannot write to many file system and registry locations, manipulate services, and much more. Also, most of them cannot interact (use keyboard, mouse and menu commands, send messages, use hooks, etc) with programs that have higher privileges. QM can interact with all programs, but UAC may be a problem for QM-created programs. Read more about it [here \(277\)](#).

Since most programs on Vista/7/8/10 run with standard privileges, your QM-created programs will be able to automate them. If they have to automate programs with higher privileges (admin or uiAccess), or if they need higher privileges for other purposes, they must be started as administrator or marked as uiAccess.

To mark a program as administrator or uiAccess program, add a manifest resource with certain tags. uiAccess programs also must be signed. A code signing certificate costs 100-500 \$/year. The default manifest specifies normal privileges (level="asInvoker" uiAccess="false"). Other level values that can be specified are "highestAvailable" and

"requireAdministrator". uiAccess can be "false" or "true". You can find more information about UAC and MS Authenticode code signing on the Internet. Also you can ask about it in the Quick Macros [forum](#).

See also: [GetProcessUacInfo \(114\)](#), [Run in separate process](#)

Properties -> Run in separate process

Normally, macros run in QM process. If you check "Run in separate process" in Properties, the macro will run in its own process. This option is available for macros and functions.

When you launch such a macro or function, QM creates and executes a .qmm or .exe file. By default uses .qmm, but you can change file extension to .exe, and then will be created .exe file. You can change exe/qmm settings in the Make Exe dialog.

If the file already exists, QM checks macro and file times etc, and creates file again if need. If settings are stored in a folder, QM also checks all macros/functions in the folder. If the macro directly or indirectly uses functions from other folders, after editing these functions need to create file again (use the Make Exe dialog or edit/run the macro).

This feature can be useful when creating/debugging exe that is not to be run from QM. Instead of invoking menu Run -> Make Exe and Run -> Run Exe, you can just click the Run button. The exe file is exactly the same as if you create it using menu Run -> Make Exe. If you initially have .qmm file, change file extension to .exe.

UAC integrity level

The "Run in separate process" feature can be useful on Windows Vista/7/8/10, when some functions require certain privileges while QM is running with different privileges. There are several predefined sets of privileges - [integrity levels](#) (IL). Process IL is assigned when starting the process and cannot be changed later. QM itself can run as administrator (High IL), normal user (Medium IL) or uiAccess (Medium+uiAccess IL); you can set it in Options. Macros running in separate process can run as:

0	As QM	The same IL as of QM.
1	User	Medium IL, even if QM is running as administrator.
2	Administrator	High IL. On non-admin accounts also runs as admin but shows a password dialog.
3	Highest available	High IL on admin accounts. As QM on non-admin accounts.
4	Low	Low IL, like Internet Explorer running in protected mode.
5	As QM, -uiAccess	The same IL as of QM, but without uiAccess privileges.

To run exe files with various IL, QM uses function [StartProcess \(114\)](#). You can also use it in macros, except in exe. The first argument can be one of values from the table.

Integrity levels are used only on Vista/7/8/10, and only if UAC (User Account Control) is on. Otherwise, exe will run with same privileges as QM, regardless what IL is selected. The only exception is "Administrator", which also is applied on non-administrator accounts on other OS.

On non-administrator accounts, if IL is different than QM, exe may run as other user than QM. Possible problems with user-specific special folders ("\$personal\$, "\$my qm\$, etc) and registry keys. If exe does not run, try to change its path (in Make Exe dialog) from "\$my qm\$" to some common folder, for example "\$common appdata\$\qmexe".

Vista/7/8/10 IL that is specified in Properties is applied only if the exe is started by QM like a macro. It is not applied if the exe is started using [run](#) or when you double click exe icon in Windows Explorer. To apply it when you double click exe icon, check "Run in separate process", create shortcut to the macro, and use the shortcut.

There are some limitations in [portable QM \(259\)](#).

Notes

When you run the macro using [mac \(100\)](#), a command line can be passed to the exe process. The main exe thread receives the string through the _command variable. String and numeric arguments also can be passed (use [function](#)). Don't pass pointers and references.

```
mac "macro name" "command line" 5 "string arg"
```

To manage the exe process (display running macro icon, end macro on Pause, etc), QM creates a thread in QM. It is listed in the Running Items list and in the Threads dialog. If you end the thread (or end macro using Pause), it also ends the process. If you use [mac](#) as function (ie assign to a variable), it returns thread handle. The caller macro can wait until macro ends.

```
int h=mac("macro name")
wait 0 H h
```

Note: This does not work if both the macro and the caller are macros. At least one of them must be function.

If you use "On Run" function, it receives the command line and arguments through its first parameter. The `_command` variable contains exe file path. The function should run the exe file and wait until it exits (e.g., [run \(64\)](#) with flag 0x400, or [StartProcess \(114\)](#) and [wait \(89\)](#) for handle). If it does not wait, QM cannot manage the exe process. Also, it should exit soon after the process ends.

```
function# $ca
run _command ca "" "" 0x400
```

In some cases, the macro does not run in separate process, even if "Run in separate process" is checked: 1. If it is a function called from code. 2. If it is a function started from exe (using [mac](#)). 3. If it is a function started by a QM startup/exit trigger and is set to run synchronously. 4. If it is a function called by QM using some special way, e.g. like a filter function.

As described earlier in this topic, some QM features are unavailable or different when the macro runs in separate process. For example, global variables are not shared by processes.

.qmm files

A .qmm file contains compiled macro and related resources. Unlike .exe, it is not executable. The macro is executed by qmmacro.exe when you run the macro from QM as separate process. Executing not from QM also is possible, but not recommended, because .qmm file version must match qmmacro.exe version.

The .qmm file format is the same as of a resource-only dll. For example, you can add icons and use as an icon library.

Possible problems with antivirus software

1. Macros that run in separate process may start with a delay. Probably your antivirus program scans the .exe or .qmm file, and also qmmacro.exe that executes .qmm files.
2. Macros running in separate process may not run normally when using .exe (should not be problems with .qmm). Your antivirus program may show a warning. For an antivirus program, unknown (not commonly downloaded) and unsigned program files may look suspicious. For example, Avast puts them into its sandbox and terminates.

Antivirus programs usually allow you to exclude some files and folders from scanning/autosandboxing. Make exclusions for your exe folder (My QM) and QM folder. It should solve the above problems. Or change autosandbox settings, if it triggers too often for other programs too.

License

With Quick Macros you can create, use and distribute programs without restrictions.

Reference [Syntax \(44\)](#), [Programming \(45\)](#), [Top 20 \(2\)](#)

QM scripting language

Flow: [goto \(122\)](#), [if else \(123\)](#), [iif \(124\)](#), [sel case \(125\)](#), [rep \(126\)](#), [for break continue \(127\)](#), [foreach \(128\)](#), [err \(129\)](#), [ret \(130\)](#), [end \(131\)](#), [call \(132\)](#)

Operators: = + - * / % *Bitwise:* & | ~ ^ >> << *Logical:* and or *Compare:* = ! < <= > >= *Unary:* ! ~ - * + @ [priority \(135\)](#), [precision \(136\)](#)

Variables, constants: [def \(139\)](#), [declaration \(141\)](#), [scope/storage \(142\)](#), [predefined \(144\)](#), [OLE types \(145\)](#), [arrays \(146\)](#), [pointers \(147\)](#), [memory \(148\)](#)

Functions, types, libraries: [function \(152\)](#), [dll \(153\)](#), [type \(154\)](#), [unions \(156\)](#), [class \(157\)](#), [category \(159\)](#), [interface \(165\)](#), [typelib \(164\)](#), [COM \(162\)](#), [COM functions \(167\)](#), [ref \(160\)](#), [declarations \(242\)](#)

Directives: [#if #else #endif \(172\)](#), [#ifdef #ifndef \(173\)](#), [#compile \(174\)](#), [#set \(177\)](#), [#opt \(176\)](#), [#err \(175\)](#), [#out #warning #error \(178\)](#), [#exe \(179\)](#), [#ret \(181\)](#), [#sub \(182\)](#), [#region #endregion \(180\)](#)

Other: [comments \(95\)](#), [sizeof \(109\)](#), [uuidof \(110\)](#)

Functions

Mouse: [lef rig mid dou \(53\)](#), [mou \(54\)](#), [xm ym \(55\)](#)

Keys, text: [key \(56\)](#), [paste \(58\)](#), [out \(57\)](#), [ifk \(59\)](#)

Dialogs: [mes \(62\)](#), [inp \(60\)](#), [inpp \(61\)](#)

Files: [run \(64\)](#), [cop ren \(65\)](#), [del \(66\)](#), [mkdir \(68\)](#), [zip \(71\)](#)

Window: [act \(72\)](#), [clo \(73\)](#), [min max res \(74\)](#), [mov siz \(75\)](#), [hid \(76\)](#), [ifa \(78\)](#), [win wintest \(77\)](#)

Controls: [but \(81\)](#), [id \(82\)](#), [child childtest \(83\)](#), [acc acctest \(85\)](#), [htm \(86\)](#), [men \(80\)](#), [scan \(87\)](#)

Time: [wait \(88\)](#), [wait for \(89\)](#), [spe \(90\)](#), [perf \(91\)](#), [tim \(92\)](#), [DATE \(93\)](#)

Internet: [web \(94\)](#), [run \(64\)](#), [htm \(86\)](#)

QM: [mac \(100\)](#), [dis \(102\)](#), [qmitem \(107\)](#), [newitem \(108\)](#), [net \(101\)](#), [atend \(103\)](#), [opt \(97\)](#), [getopt \(98\)](#), [lock \(112\)](#), [deb \(99\)](#)

Other: [bee \(96\)](#), [shutdown \(104\)](#), [pixel \(105\)](#), [scan \(87\)](#), [rset rget \(106\)](#), [string map \(115\)](#), [XML \(117\)](#), [CSV \(116\)](#), [QM dll functions and controls \(114\)](#), [dialogs \(63\)](#)

String functions (183)

Global: [len \(184\)](#), [empty \(185\)](#), [val \(186\)](#), [numlines \(187\)](#), [find \(188\)](#), [findw \(189\)](#), [findt \(190\)](#), [findl \(191\)](#), [tok \(192\)](#), [findc findcr findcs findcn \(194\)](#), [findb \(195\)](#), [findrx \(197\)](#), [matchw \(196\)](#)

str:

Convert: [lcase ucase \(226\)](#), [unicode ansi \(238\)](#), [escape \(210\)](#), [encrypt decrypt \(209\)](#)

Modify: [trim ltrim rtrim \(237\)](#), [set \(232\)](#), [insert \(225\)](#), [remove \(228\)](#), [replace \(229\)](#), [findreplace \(211\)](#), [replacerx \(230\)](#), [addline \(204\)](#)

Get: [left right get geta \(227\)](#), [gett \(223\)](#), [getl \(218\)](#), [getpath getfilename \(221\)](#)

Format: [from \(214\)](#), [fromn \(215\)](#), [format formata \(213\)](#)

Compare: [= ~ ! \(133\)](#), [beg begi end endi mid midi \(206\)](#)

Other: [all \(205\)](#), [fix \(212\)](#), [getwintext setwintext getwinclass getwinexe \(224\)](#), [getclip setclip getsel setsel \(216\)](#), [getfile setfile \(217\)](#), [searchpath expandpath \(231\)](#), [dospath \(208\)](#), [getmacro \(219\)](#), [setmacro \(220\)](#), [dllerror \(207\)](#), [timeformat \(236\)](#), [getstruct setstruct \(222\)](#), [swap \(233\)](#)

QM extensions, dll functions, command-line tools

This Help file includes only functions implemented in qm.exe and its dlls.

Many other functions and classes are implemented using QM scripting language. Their code is in QM System folder and [forum](#). Their Help is displayed in QM window when you click a function name or class name in the code editor and press F1. To search all, use the ["Find help, functions, tools" \(4\)](#) field in QM window. Also use [floating toolbar dialogs](#), [function lists](#), [status bar info](#), [F1 \(46\)](#), [categories \(113\)](#).

Also you can use external functions - Windows API, COM components, other dlls, other languages. See [MSDN Library \(256\)](#), [API references \(160\)](#), [COM type libraries \(164\)](#), [math functions \(120\)](#), the [script](#) category.

Also you can execute command-line programs (Windows and other) with [run \(64\)](#) or [RunConsole2](#).

Terms, tables

[statement \(44\)](#), [expression \(244\)](#), [window expression \(274\)](#), [child, top-level, pop-up, client area \(275\)](#), [thread \(49\)](#), [flags \(247\)](#), [trigger coding \(264\)](#), [QM key codes \(251\)](#), [virtual-key codes \(270\)](#), [character codes \(239\)](#), [Windows keyboard shortcuts \(276\)](#), [special folders \(246\)](#), [format fields \(248\)](#), [regular expression \(198\)](#), [errors \(48\)](#), [declarations \(242\)](#)

Mouse click

Syntax

```
lef [+|-] [x y] [window] [flags]
rig [+|-] [x y] [window] [flags]
mid [+|-] [x y] [window] [flags]
dou [x y] [window] [flags]
```

Parameters

x y - mouse pointer [coordinates \(241\)](#). If omitted, mouse pointer is not moved.

[\(274\)](#)**window** - top-level or [child \(275\)](#) window. If omitted or literal 0, coordinates are relative to the top-left corner of the screen.

flags (247):

1	Coordinates are relative to the top-left corner of window's client area. If window is literal 0 - to the work area.
2	Don't activate window . Not error if point x y does not belong to window or its top-level parent window.
4	QM 2.3.0. Finally return the mouse pointer to previous position.

Options:

+	button down (press).
-	button up (release).

Remarks

lef clicks the left mouse button.

rig clicks the right mouse button.

mid clicks the middle mouse button.

dou double clicks the left mouse button.

When **window** is used:

- Restores minimized window.
- Activates inactive window (depends on **flags** and window style).
- Error if the point belongs to another top-level window (depends on **flags**).

The speed depends on [spe \(90\)](#) and [opt slowmouse \(97\)](#).

Tips

You can use function [BlockInput](#) to block keyboard and mouse input while macro is running. Place [BlockInput 1](#) at the beginning. If you want to manually end macro when input is blocked, at first press Ctrl+Alt+Delete.

See also: [MouseButtonX](#) and other functions in the mouse [category \(159\)](#).

Examples

```
lef ;;mouse left button click
lef+ ;;mouse left button down
lef- ;;mouse left button up
lef 100 200 ;;mouse left button click at 100, 200 pixels of the screen
lef 50 30 "Notepad" ;;mouse left button click at 50, 30 pixels of "Notepad" window
lef 10 10 id(131 "Calc") ;;mouse left button click at 10, 10 pixels of button with id=131 in "Calc" window
lef 0.5 0.5 "Notepad" ;;mouse left button click at middle of "Notepad" window
```

Click in "Notepad" window, x=100 (relative to "Notepad"), don't change y:

```
int w = win("Notepad")
lef 100 ym(0 w) w
```

Mouse move

Syntax1 - absolute movement

```
mou [x y] [window] [flags]
```

Syntax2 - relative movement

```
mou (+|-) x y
```

Syntax3 - relative movements, used for compact recording

```
mou "recorded unreadable text"
```

Syntax4 - restore mouse position

```
mou
```

Parameters

x y - mouse pointer [coordinates \(241\)](#).

[\(274\)window](#) - top-level or child window. If omitted, coordinates are relative to the top-left corner of the screen.

flags: 1 - coordinates are relative to the top-left corner of window's client area. If **window** is literal 0, the work area. Default: 0.

Options:

+	coordinates are relative to the mouse pointer position.
-	coordinates are relative to the previous mouse movement/click command coordinates.

Remarks

Syntax4: Restores cursor position as it was before first mouse movement/click command in current macro.

The speed depends on [spe \(90\)](#) and [opt slowmouse \(97\)](#).

Examples

```
mou 500 300 ;;move the mouse pointer to 500, 300 pixels of the screen
mou 100 150 "Notepad" ;;move the mouse pointer to 100, 150 pixels of "Notepad" window
mou+ 20 -10 ;;move the mouse pointer 20, -10 pixels from current position
mou ;;restore mouse pointer position as it was before first mouse movement/click command in current macro
```

Move the mouse pointer in "Notepad" window, x=100 (relative to "Notepad"), don't change y:

```
int w = win("Notepad")
mou 100 ym(0 w) w
```

Get mouse pointer position, ..., restore mouse pointer position:

```
int px(xm) py(ym)
...
mou px py
```

Get cursor (mouse pointer) position

Syntax

```
int xm([pp] [window] [client])
int ym([pp] [window] [client])
```

Parameters

pp - variable of type POINT. If used, each function also stores x and y coordinates into this variable. Default: 0.
(274)window - top-level or child window. If omitted or literal 0, coordinates are relative to the top-left corner of the screen.
client - if 1, coordinates are relative to the top-left corner of window's client area or, if **window** is literal 0, the work area. Default: 0.

Remarks

Function **xm** returns cursor x position in [pixels. \(241\)](#) **ym** - y position. To get x and y in one call, use **pp** with either function.

The POINT type is used to specify coordinates.

```
type POINT x y
```

Examples

```
int x = xm

lef 100 ym(0 "Notepad") "Notepad"

POINT p
xm p "Quick"
out "Cursor coordinates relative to window 'Quick' are:[]x=%i y=%i" p.x p.y
```

Send keys and text

Syntax1

`key` [+|-] keys

Syntax2

'keys

Remarks

Generates (synthesizes, simulates) keyboard events. It works like you would press keyboard keys manually. The key events will be sent to the focused window or to the application that uses them as a global hotkey.

To create code for `key`, you can use dialogs 'Text' and 'Keys'.

keys can include any number of parts of these types:

- Parts without quotes and parentheses. Sends keys specified using [QM key codes \(251\)](#).
 - Use uppercase letters for non-text keys, such as Ctrl, Enter, F2, and numeric keypad keys.
 - Example: `key TT Y F2 N3 ;;Tab Tab Enter F2 Num3.`
 - Use lowercase letters and other characters for text keys.
 - Example: `key a/2 ;;A / 2.`
 - Characters ; " { } () have special meaning. Instead use : ' [] 90 or enclose in double quotes.
 - Add modifier keys (Ctrl, Shift, Alt, Win) before the modified key.
 - Example: `key Wf CSF2 ;;Win+F Ctrl+Shift+F2.`
 - If several keys, enclose in { }. Example: `key A{ev} ;;Alt+E+V.`
- Text in double quotes. Sends keys that type the text.
 - Example: `key "Text" ;;types Text (sends Shift+T E X T).`
 - Can be used [escape sequences \(137\)](#), like in all strings. Example: `key "a[] [9]b" ;;A Enter Tab B.`
 - Can be [F-string \(138\)](#). Example: `int x=5; key F"x={x}" ;;x=5.`
 - The key messages received by the target window depend on [opt keychar \(97\)](#).
- String expression (e.g. variable) in parentheses. Types text, like in case 2.
 - Example: `str s="Text[]more text"; key (s).`
 - To type a non-string variable, convert it to string. Example: `int x=5; str s=x; key (s).` Or use F-string, see above.
- Windows [virtual-key code \(270\)](#) in parentheses. Sends the key.
 - Example: `key (VK_VOLUME_UP).`
 - Example: `int vk='A'; key (vk).`
- Scan code in parentheses. Sends the key.
 - Use flag 0x10000. If it is an extended key, add flag 0x20000.
 - Example: `key (0x30000|0x38) ;;Right Alt (scan code 0x38, extended key).`
- Unicode character code in parentheses. Types the character.
 - Use flag 0x40000.
 - Example: `key (0x3A3|0x40000) ;;Σ.`
 - In [Unicode \(267\)](#) mode you can instead simply use Unicode characters in text enclosed in double quotes (case 2).
- Double (floating-point) expression in parentheses inserts a delay.
 - Example: `key a (1.0) b ;;A, wait 1 s, B.`
 - Example: `double d=0.5; key a (d) b.`
- (QM 2.2.1). Flags in parentheses.
 - 0x01000000 - same as option +.
 - 0x02000000 - same as option -.
 - 0x03000000 - removes the above flags and options + -.

9 (QM 2.3.3). Key repeat count in parentheses, with # prefix.

- Example: `key T (#10) ;;press Tab 10 times.`
- Example: `int i=5; key a (#i) ;;press A i times.`

Options:

+	press keys down but don't release.
-	release keys, don't press.

Options and flags 0x01000000 and 0x02000000 can be used in case 1, 4 and 5.

Syntax2 can be used to make macro smaller.

The speed depends on [spe \(90\)](#) and [opt slowkeys \(97\)](#).

QM 2.2.1: `key` can be used as function. Then it does not send keys, but instead returns virtual-key codes. [Read more](#)

If used as function, `key` returns `ARRAY(KEYEVENT)`. Definition of `KEYEVENT`:

```
KEYEVENT !vk !flags @sc [0]wt
```

vk - virtual-key code.

flags - `KEYEVENTF_x` flags. Also can be 0x80, which says that next element contains wait time (see below).

sc - scan code, or UNICODE character, depending on flags.

wt - wait time in milliseconds. A delay is specified using two elements. First element contains nothing more than flag 0x80. Next element contains nothing more than wait time. Note that **wt** is in [union \(156\)](#) with other members.

All this info (except delay) is prepared to use with [SendInput \(256\)](#). Also can be used with some other Windows API functions, for example `keybd_event`, `PostMessage(WM_KEYDOWN/WM_KEYUP)`.

It is convenient to use `key` as an argument of a function that accepts `ARRAY(KEYEVENT)`. Example:

```
KeyPostToControl key(abc) id(15 "Notepad")
```

Function `KeyPostToControl`:

```
/
function ARRAY(KEYEVENT) 'a hwnd
...

```

Tips

You can use function [BlockInput](#) to block keyboard and mouse input while macro is running. If you want to manually end macro when input is blocked, at first press Ctrl+Alt+Delete.

When not using [low level keyboard hook \(13\)](#), if the user presses keys simultaneously with the `key` command, user-pressed keys may be inserted between `key`-sent keys. To avoid it, use [opt keymark 1 \(97\)](#).

See also: [Windows keyboard shortcuts \(276\)](#), [opt slowkeys/keysync/hungwindow \(97\)](#), [escaping special characters \(137\)](#), [admin apps on Vista/7/8/10 \(277\)](#)

Examples

```
key ab c F12 T Y      ;;A, B, C, F12, Tab, Enter
key Ca                ;;Ctrl+A
key CSf A{ec} Wd      ;;Ctrl+Shift+F, Alt+E+C, Win+D
key "Send Keys"       ;;type "Send Keys"
key SsendVSkeys        ;;type "Send Keys" using QM key codes
key (s)               ;;type string variable s
key F"{x}"            ;;type non-string variable x
key (VK_F2)            ;;F2 (use virtual-key code constant)
key A(44)              ;;Alt+PrintScreen (use virtual-key code)
key a (0.5) b          ;;A, wait 0.5 s, B
key Skeys "Keys" A(44) AT (0.5) Cv A{fa} Y
```

'AT

;;Alt+Tab (same as key AT)

Ctrl+Shift+click:

key+ CS

lef

key- SC

Display text and variables in QM output

Syntax1

```
out expression
```

Syntax2

```
out formatstring ...
```

Syntax3 - clear output

```
out
```

Parameters

expression - any expression (string, number, variable, etc).

formatstring - string with [format fields \(248\)](#), as with [str.format \(213\)](#). Cannot be variable.

... - variables or other values that will replace format fields in **formatstring**. Must be of [intrinsic types \(140\)](#).

Remarks

Sends text to QM output. It is useful when [debugging \(47\)](#) macros, learning/testing various functions, or to show information to the user when it is more appropriate than message box etc.

Mostly used format fields: %i integer, %s string, %1.6f double, %c character, %x hex int, %l64i long.

QM 2.3.0. Supports [tags \(245\)](#) for colors, links, etc. To display text with tags, the text must begin with "<>".

See also: [variables in strings \(138\)](#), [escaping special characters \(137\)](#), [mes \(62\)](#) (message box), [ShowText](#) (text box), [OnScreenDisplay](#), [RedirectQmOutput \(114\)](#), [ExeOutputWindow](#)

Examples

simple text:

```
out "text"
```

2 lines:

```
out "Line1[]Line2"
```

variable a:

```
int a = 10
```

```
out a
```

```
out a + b ;;sum of variables a and b
```

Call function len and display its return value:

```
lpstr s = "String"
```

```
out len(s)
```

variable i in decimal and j in hexadecimal:

```
out "%i 0x%X" i j
```

same as above

```
out F"{i} 0x{j}"
```

Paste text

Syntax1

`paste` *[+]* expression

Syntax2

`paste` *[+]* formatstring ...

Syntax3 - the paste keyword is optional

"text"
"formatstring" ...

Parameters

expression - any expression (string, number, variable, etc).

formatstring - string with [format fields \(248\)](#), as with [str.format \(213\)](#). Cannot be variable.

... - variables or other values that will replace format fields in **formatstring**. Must be of [intrinsic types \(140\)](#).

Options:

Default	Paste through clipboard.
+	QM 2.3.3. Hybrid paste. If text is short, instead sends keys. Read more below.

Remarks

The `paste` keyword added in QM 2.3.3. It is the same as `outp`, which can be used for backward compatibility.

`paste` stores text into the clipboard and sends keys Ctrl+V.

`paste+` (hybrid paste) uses clipboard only if text length is >100 or contains new lines. Else instead sends keys, like `key "text"`. Sends keys fast, as if with [opt \(97\)](#) `keysync 1`, `keychar 1`, `keymark 1`, `slowkeys 0`.

In Options -> General you can set to use hybrid paste in menu, toolbar and autotext list items where `paste` is omitted (syntax3), like `: "text"`.

If **expression** or **formatstring** is simple text in double quotes, keyword `paste` can be omitted (syntax3).

Restores previous clipboard content (text only), unless [run-time option \(97\)](#) `opt clip 1` is set.

In console windows uses different method to paste, because Ctrl+V does not work.

The speed depends on [spe \(90\)](#).

See also: [str.setsel](#), [str.getsel](#), [str.getclip](#), [str.setclip \(216\)](#), [key \(56\)](#), [variables in strings \(138\)](#), [escaping special characters \(137\)](#), [admin apps on Vista/7/8/10 \(277\)](#)

Examples

```
paste "text" ;;paste simple text
"text" ;;the same
paste x ;;paste variable x
paste "%i[]" x ;;paste int variable x and new line
paste F"{x}[]" ;;paste variable x and new line
```

If key is pressed

Syntax

```
ifk[-] keycode [toggled]
(tab) statements
(tab) ...
[else
(tab) statements
(tab) ...]
```

Can be single line:

```
ifk[-] (keycode [toggled]) statements
[else statements]
```

Parameters

keycode - [QM key code \(251\)](#).

- Also can be [virtual-key code \(270\)](#) enclosed in ().
- Also can be mouse button in (): (1) left, (2) right, (4) middle, (5) X1, (6) X2.
- Can be two keys. Then checks if both are pressed.
- For comma use <, not ,.

toggled - if nonzero, checks if toggled.

Options:

-	not.
---	------

Remarks

Similar to [if \(123\)](#).

If the specified key is pressed, executes **statements** after **ifk** and skips **statements** after **else** (if used). Else skips **statements** after **ifk** and executes **statements** after **else** (if used).

Not all keys and key combinations give correct result. For example, on some operating systems you cannot test the Pause key.

On [Vista/7/8/10 \(277\)](#), if the active window has higher integrity level (eg QM - standard user, window - administrator), **ifk** works only with some keys: modifier keys (Ctrl, Shift, Alt, Win), lock keys (CapsLock, NumLock, ScrollLock), Back, Tab, Enter, Esc, and mouse buttons.

On Vista/7/8/10, **ifk** can reliably check toggled state only for lock keys. The toggled state for other keys is process-specific or thread-specific.

Internally **ifk** uses QM function [RealGetKeyState \(114\)](#). You can use it instead of **ifk**.

QM 2.3.3. Can be used as function. Returns 1 if pressed, 0 if not.

See also: [GetMod \(114\)](#)

Examples

```
ifk(F2) bee ;;if key F2 pressed, beep
ifk(K 1) key K ;;if key CapsLock toggled, press CapsLock
ifk((1)) bee ;;if left mouse button pressed, beep
```

Repeatedly execute some code; stop when key F12 is pressed

```
rep
...
ifk(F12) break
```

RealGetKeyState example

```
if RealGetKeyState(VK_SHIFT) and RealGetKeyState(VK_LBUTTON)
```

```
_out "Pressed Shift key and mouse left button"
```

Can be used as function

```
if(ifk(C) and !ifk(S) and ifk(J 1)) out "Ctrl pressed, Shift not, ScrollLock toggled"
```

Input box

Syntax

```
int inp[-](var [text] [caption] [default] [checkvar] [checktext] [func] [hwndowner])
```

Parameters

var - receives input. Variable of intrinsic non-pointer type except lpstr.

text - text above the edit field. String. Default: "" (empty). Supports [links \(254\)](#).

caption - input box title. String. Default: "" ("QM Input").

default - edit control text. String. Default: "" (empty).

checkvar - integer variable. Sets and receives checkbox state (0 or 1). Default: 0 (hide checkbox).

checktext - checkbox text. String. Default: "".

func - name of and user-defined function that is called when text changes. The function must begin with [function#str&s](#). Here **s** is text of the input box edit field. The function can modify it. The function should return 0. Or, it can return 1 to press OK or 2 to press Cancel. Default: literal 0.

hwndowner (QM 2.2.1) - owner window handle. The dialog box will be on top of the owner window. The owner window, if belongs to the same thread, will be disabled. Default: 0.

Options:

-	on 'Cancel' end macro.
---	------------------------

Remarks

Displays input dialog box. If user selects OK, populates **var** with input value and returns 1. Else returns 0.

If **default** begins with "", text is hidden (use for password). Else, if **default** begins with "\n" (new line), text can be multiline.

See also: [InputBox](#).

Tips: If need more input fields, instead create a [dialog \(63\)](#).

Examples

```
int i
inp i

str p
int rem = 1
inp- p "Password" "" "*" rem "Remember"
if(p = "p11") ret rem
else mes- "x" "Password incorrect" ""
```

Password dialog

Syntax1

```
inpp password [text] [caption] [flags] [hwndowner]
```

Syntax2

```
int inpp(password [text] [caption] [flags] [hwndowner])
```

Parameters

password - user must enter this password. Can be encrypted (in Options -> Security, use "inpp" as function name).

text - text above password field. Default: "" ("Password:"). Supports [links \(254\)](#).

caption - dialog box title. Default: "" ("QM - Password").

flags (247):

1	password is case insensitive.
---	-------------------------------

hwndowner (QM 2.2.1) - owner window handle. The dialog box will be on top of the owner window. The owner window, if belongs to the same thread, will be disabled.

Remarks

Shows password input dialog box. If entered password is incorrect, or pressed Cancel, ends macro.

If used as function (syntax2), does not end macro, but returns 1 if password is correct, or 0 if not.

Tips: You can also encrypt macro in Options -> Security.

Examples

Ask for password, and end macro if entered password is not "55hH7pKJ":

```
inpp "55hH7pKJ"
```

Ask for password, and throw error if entered password is not "bnfg" (in code it is encrypted):

```
if(!inpp("[*E12073E7509E09F804*]" "" "" 1)) end "password incorrect"
```

Message box

Syntax

```
int mes [-] (text [caption] [style])
```

Parameters

text - message text. Can be string or numeric. Use [F-string \(138\)](#) to insert variables. Supports [links \(254\)](#).

caption - message box title. Default: "" ("QM Message").

style - message box style. Read in Remarks. Default: "" (OK button only).

Options:

-	on 'Cancel' and 'No' end macro. Also, ends macro if there is only OK button.
---	--

Remarks

Shows standard message box dialog. Returns the first character of selected button's name in English ('O' for OK, 'Y' for Yes, etc).

style can be string consisting of one or more characters or substrings. Syntax:

```
" [O|OC|YN|YNC|ARI|RC|CTE] [1|2|3] [?!|x|i|q|v] [s] [a|n] [t] "
```

O, OC, YN, YNC, ARI, RC, CTE - buttons. Default: OK.

O	OK
OC	OK Cancel
YN	Yes No
YNC	Yes No Cancel
ARI	Abort Retry Ignore
RC	Retry Cancel
CTE	Cancel Try-Again Continue

1, 2, 3 - default button. Default: 1.

?, !, x, i, q, v - icon. Default: no icon.

?	question (question-mark icon).
!	warning (exclamation-point icon).
x	error (stop-sign icon).
i	information (i in a circle icon).
q	QM icon. In exe - exe icon.
v	On Vista and later - shield icon. On older Windows - silent warning icon (same as !s). Added in QM 2.2.0.

a - always activate message box window.

n - prevent activating message box window. See also [RtOptions \(114\)](#).

t - topmost (always on top of other windows). Use only if owner window is specified; else message box is topmost by default.

s - silent. Don't play sounds.

style also can be an integer - MB_x flags used with [MessageBox \(256\)](#).

style also can be a variable of type MES.

```
type MES ~style x y timeout default hwndowner
```

style - message box style. Same as **style** parameter of [mes](#).

x and **y** - message box coordinates. Changed are only if nonzero. If negative - relative to the screen right and/or top edge.

timeout - max time (seconds) to show the message box.

default - button to choose (e.g., 'C'), or some other value to return on timeout. Values 1-15 are mapped to button characters that match Windows API standard button constants, eg IDOK to 'O', IDCANCEL to 'C'.

hwndowner - owner window handle. The owner window, if belongs to the same thread, will be disabled.

You can set only members that you need.

Examples

```
mes- "Click OK button"

mes F"Missing file {path}." "Error" "x"

if(mes("Save?" "" "YN?") != 'Y') ret

MES m
m.style="YNCn"
m.x=100
m.y=1
m.timeout=15
m.default='C'
int i=mes("message" "title" m)
```

Dialogs

[How to create dialog](#)
[Working with the Dialog Editor](#)
[How to show dialog](#)
[Dialog types \(modal, modeless, child\)](#)
[How to get data from controls](#)
[How to initialize controls](#)
[Variable names](#)
[Using a user-defined type](#)
[Smart dialogs](#)
[Multi-page dialogs](#)
[Multiple dialogs in a macro](#)
[Unicode](#)
[Fonts and colors](#)
[Menus and accelerators \(hotkeys\)](#)
[Common controls, other controls](#)
[ActiveX controls](#)
[Encrypted macros](#)
[Dialogs now and before QM 2.1.9](#)
[Dialog examples and tutorials](#)
[Dialog programming info at MSDN](#)

How to create dialog

To create new dialog, use menu File -> New -> New Dialog. It creates initial dialog definition (BEGIN DIALOG ... END DIALOG) and opens it in the Dialog Editor. You can add controls, order them, change text, etc. When you click Save, it updates the dialog definition.

To edit a dialog later, open the macro that contains the dialog definition and click 'Dialog Editor' button or menu item in QM window. If current macro does not contain a dialog definition, this button/menu creates new dialog.

You can also edit dialog definitions directly. The format is simple: The first line defines dialog window, other lines - controls. Line format: id class style exstyle x y width height text tooltip. Text and tooltip are optional. The position/size is in dialog units, not in pixels.

Working with the Dialog Editor

Add control	Click or drag&drop a control from the toolbox.
Move control	Drag with the mouse left button. Or use arrow keys.
Resize control or dialog	Drag with the mouse right button. Or use Shift+arrow keys.
Select control	Click it. A small red rectangle will appear at the top-left corner. Also you can use Tab or Shift+Tab to select next or previous control in the Z order.
Send control to the top of the Z order and tab order	Shift+click.
Send control to the bottom of the Z order	Ctrl+click.
Place control behind the currently selected control in the Z order	Ctrl+Shift+click.
Z-order all controls	Select dialog, then Ctrl+Shift+click all controls starting from the control that must be at the top of the Z order. Alternatively, close the Dialog Editor and reorder lines in the dialog definition. The order of lines matches the Z order.
Underline character in control's text	Place ampersand (&) before.
Create group of option buttons	Add "Option first" and then several "Option next".
Move multiple controls	If the controls are in a Group control, Shift+drag it. Else create/drag a temporary Group control for it.
Move/resize with high precision.	Use Alt when you drag or right-drag.
Add control similar to a	Move the mouse over the control and press the hotkey defined in Dialog Editor Options.

In the Dialog Editor, you can set text for some controls. For other controls, text is used only as control's name, making it easier for you to recognize the control in the Dialog Editor. The first three characters also are used for variable name. Generally, controls are identified by an unique control id - numeric value that you can see in variable name. For example, if variable name is b3But, you know that control id is 3.

How to show dialog

Use function [ShowDialog](#). The Dialog Editor creates code for it. When you click Save, it updates the code or displays in output, depending on dialog settings. The code also may contain declarations of variables for controls (read below). By default the code must be in the same function as the dialog definition, but it can be anywhere if not using a sub-function dialog procedure.

The minimum code to show the dialog is created by the Dialog Editor. But [ShowDialog](#) has more parameters that you can use.

```
function# [$macro] [dlgproc] [!*controls] [hwndowner] [flags] [style] [notstyle] [param]
[x] [y] [$icon] [$menu] ;;flags: 1 modeless, 4 set style (default is to add), 64 raw x y.
```

macro - where to look for dialog definition. Can be:

- A variable containing dialog definition. This is recommended.
- Name of macro (or function etc) that contains dialog definition.
- If "", [ShowDialog](#) looks for dialog definition in the caller macro. If it is a [sub-function \(182\)](#), looks in whole text, not just in that sub-function.

dlgproc - address of a dialog procedure. Optional. Used with smart dialogs (read below).

controls - array of strings (address of first str variable in the array) for controls, as explained later in this topic.

hwndowner - owner window handle. If the dialog has WS_CHILD style - parent window handle.

flags (247):

1	Modeless dialog.
4	Replace dialog style with style . Default - add style .
64	Raw x y . Read more below.
128	QM 2.3.3. Start hidden. <ul style="list-style-type: none"> • If later need to unhide, use act (shows and activates) or hid- (does not activate). • It also can be used to create visible dialog without activating it. Create hidden dialog, and under case WM_INITDIALOG call hid- hDlg. • When this flag used, does not disable owner window (hwndowner). • Without this flag, modal dialogs always start visible, regardless of WS_VISIBLE style.
0x100	QM 2.4.2. Use x y specified in dialog definition. Read more below. Also you should remove DS_CENTER style.

style - combination of window styles (WS_ and/or DS_ constants). Changes style that is set in the Dialog Editor.

notstyle - combination of styles to remove from default style.

param - some value to pass to the dialog procedure. The dialog procedure can retrieve it using [DT_GetParam](#) function.

x, y - dialog coordinates. Read more below.

icon - title bar icon.

- If omitted or "", uses default icon. In QM 2.3.0 and later it is icon of thread main function or macro. In older versions - default macro icon. In exe - in all QM versions it is exe icon.
- Supports [macro resources \(261\)](#) (QM 2.4.1) and [exe \(51\)](#) resources.

menu - menu.

- Can be name of QM item that contains menu definition (read below), or menu definition itself (variable). In [exe \(51\)](#) also can be menu resource id, like ":1".
- Menu definition can contain accelerators (hotkeys). If using resource, uses accelerator table with the same id as the menu (if exists).

Actual dialog coordinates depend on **x, y**, **hwndowner**, some styles, flag 64, and [_monitor \(144\)](#). If **x** and **y** are omitted or 0, by default dialogs are shown in screen center.

- If flag 0x100 used, **x y** are ignored. Instead are used x y specified in dialog definition (dialog units, not pixels) and styles DS_CENTER, DS_CENTERMOUSE, DS_ABSALIGN.
- Else if WS_CHILD style used, uses raw **x** and **y** in parent's client area.
- Else if flag 64 used, uses raw **x** and **y** relative to the primary monitor. If DS_CENTER style used (it is default), ensures

that the dialog is completely in the work area.

- Else if DS_CENTERMOUSE style used, the dialog is placed by the mouse pointer. Nonzero **x y** can be used to add offset.
- Else if DS_CENTER style used (it is default), negative **x** and/or **y** values are interpreted as offset from the right and/or bottom edge of the work area, 0 values - screen center. Positive coordinates are interpreted as coordinates in the work area of the monitor. The monitor depends on the `_monitor` variable.
- Else if **hwndowner** used and DS_ABSALIGN style not used, uses raw **x** and **y**, relative to the owner's client area.
- Else uses raw **x** and **y** relative to the work area of the monitor. The monitor depends on the `_monitor` variable.

If your dialog shows message boxes or other unowned dialogs, it should do it in the same monitor. Under `case WM_INITDIALOG` insert `_monitor=hDlg`. Without this, these unowned dialogs will be in the primary monitor regardless where your dialog is.

The return value depends on dialog type. Read the next chapter. Error if dialog cannot be shown, for example if a control class is not registered.

Dialog types (modal, modeless, child)

A dialog can be modal or modeless. If modal, `ShowDialog` returns only when the dialog is closed. If modeless, `ShowDialog` creates the dialog and returns immediately.

The Dialog Editor creates code to show modal dialog, but you can edit it to show modeless dialog (add flag 1, etc).

`ShowDialog` for a modal dialog returns 1 on OK, 0 on Cancel. When you click some other push-button in a dialog that does not have dialog function (that is, not a "smart" dialog), it closes the dialog, and `ShowDialog` returns button id.

`ShowDialog` for a modeless dialog returns dialog window handle. It does not wait until the dialog is closed, and does not populate dialog variables on OK. To make a modeless dialog useful, use dialog procedure (see [smart dialogs](#)). The thread must stay running and process messages. [Example](#).

```
\Dialog_Editor

str controls = "3 4"
str c3Che e4
int hDlg=ShowDialog("" &sub.DlgProc &controls 0 1)
MessageLoop 2 ;;flag 2 enables keyboard navigation, which by default does not work in modeless dialogs.
out c3Che
out e4

BEGIN DIALOG
0 "" 0x90C80AC8 0x0 0 0 223 135 "Dialog"
1 Button 0x54030001 0x4 120 116 48 14 "OK"
2 Button 0x54030000 0x4 170 116 48 14 "Cancel"
3 Button 0x54012003 0x0 8 10 48 12 "Check"
4 Edit 0x54030080 0x200 8 24 96 14 ""
END DIALOG
DIALOG EDITOR: "" 0x2040104 "*" "" "" ""

#sub DlgProc
function# hDlg message wParam lParam

sel message
case WM_INITDIALOG
case WM_DESTROY
PostMessage 0 2000 0 0 ;;ends MessageLoop
case WM_COMMAND goto messages2
ret
messages2
sel wParam
case IDOK
DT_GetControls hDlg ;;populated dialog variables. Optional.
case IDCANCEL
ret 1
```

Dialogs also can be used as child windows. Add WS_CHILD style, and pass parent window handle to `ShowDialog`. Child

dialogs are modeless.

QM 2.3.4. If a modeless dialog still exists when the thread ends, it receives `WM_QM_ENDTHREAD` message. It destroys the dialog, unless dialog procedure returns nonzero (then you must explicitly destroy the dialog before the thread completely ends).

How to get data from controls

The Dialog Editor creates and updates `ShowDialog` code that may contain declarations of variables for some controls. You can use the variables to initialize the controls and get data from them. Example:

```
str controls = "3 5"
str e3Nam c5Var
if(!ShowDialog("Dialog2" 0 &controls)) ret

if(c5Var=1) ;;checked
out e3Nam
```

Don't edit the declarations (the first two lines) manually because the variables must be in a particular order. The first str variable contains identifiers of controls. Then follow str variables for each control whose id is included in the first variable, in the same order. Variable names include control id. The third argument of `ShowDialog` must be address of the first variable.

When the user clicks OK, `ShowDialog` gets data from controls and populates the variables. Data format:

Control type	Variable	Notes, examples
Check, Option	"1" if checked, "0" if unchecked.	<code>if(c3Exa=1) out "checked"</code>
ComboBox, single-selection ListBox	"index text". Here index is 0-based index of selected item, or -1 if no selection.	<code>int iSel=val(cb3)</code> <code>int i=TO_CBGetItem(cb4)</code> <code>out i; out cb3</code>
Multi-selection ListBox	String consisting of '0' and '1' characters for unselected and selected items. For example, if selected are items 0 and 3 in a 4-item list box, the string will be "1001".	<code>if(lb3[0]='1') out "selected"</code>
Edit, QM_Edit (118) , rich edit	Control text.	Before QM 2.3.4: unchanged if read-only.
QM_Grid (119)	Table in CSV (116) format.	You can use ICsv (116) interface to work with CSV.
QM_ComboBox (118)	Depends on control style and flags.	Editable: control text; read-only: item index; check boxes: string of '0' and '1' characters; flag 2: hex value; flag 0x100: CSV.
Static, Button, Group, ActiveX	Unchanged.	The Dialog Editor does not add variables for most of these controls.
Other	Control's text. Before QM 2.3.4: unchanged.	The Dialog Editor adds variables only for some control types. To add for others, you can use QmSetWindowClassFlags (114) or Dialog Editor Options.

To get data from controls in a [dialog procedure](#), you can use function `DT_GetControl` or `DT_GetControls`. Also you can use window/control functions - `str.getwintext (224)`, `but (81)` (get check box state), `CB_SelectedItem` and other. Look in [category \(159\)](#) control. Also you can use control messages, most of them are documented in the MSDN Library.

How to initialize controls

In the Dialog Editor you can set text for these controls: button, group, static text, read-only edit, SysLink. To initialize other controls, before showing dialog assign certain strings to the control variables.

Control type	Variable	Control contents	Notes, examples
Check, Option	"1"	Checked.	<code>c3=1</code>
ListBox, ComboBox	List of strings. Some strings can begin with &.	Each line adds item. The & selects the item.	<code>cb3="&Zero[]One[]Two"</code> <code>lb4="Zero[]&One[]&Two"</code>
Static icon	Icon file path or macro resource (261) name. Can begin with &.	Displays the icon. The & displays	<code>si3="\$my qm\$file.ico"</code> <code>si4="&shell32.dll,15"</code>

	Can contain exe (51) resource id.	63. Dialogs 32x32 icon instead of 16x16 icon.	<pre>si6="resource:test.ico" ":150 file.ico" "&:150 file.ico"</pre>
Static bitmap	bmp, jpg or gif file path or macro resource name. Can contain exe resource id. QM 2.3.4: can be png.	Displays the picture.	<pre>sb3="\$my qm\$\file.gif" sb7="resource:test.png" sb8=":8 \$personal\$\file.bmp"</pre>
Dialog itself (id=0)	Any text.	Displays the text in the title bar.	<pre>d0="New Title"</pre> <p>Include 0 in Dialog Editor Options -> Add variables...</p>
Edit, QM_Edit (118)	Any text.	Displays the text.	<pre>e3="text"</pre>
Rich edit	Any text. Can be .rtf file path or macro resource name, preceded by &. The path can contain exe resource id (QM 2.3.0).	Displays the text or the file.	<pre>rea3="text" rea4="&\$personal\$\rich text.rtf" rea8="&resource:Document.rtf" rea5="&:10 &\$personal\$\text.txt"</pre>
Web browser control	Web page, folder or file that can be opened in web browser.	Opens the web page, folder or file. If "", opens empty page.	<pre>ax3SHD="www.quickmacros.com" ax4SHD="\$my qm\$\animated.gif" ax5SHD=""</pre>
QM_Grid (119)	Table in CSV (116) format.	Displays the table.	<pre>qmg3="a1,b1,c1[]a2,b2,c2"</pre> <p>You can use ICsv (116) interface to work with CSV.</p>
QM_ComboBox (118)	CSV. First row - control properties. Other rows - list items.		<pre>qmcb3="0[]Zero[]One[]Two" qmcb4=",,1[]Zero[]One,1[]Two"</pre>
Other	Any text.	Sets text of the control. The control can display it or not.	The Dialog Editor adds variables only for some control types. To add for others, you can use QmSetWindowClassFlags (114) or Dialog Editor Options.

Example:

```
str controls = "3 5 6 7 4"
str c3 cb5 e6 si7 sb4
c3 = 1 ;;checkbox checked
cb5 = "Sunday[]&Monday[]Tuesday" ;;combobox with three items; Monday is selected
e6 = "Default text" ;;edit control's text is "Default text"
si7 = "&files.ico" ;;32x32 icon from files.ico file
sb4 = "c:\pictures\tree.bmp" ;;picture
if(!ShowDialog("Dialog2" 0 &controls)) ret
```

To set control data in a [dialog procedure](#), you can use function [DT_SetControl](#) or [DT_SetControls](#). Also you can use window/control functions - [str.setwintext \(224\)](#), [but \(81\)](#) (set check box state), [CB_SelectItem](#) and other. Look in [category \(159\)](#) control. Also you can use control messages, most of them are documented in the MSDN Library.

Variable names

The Dialog Editor creates names for control variables. A variable name is composed from abbreviated class name, id and max 3 characters of control's name. For example, if combo box with id 15 is named "Value", variable name is cb15Val.

If you change variable names (change variable's text in the Dialog Editor), need to update variable names in the macro. The Dialog Editor can do it automatically on Save, depending on dialog settings.

Using a user-defined type

Instead of multiple str variables for controls, you can use a user-defined type. You can enter type name in Dialog Editor Options. Example:

```
type TYPE_A ~controls ~c3Che ~e4 ~cb5
TYPE_A d.controls="3 4 5"
```

```
d.cb5="Sunday[]&Monday[]Tuesday"

if(!ShowDialog("Dialog4" 0 +&d)) ret

TO_CBGetItem d.cb5
out d.cb5
```

Examples of getting control values in a dialog procedure:

```
modal dialog (don't use this for modeless dialogs)
TYPE_A& d+=DT_GetControls(hDlg)
out d.e4

modal or modeless dialog
TYPE_A d.controls="3 4 5"
DT_GetControls(hDlg &d)
out d.e4
```

Example of using the registry to save dialog data:

```
type DLGVAR ~controls ~e3 ~c4Che
DLGVAR d.controls="3 4"

if(rget(_s "DLGVAR" "\test")) _s.setstruct(d)

if(!ShowDialog("Dialog33" 0 &d)) ret

_s.getstruct(d); rset(_s "DLGVAR" "\test")
```

Smart dialogs

When you use a simple dialog, you can only initialize controls before calling [ShowDialog](#), and get data from them when [ShowDialog](#) returns. While the dialog is open, the macro waits until it is closed. That is enough in many cases. But sometimes you need to execute some code while dialog is open. For example, show an Open File dialog when the user clicks a button. Then you need a special function - *dialog procedure*.

Dialog procedure

A dialog procedure is a user-defined function (can be [sub-function \(182\)](#)) that begins with `function# hDlg message wParam lParam`. Its address is passed to [ShowDialog](#) as the second argument. To create a dialog with a dialog procedure, use menu File -> New -> New Dialog. Select 'New smart dialog'.

Example:

`\Dialog_Editor`

This code defines and shows dialog.

Everything is like with simple dialogs, but we use the second ShowDialog argument - address of dialog procedure.

```
str dd=
BEGIN DIALOG
0 "" 0x90C80AC8 0x0 0 0 224 136 "Dialog"
3 Button 0x54032000 0x0 8 8 48 14 "Test"
4 ComboBox 0x54230243 0x0 8 28 96 213 ""
1 Button 0x54030001 0x4 116 116 48 14 "OK"
2 Button 0x54030000 0x4 168 116 48 14 "Cancel"
END DIALOG
DIALOG EDITOR: "" 0x2040108 "*" "" "" ""

str controls = "4"
str cb4
cb4="&A[]B[]C"
if(!ShowDialog(dd &sub.DlgProc &controls)) ret

#sub DlgProc ;this is the dialog procedure
```

```
function# hDlg message wParam lParam

sel message
  _case WM_INITDIALOG
  _out "dialog created"

  _case WM_DESTROY
  _out "closing"

  _case WM_COMMAND goto messages2
ret
messages2
sel wParam
  _case IDOK
  _out "button OK clicked"

  _case 3
  _out "button Test clicked"

  _case CBN_SELENDOK<<16|4
  _i=CB_SelectedItem(lParam)
  _out F"ComboBox item {_i} selected"
ret 1
```

The dialog procedure is called by the system whenever the dialog receives a message (e.g., when the user clicks a button or selects a combobox item).

The first [sel \(125\)](#) is for dialog-related messages. The second [sel](#) is for control-related messages (WM_COMMAND). When a message arrives, is executed code that follows the [case](#) statement for that message.

To add [case](#) statements for various messages, you can use the Events dialog in the Dialog Editor. Then add your code in or below the [case](#) line.

Some often used messages.

WM_INITDIALOG - the dialog has been created. Here you can add code that initializes some controls, sets timers etc.

WM_DESTROY - the dialog will be destroyed. It is already hidden, but controls still exist. If the dialog procedure has allocated data that must be deleted when the dialog is closed, do it here.

WM_CREATE - although Windows does not send this message to dialogs, QM sends it before initializing controls. WM_INITDIALOG is sent after initializing controls. This message is rarely used.

WM_COMMAND - an input event in a standard control. wParam contains control id (in [low-order \(250\)](#) word) and notification code (in high-order word). lParam is control handle. Several often used notification codes:

BN_CLICKED - button is clicked. There are two special control ids:

IDOK (1) - OK button. Return 1 (default) to close the dialog, or 0 to not close.

IDCANCEL (2) - Cancel button. Return 1 (default) to close the dialog, or 0 to not close.

CBN_SELENDOK - selected combo box item. To determine which item is selected, use [CB_SelectedItem](#).

LBN_SELCHANGE - selected list box item. To determine which item is selected, use [LB_SelectedItem](#).

The [case](#) statements that handle WM_COMMAND notification codes may include notification code and control id (e.g. [case CBN_SELENDOK<<16|3](#)). Since BN_CLICKED is 0, it can be omitted ([case 3](#)).

Menus also send WM_COMMAND with notification code 0 (same as BN_CLICK). Accelerators (in QM only) and toolbar buttons too. The low-order word is the menu item id (or accelerator id, or toolbar button id). It means that you can handle menu item clicks, accelerators and toolbar button clicks in the same way as button clicks. It also means that menu item ids should not be the same as button ids, unless you need that they would behave identically.

Parameters and the return value of dialog procedure

Parameters:

hDlg - dialog window handle. In dialog procedure always use hDlg, not `win("Dialog name")`. Example: `int hwndControl=id(3 hDlg)`.
message - an integer value of the message, for example WM_INITDIALOG.
wParam, lParam - two integer values that depend on message.

The dialog procedure can return:

- 0 - the message will be processed by the system. Return 0 for all unused messages and for most used messages.
- 1 - the message will not be processed by the system.
- For WM_COMMAND can return any value, it is ignored. However on IDOK and IDCANCEL should return 1, or the dialog will not be closed.
- To return certain non-zero value as documented in the message reference, use `ret DT_Ret(hDlg theValue)`.

How to get/set control values in dialog procedure

The control variables are used before and after `ShowDialog`, but not in the dialog procedure.

To get/set control values in dialog procedure, you can use functions from the [control category \(159\)](#). For example, [str.getwintext/setwintext \(224\)](#) to get/set Edit control text; [but \(81\)](#) to click button, check/uncheck or get checked; [CB_SelectedItem](#) to get combo box selected item index. Also you can send control messages with [SendMessage](#). Control messages are documented in the [MSDN Library \(256\)](#).

Alternatively use function [DT_GetControl](#) to get control data in the format described in the [How to get data from controls](#) chapter. You can use function [DT_GetControls](#) to get data from multiple controls. Use function [DT_SetControl](#) or [DT_SetControls](#) to set control data.

More info

Usually don't need to add code to close dialog. It is automatically closed on OK and Cancel, unless you return 0 on IDOK/IDCANCEL. To close explicitly from the dialog procedure, use `clo hDlg` or [DT_Cancel](#) (same as pressing Cancel) or [DT_Ok](#) (same as pressing OK). Don't use [EndDialog](#) and [DestroyWindow](#). To close a dialog from outside if you don't have its handle, use `clo` like when you close any other window.

There are more functions that can be used in dialog procedures. Their names begin with DT_. They are in category `_other.__in_dlgproc`. For example, `other.__in_dlgproc.DT_SetAutoSizeControls`.

Examples

[Dialog examples and tutorials](#)

Also you can look for examples in "\System\Tools" folder. QM floating toolbar dialogs are implemented there. They use several private functions, usually their names begin with TO_. Don't use the private functions in your macros, because they may be changed or deleted in the future, but you can make their copies. You can use public functions from folders "\System\Functions", "\System\Dialogs\Dialog Functions" and "\System\Dialogs\Dialog Control Classes". Also you can use private functions that don't show warning "private System function X in user code..."

Multi-page dialogs

QM dialogs don't have support for real property pages (child dialogs), but you can easily show and hide groups (*pages*) of related controls so that only one page would be visible at a time. In the Dialog Editor, use buttons < and > to switch pages and create new pages. Controls with id 1 to 999 are always visible (common). Controls with id greater or equal 1000 belong to pages that can be easily shown and hidden. Each page can have maximum 100 controls. For example, controls in page 0 have id 1000 to 1099, controls in page 1 have id 1100 to 1199, and so on.

To switch pages at run time, in the dialog procedure call function [DT_Page](#).

```
function hDlg index [~map]
```

hDlg - hDlg.

index - 0-based page index. Use -1 to hide all pages.

map - can be used to map **index** to different page.

- It is list of page indices separated by spaces. For example, "0 0 1 2 2 -1" shows page 0 when **index** is 0 or 1, page 1 when **index** is 2, page 2 when **index** is 3 or 4, and none when **index** is 5.
- Use parentheses to show several pages simultaneously. For example, "0 (1 3) 2" shows pages 1 and 3 when **index** is 1.

- You can also use the same string in dialog editor Options. Then mapping will be applied at design time.

Multiple dialogs in a macro

A macro can contain more than 1 dialog. Use [sub-functions \(182\)](#) for it.

```
\Dialog_Editor

str dd=
  BEGIN DIALOG
  ...
if(!ShowDialog(dd &sub.DlgProc 0)) ret
...

#sub DlgProc
function# hDlg message wParam lParam
...

#sub Dialog2
function# [hwndOwner]

str dd=
  BEGIN DIALOG
  ...
if(!ShowDialog(dd &sub.DlgProc2 0 hwndOwner)) ret
...

#sub DlgProc2
function# hDlg message wParam lParam
...
```

In this example there are 2 dialogs. One in the parent function and one in sub-function Dialog2. When opening the Dialog Editor, you can choose which dialog to edit. If a sub-function or parent function contains multiple dialogs, the Dialog Editor uses the first dialog in it (the first dialog definition, ShowDialog statement, variable declarations, and the first found dialog procedure below it).

Unicode

QM 2.3.0 and later versions support [Unicode \(267\)](#). To enable it for whole program, check the checkbox in Options. To enable it for a dialog, select "Unicode" or "As QM" in dialog editor Options. By default, dialogs work in ANSI mode, like in previous QM versions.

If QM is running in Unicode mode, most controls work well with any text regardless of dialog Unicode mode. To use Unicode text with rich edit controls, either set Unicode mode for the dialog, or use control of class RichEdit20W or RichEdit50W instead of RichEdit20A.

When Unicode is enabled for a dialog, its dialog procedure receives Unicode versions of some messages, particularly those that can pass or query text. Then the text is in Unicode UTF-16 format. QM string functions don't understand UTF-16. You can use [str.ansi \(238\)](#) to convert the text to UTF-8 or ANSI. The [_unicode \(144\)](#) variable tells you in which mode QM is running.

For example, in a dialog procedure of a Unicode dialog, lParam of WM_SETTEXT is UTF-16 text. If the dialog is ANSI, it is ANSI text. Common controls (tree view, list view, etc) have their own Unicode mode, which is enabled in Unicode dialogs, and disabled in ANSI dialogs. In Unicode mode they send different notification messages (codes used with WM_NOTIFY message). For example, in Unicode mode a tree view control sends TVN_SELCHANGEDW instead of TVN_SELCHANGED. Control Unicode mode can be turned on/off using CCM_SETUNICODEFORMAT message, documented in [MSDN library \(256\)](#).

Fonts and colors

QM 2.2.1. Use class [__Font. Example](#).

```
Function Dialog_font_sample
\Dialog_Editor
```

```

function# hDlg message wParam lParam
if(hDlg) goto messages

str controls = "6 3"
str lb6 e3
e3="Text"
lb6="item1[]item2"
if(!ShowDialog("Dialog_font_sample" &Dialog_font_sample &controls)) ret

BEGIN DIALOG
0 "" 0x10C80A44 0x100 0 0 173 99 "Dialog Fonts"
4 Static 0x54000000 0x4 10 22 48 14 "Text"
6 ListBox 0x54230101 0x200 104 20 60 34 ""
3 Edit 0x54030080 0x204 10 38 82 16 ""
1 Button 0x54030001 0x4 4 82 48 14 "OK"
2 Button 0x54030000 0x4 56 82 48 14 "Cancel"
5 Button 0x54020007 0x4 4 6 166 54 "Text"
END DIALOG
DIALOG EDITOR: "" 0x2020103 "*" "" ""

ret
messages
sel message
case WM_INITDIALOG
__Font-- f
f.Create("Courier New" 14 1)
f.SetDialogFont(hDlg "3-5")
f.SetDialogFontColor(hDlg 0xff0000 "3 4")

__Font-- f2
f2.Create("Comic Sans MS" 7 2)
f2.SetDialogFont(hDlg "2")

__Font-- f3
f.SetDialogFontColor(hDlg 0x008000 "6")

note: __Font variables must not be local because they must exist while the dialog is open.
note: in a dialog, use different __Font variables for different fonts.

case WM_DESTROY
case WM_COMMAND goto messages2
ret
messages2
sel wParam
case IDOK
case IDCANCEL
ret 1

```

Menus and accelerators (hotkeys)

ShowDialog has an optional **menu** parameter. It can be the name of a macro (or item of other type) that contains menu definition. It also can be menu definition itself.

To create menu definitions, use the Menu Editor. You can find it in floating toolbar -> More Tools.

Example menu definition:

```

BEGIN MENU
>&File
&Open : 101 0 0 Co
-
>&Recent
Empty : 102 0 3
<

```

```

<
>&Edit
Cu&t : 103 0 0 Cx
-
Select &All : 104 0 0 Ca
<
&Help : 105 0
END MENU

```

Menu definition format

You can see that it is similar to QM menus. A single line (except <), creates a menu item. Items that open popup menus or submenus begin with >. They usually don't have an id and accelerator. The end of a popup menu or submenu is marked by < line. Hyphen is used for separators. A line can optionally begin with any number of spaces and tabs.

Syntax used for menu items:

Label : id [type] [state] [accelerator]

label - menu item text. Use ampersand to underline characters. Can be used escape sequences (not in Menu Editor).

id - menu item id. Should be 0 to 65535 (0xFFFF). When you click the menu item, the dialog procedure receives WM_COMMAND message that can be handled in the same way as it would be a button with the id. Use bigger values (e.g. above 10000) to avoid conflicts with buttons.

type - numeric combination of menu item types. Normally it is omitted or 0.

state - numeric combination of menu item states. Useful states: 3 disabled, 8 checked.

accelerator - a hotkey in [QM format \(251\)](#) (same as with [key](#)). The hotkey string is appended to the menu item label automatically, unless label contains a tab character.

id, **type** and **state** must be single numbers, without operators and named constants. More info about menu item types and states can be found in [MSDN Library \(256\)](#) (search for MENUITEMINFO structure).

Other info

In [exe \(51\)](#) also can be used menus from resources. The **menu** argument of [ShowDialog](#) must be semicolon followed by resource id, e.g. ":15". If the resource contains accelerator table with the same id, the table is used for accelerators, although accelerator text is not added (must be included in label text, after tab).

To change or remove menu at run time, call [DT_SetMenu](#) in dialog procedure. To create or load menu, use [DT_CreateMenu](#).

Menu bars and accelerators can be used with modal and modeless dialogs, but not with child dialogs.

QM 2.4.2. Menu definitions also can be used to show popup menus. Use [ShowMenu](#) or [MenuPopup.Create](#).

QM 2.4.2. You can set menu icons with [DT_SetMenuIcons](#).

Common controls, other controls

The Dialog Editor and dialog functions support controls of the following classes: Button (push button, checkbox, group), Edit, RichEdit20A, RichEdit20W, Static (static text, bitmap, icon, line), ComboBox, ListBox and [QM_Grid \(119\)](#). Working with controls of these classes is easy. You can initialize them before you call [ShowDialog](#), and get data (text, checked, etc) from them when dialog is closed.

Dialogs can contain controls of other classes too. For example, TreeView32, SysTabControl32. However, only programmers can use them. The dialog must be a "smart" dialog, otherwise the controls will be useless. To interact with these controls, are used various messages (with [SendMessage](#)). To notify the dialog about various events, these controls usually send WM_NOTIFY message, not WM_COMMAND. Messages, styles and other information can be found in [MSDN library \(256\)](#). If you don't know the value of a constant, click it, press [F1 \(245\)](#), and search for definition on the Internet.

Several samples can be found in the [forum](#). Search for the class name.

ActiveX controls

Controls that you usually use in dialogs are Windows controls. Also you can use ActiveX controls. ActiveX controls are [COM \(162\)](#) objects that provide graphical user interface. They are used differently than Windows controls. To work with Windows controls, are used messages. To work with ActiveX controls, are used functions that they provide.

To insert an ActiveX control, click "ActiveX controls..." in the Dialog Editor. It opens the [COM Libraries \(163\)](#) dialog that displays available (registered) ActiveX controls. Select the control you need and click Add Control To Dialog.

ActiveX controls are defined in [type libraries \(164\)](#), as COM classes. To use an ActiveX control in a dialog, at first declare the type library where the control class is defined. To insert type library declarations, use the COM Libraries dialog. When adding a control to a dialog, QM prompts to insert type library declaration if it still does not exist in current macro. Several type libraries, including SHDocVw (web browser control) are already declared.

You can read more about type libraries and ActiveX controls in the [Using COM Components \(163\)](#) topic.

When you have an ActiveX control created, you usually need some way to communicate with it. For this purpose, declare a variable of that class and call [_getcontrol \(167\)](#). Pass handle of ActiveX control's host window (which itself is a Windows control of "ActiveX" class). Then you can [call functions \(168\)](#) with the variable to manipulate the control. To receive events, use [_setevents \(169\)](#). The variable must be local (not global or thread). Call [_getcontrol](#) every time before using it.

Example:

```
...
sel message
  _case WM_INITDIALOG ;;dialog created
  SHDocVw.WebBrowser b
  b._getcontrol(id(3 hDlg))
  b._setevents("sub.b")
  b.Navigate("http://www.google.com")
...
  _case 4 ;;Back button
  b._getcontrol(id(3 hDlg))
  b.GoBack
...
```

There is a sample dialog with web browser control in the [forum](#).

You cannot use [str.setwintext \(224\)](#) with ActiveX controls, except web browser control. If the text is URL or path of a html, image or other file, web browser control opens that web page or file. Does not wait until finished loading. Sets silent mode to avoid script error messages. If the text is "", loads "about:blank" or clears the control. To load HTML, use function [HtmlToWebBrowserControl](#).

ActiveX controls can be used in other windows (non-dialog) too. To create an ActiveX control, use [CreateControl](#) or [CreateWindowEx](#). Class name and text must be like in dialog definitions: class "ActiveX", text like "Typelib.Class {GUID}".

QM supports run-time licensing of ActiveX controls. It allows you to add purchased ActiveX controls to macros and exes and distribute these macros and exes. When you insert an ActiveX control in the Dialog Editor using the COM Libraries dialog, QM automatically attaches encrypted run-time license key, if the control can generate it. With attached run-time license key, the control can be used on any computer. If you distribute macro (not exe), you should encrypt the macro that contains the dialog definition to prevent others to reuse the run-time license key in their macros. See [Encrypted Macros](#). Even if the macro is not encrypted, the key can only be used in Quick Macros, because only QM can decrypt the key. Also, others will not be able to create exes with the key. Note that some controls may use different ways to verify license. For example, you may have to call certain function and pass the license key.

Some ActiveX controls don't work well in QM. This is because they are designed for full-featured containers, such as Visual Basic forms, or only for certain programs. At design time, you can only move and resize ActiveX controls (or control placeholders). Properties can be changed only at run time.

Many other COM components are not controls, and must be created with [_create \(167\)](#) or other functions instead. Such components usually are in dll files. Controls usually are in ocx or dll files.

QM 2.3.4. It is possible to use some ActiveX controls even if the COM component is not registered on that computer. In dialog definition add full path of the COM component (dll or ocx) to the ActiveX text string, like ["Typelib.Class {CLSID} 'dll:\\$qm\\$\file.ocx'"](#). If the file not found, tries to create as registered. In exe, if path begins with "\$qm\$", searches for the dll in exe folder; if not found there, searches in QM folder if QM is installed on that computer.

QM 2.3.5. To create COM components without registration, you can use manifest and [__ComActivator](#) class.

See also: [Using COM components \(163\)](#).

Encrypted macros

If you encrypt macro that contains dialog definition, the dialog will not work unless the dialog definition is in a variable. Examples:

This macro will not work encrypted.

```
BEGIN DIALOG
0 "" 0x90C80AC8 0x0 0 0 224 136 "Dialog"
1 Button 0x54030001 0x4 116 116 48 14 "OK"
2 Button 0x54030000 0x4 168 116 48 14 "Cancel"
END DIALOG
DIALOG EDITOR: "" 0 "*" "" "" ""

if(!ShowDialog("This macro" 0 0)) ret
```

This macro will work always.

```
str dd=
BEGIN DIALOG
0 "" 0x90C80AC8 0x0 0 0 224 136 "Dialog"
1 Button 0x54030001 0x4 116 116 48 14 "OK"
2 Button 0x54030000 0x4 168 116 48 14 "Cancel"
END DIALOG
DIALOG EDITOR: "" 0 "*" "" "" ""

if(!ShowDialog(dd 0 0)) ret
```

The same is with menu definitions. Use a variable if want to encrypt the dialog.

Dialogs now and before QM 2.1.9

In QM versions < 2.1.9, the dialog procedure must call [DT_Init](#) on WM_INITDIALOG, [DT_DeleteData](#) on WM_DESTROY, [DT_Ok](#) on OK, and [DT_Cancel](#) on Cancel. Starting from QM 2.1.9, these functions are not necessary. Now you have to use them only if the dialog will be used by someone that has an older QM version (this does not apply to [exe-macros \(51\)](#)). Also, to avoid any problems, you should leave these functions in dialogs where they already exist.

A note about IDOK and IDCANCEL. Before QM 2.1.9, [DT_Ok](#) and [DT_Cancel](#) functions were used to close the dialog. If they were not called, the dialog was not closed. Now dialogs are closed automatically. To prevent this, return 0 on IDOK or IDCANCEL. For backward compatibility, a dialog also is not closed automatically if it calls [DT_Init](#) on WM_INITDIALOG (it is the way QM recognizes old dialogs).

Dialog examples and tutorials

Search for *ShowDialog* or *dialog* in the [forum](#).

Several topics to get started:

Tips how to [copy/paste QM code in the forum](#), and how to [use dialog examples](#).

A tutorial for a [simplest dialog with checkboxes](#).

[Running and ending macros from a dialog](#) (without closing the dialog).

[Executing code from a dialog with a combo box](#) (without closing the dialog). An [example with a list box](#).

[Collection of dialogs and other code](#)

Dialog programming info at MSDN

[Dialog box programming info at MSDN](#)

Dialog programming in QM is similar to dialog programming in C++ using Windows API, as documented in MSDN. In QM it is simplified. When you know how dialogs work at Windows API level (dialog procedure, messages, notification messages, etc), you can also understand how it works in QM.

Run or open file, folder, www page, create e-mail message

Syntax

```
run file [par] [verb] [dir] [flags] [window] [hwnd]
```

Parameters

file - [full path or filename \(246\)](#).

par - command line parameters. Used only with executable files.

verb - "open", "edit", "print", "explore" or other action that is listed in file's right click menu. Not all actions can be used. The default verb string is the one that is bold in the menu. Usually it is "open". See also flag 0x40000.

dir - default directory. Use  to extract from **file**.

flags (247):

First 8 bits	Window show state: 0 or 1 - normal (default), 2 - minimized, 3 - maximized, 4 - inactive, 7 - min. inactive, 16 - hidden. Most programs don't support this.
0x100	On error, don't show error message box.
0x200	Wait for input idle, i.e. until the program is ready to accept user input (keyboard, mouse). It works not with all windows.
0x400	Wait until the program exits. <ul style="list-style-type: none"> • Waits only if the program is actually started, and file is not shortcut.
0x800	After the program is started, wait until window window is active. <ul style="list-style-type: none"> • Waits max 5 minutes (1 minute before QM 2.3.3). Error on timeout. • Without this flag, if window and hwnd used, waits until the window is visible (can be inactive). QM 2.3.3: If flag 16 (hidden) used, waits until created (can be invisible).
0x1000	If window window exists, do nothing. If this flag not used, activates it.
0x2000	Run even if window window already exists. <ul style="list-style-type: none"> • If need to wait for window, waits for another (new) window. Therefore will fail if the attempt to run the program just activated the existing window.
0x4000	QM 2.3.4. Disable file system redirection on 64-bit Windows. Use this to run 64-bit system programs.
0x10000	QM 2.2.0. Run as administrator. On Vista/7/8/10 administrator account it does not show a consent or Run As dialog, except in exe (51) and portable (259) . To show the dialog, use verb "runas" instead. Read more in remarks.
0x20000	QM 2.2.0. The same as above, but only on administrator account. On standard user account the program will run as standard user.
0x30000	QM 2.2.0. Run as administrator if QM is running as administrator.
0x40000	QM 2.4.1. Support verbs from shell menu extensions, including "Properties".

(274)window - a window of the program.

- If the window exists, **run** only activates it (unless used **flags** 0x1000 or 0x2000).
- Otherwise runs **file**. If used flag 0x800 or **hwnd**, waits for the window.

hwnd - int variable that receives handle of window **window**.

Remarks

file can be program, document, shortcut, folder or Internet resource ("http://...", "mailto:...", etc). To open web pages, you can also use [web \(94\)](#).

When using as function, **run** returns handle of the started process (running program) or -1. Later it must be closed with [CloseHandle](#), unless assigned to a `__Handle` variable. Example:

```
__Handle h=run("notepad.exe"); wait 0 H h; out "closed".
```

When using as function with flag 0x400 (wait for exit), **run** returns program's exit code.

The speed depends on [spe \(90\)](#).

To run/open objects that cannot be specified by path (e.g., Control Panel objects), can be used [ITEMIDLIST string \(246\)](#).

Class id strings in format "{XXXX}{XXXX}" also are supported. Starting from QM 2.2.0, some other functions also can use it, e.g. can get icon, create shortcut.

On [Vista/7/8/10 \(277\)](#), if UAC is on, most programs don't have administrator privileges even on administrator account. They have Medium integrity level (IL). If a program has Administrator IL, programs launched from it also run as administrator. However [run](#) behaves differently, except in [exe \(51\)](#) and [portable \(259\)](#). Even when QM is running as administrator, programs launched by [run](#) have Medium IL. To run a program as administrator, use flags 0x10000-0x30000. Function [web \(94\)](#) also launches IE as not administrator. Functions [StartProcess \(114\)](#) and [RunAs](#) also can launch programs with different IL. Other functions ([RunConsole2](#), [CreateProcess](#), etc) launch programs with the same IL as of QM, but without uiAccess.

Tips

You can drag and drop a file onto the macro text to insert [run](#) command for that file. Ctrl can be used to insert shortcut path instead of target path. You also can drag Internet links, virtual folders/objects, multiple files. You can also drop onto a toolbar.

If **file** is document, opens it in default program for that file type. To open in certain program, use program in **file** and document in **par**. Example: `run "wordpad.exe" "c:\x.txt"`.

If macro intends to do something with new window, but program loads slowly, try flag 0x200, or/and **window** together with flag 0x800. Or, after [run](#) include [wait \(88\)](#) or [wait for \(89\)](#) command ("wait", "wait for active window", etc).

Sometimes, program started by [run](#) shows a dialog. Macro should close the dialog, but [run](#) waits until you manually close the dialog. In such case, create a function that closes the dialog, and start it from the macro using [mac](#). See example.

To run a console program and capture its output, use [RunConsole2](#) instead.

To run a program as another user, use [RunAs](#) instead. It does not require user interaction if you specify encrypted password. It cannot be used on Vista/7/8/10 to run as current user with elevated privileges (instead use [run](#) with flags 0x10000-0x30000 or **verb** "runas", or [StartProcess \(114\)](#)).

64-bit Windows has two System32 and Program Files folders. [Read more \(277\)](#).

Before QM 2.3.3, did not support [relative path \(246\)](#) for files in QM folder.

How to know if a program is running? Search for its window with [win \(77\)](#). If does not have windows, use function [ProcessNameTold \(114\)](#).

Examples

```
run "c:\f\text.txt" ;;open text.txt
run "notepad.exe" ;;run Notepad
run "c:\f\my file.lnk" ;;run shortcut
run "c:\m" "" "explore" ;;explore folder
run "c:\t.txt" "" "print" ;;print "t.txt"
run "control" "appwiz.cpl" ;;open Control Panel "Add/Remove Programs"
run "notepad.exe" "" "" "" 3 ;;run Notepad, maximized
```

Run or activate Notepad:

```
run "notepad.exe" "" "" "" 0 "Notepad"
```

Run Notepad with parameters "s.cpp", default directory "c:\f":

```
run "notepad.exe" "s.cpp" "" "c:\f"
```

Run program and wait max. 15 s until CPU usage is < 10%:

```
run "app.exe"; wait 15 P 10
```

```
run "http://www.aaa.com" ;;open web page
run "mailto:name@isp.com" ;;create new e-mail message
run "mailto:name@isp.com?subject=Question" ;;create new e-mail message
```

Run program with command line with variables:

```
str x.expandpath("$documents$\test.txt")
int y=5
str cl=F"/X '{x}' /Y {y}"
run "zzz.exe" cl
```

Run Notepad and wait untill its process ends:

```
run "notepad.exe" "" "" "" 0x400
```

Run program that shows a dialog at startup, which causes run to wait:

```
mac "CloseDialog"  
run "program"  
...
```

Function CloseDialog:

```
wait(10 "Dialog Name"); err ret  
key Y
```

Copy, rename or move files and folders

Syntax

```
cop[+|-|!] from to [flags]
ren[*|+|-|!] from to [flags]
```

Parameters

from - source file or folder.

to - destination folder or file. If option * (rename) used, must be new filename without path.

flags (247) - see [here \(67\)](#).

Options:

Default	if destination file already exists, ask what to do (show dialog).
+	if destination file already exists, rename the copied or moved file.
-	if destination file already exists, replace it.
!	if destination file already exists, error. You can use err (129) to continue. This option cannot be used with list of files.
* (with ren)	rename. Default: move, or move and rename.

Remarks

[cop](#) copies a file or folder.

[ren](#) moves or/and renames a file or folder.

Like all QM file functions, support [special folders \(246\)](#). Also support relative paths (in QM 2.3.0 and above it works better).

To copy or move multiple files, can be used one of the following:

1. Use [wildcard characters \(271\)](#) in filename part of **from**. Then **to** must be folder. All matching files will be copied to the folder. To copy matching subfolders too, FOF_FILESONLY flag must be removed. Use the 'Copy Files' dialog or 'Move/Rename Files' dialog to insert correct flags.
2. Use list of files (full paths) in **from**, one file in a line. Then **to** can be either single folder or list of destination files.
3. Enumerate files. To insert the code, use the 'Enumerate Files' dialog.

The functions work in the same way as if the user would do it manually. They are quite slow with small files, especially on Vista. You can instead use faster functions: [FileCopy](#), [FileMove](#) and [FileRename](#).

QM 2.3.0. Removed autodelay.

Tips

Drag and drop a file onto the macro text to insert full path.

Examples

Copy file "x.txt" from "c:\a" to "c:\b" folder:

```
cop "c:\a\x.txt" "c:\b\x.txt"
```

Copy file. If "c:\b\x.txt" already exists, copy with name "Copy of x":

```
cop+ "c:\a\x.txt" "c:\b\x.txt"
```

Copy file. If "c:\b\x.txt" already exists, delete it:

```
cop- "c:\a\x.txt" "c:\b\x.txt"
```

Copy all text files from "c:\a" to "c:\b" folder:

```
cop "c:\a\*.txt" "c:\b"
```

Copy all files (but not subfolders):

```
cop "c:\a\*.*)" "c:\b"
```

Copy file; use variables:

```
str sfrom = "c:\a\x.txt"
```

```
str sto = "c:\b\x.txt"
cop sfrom sto
```

Move file "g.gif" from "c:\a" to "c:\b" folder:

```
ren "c:\a\g.gif" "c:\b"
```

Rename file g.gif to h.gif:

```
ren "c:\a\g.gif" "c:\a\h.gif"
```

The same:

```
ren* "c:\a\g.gif" "h.gif"
```

Copy multiple files from My Documents to a folder on desktop:

```
lpstr dest="$desktop$\from my doc"
mkdir dest
lpstr files=
    $personal$\book1.xls
    $personal$\page1.htm
    $personal$\document1.doc
cop files dest
```

Delete files and folders

Syntax

```
del [-] file [flags]
```

Parameters

file - file or folder.

flags (247) - see [here \(67\)](#).

Options:

Default	move file to Recycle Bin.
-	delete file.

Remarks

Deletes a file or folder.

Like all QM file functions, supports [special folders \(246\)](#). Also supports relative paths (in QM 2.3.0 and above it works better).

To delete multiple files, can be used one of the following:

1. Use [wildcard characters \(271\)](#) in filename part of **from**. All matching files will be deleted. To delete matching folders too, FOF_FILESONLY flag must be removed. Use the 'Delete Files' dialog to insert correct flags.
2. Use list of files (full paths) in **from**, one file in a line.
3. Enumerate files. To insert the code, use the 'Enumerate Files' dialog.

The function works almost in the same way as if the user would do it manually. It is quite slow with small files, especially on Vista. You can instead use faster function: [FileDelete](#).

QM 2.3.0. Removed autodelay.

Tips

Drag and drop a file onto the macro text to insert full path.

Examples

Send file to Recycle Bin

```
del "$desktop$\text.txt"
```

Delete file

```
del- "$desktop$\text.txt"
```

Empt Recycle Bin

```
SHEmptyRecycleBin 0 0 SHERB_NOCONFIRMATION
```

Flags for cop, ren and del

See also: [flags \(247\)](#).

FOF_MULTIDESTFILES	0x1	Indicates that the to member specifies multiple destination files (one for each source file) rather than one directory where all source files are to be deposited.
FOF_SILENT	0x4	Does not display a progress dialog box.
FOF_RENAMEONCOLLISION	0x8	Gives the file being operated on a new name (such as "Copy #1 of...") in a move, copy, or rename operation if a file of the target name already exists.
FOF_NOCONFIRMATION	0x10	Responds with "yes to all" for any dialog box that is displayed.
FOF_ALLOWUNDO	0x40	Preserves undo information, if possible. With del , uses recycle bin.
FOF_FILESONLY	0x80	Performs the operation only on files if a wildcard filename (*.*) is specified.
FOF_SIMPLEPROGRESS	0x100	Displays a progress dialog box, but does not show the filenames.
FOF_NOCONFIRMMKDIR	0x200	Does not confirm the creation of a new directory if the operation requires one to be created.
FOF_NOERRORUI	0x400	don't put up error UI
FOF_NOCOPYSECURITYATTRIBS	0x800	don't copy file security attributes
FOF_NORECURSION	0x1000	Only operate in the specified directory. Don't operate recursively into subdirectories.
FOF_NO_CONNECTED_ELEMENTS	0x2000	Do not move connected files as a group (e.g. html file together with images). Only move the specified files.
FOF_WANTNUKEWARNING	0x4000	Send a warning if a file is being destroyed during a delete operation rather than recycled. This flag partially overrides FOF_NOCONFIRMATION.

Commands	default flags
cop, ren	FOF_ALLOWUNDO FOF_FILESONLY FOF_NOCONFIRMMKDIR. If opt err 1 is set, also adds FOF_NOERRORUI.
cop+, ren+	The same as cop/ren , and adds FOF_NOCONFIRMATION FOF_RENAMEONCOLLISION
cop-, ren-	The same as cop/ren , and adds FOF_NOCONFIRMATION
del	FOF_FILESONLY FOF_NOCONFIRMATION FOF_SILENT FOF_NOERRORUI FOF_ALLOWUNDO
del-	The same as del , and removes FOF_ALLOWUNDO

If you use **flags**, it replaces default flags, except flags that are added/removed by option - or +.

Create new folder

Syntax

```
int mkdir(folder [parent])
```

Parameters

folder - name of new folder. If **parent** isn't used or is "", **folder** must be full path.

parent - container folder or drive. If **folder** includes full path, **parent** isn't used. Default: "".

Remarks

Creates new folder.

Returns 1 if successfully created. Error if failed. Returns 0 if the folder already exists.

Also creates parent folders, if need. For example, `mkdir "x:\a\b\c"` creates folders a, b and c, if they did not exist.

See also: [ChangeFileSecurity](#).

Examples

```
mkdir "$Desktop$\New Folder" ;;create "Nef Folder" on the desktop  
mkdir "New Folder" "$Desktop$" ;;the same
```

Find or enumerate files

Obsolete. Use [FileExists \(114\)](#). To create code, use dialog "If file exists" or "Get file info" or "Enumerate files".

Syntax

```
lpstr dir([file] [flags])
```

Parameters

file - file, folder or drive.

- Filename can contain [wildcard characters \(271\)](#).
- If omitted or "", finds next file that matches **file** with wildcard characters used with **dir** previously (in current function).

flags:

0	Find only files.
1	Find only folders and drives.
2	Find all.
3	If flags is 3 or omitted, uses flags used with dir previously (in current function).

Remarks

If the file exists, returns its filename with extension. Else returns 0.

See also: [str.searchpath \(231\)](#), [special folders \(246\)](#)

Examples

```
If file exists:
if(dir("$desktop$\test.txt"))
_out "file exists"

.txt files in desktop folder:
lpstr s=dir("$desktop$\*.txt")
rep
_if(s = 0) break
_out s
s = dir

Folders in c:\windows:
lpstr s=dir("c:\windows\*" 1)
rep
_if(s = 0) break
_out s
s = dir

Drives:
Wsh.FileSystemObject fso._create
Wsh.Drive dr
foreach dr fso.Drives
_out dr.Path
_out dr.DriveType ;;0 unknown, 1 removable, 2 fixed, 3 network, 4 CD-ROM, 5 RAM disk
```

If file or folder exists

Obsolete. Use [FileExists \(114\)](#) or [str.searchpath \(231\)](#).

Syntax

```
iff[-] file
(tab) statements
(tab) ...
[else
(tab) statements
(tab) ...]
```

Can be single line:

```
iff[-](file) statements
[else statements]
```

Parameters

file - file to search for. Can be file or folder. Also can be drive. Can be full path or filename.

Options:

-	not.
---	------

Remarks

Similar to [if \(123\)](#).

If **file** exists, executes **statements** after **iff** and skips **statements** after **else** (if used). Else skips **statements** after **iff** and executes **statements** after **else** (if used).

QM 2.3.3. Can be used as function. Returns 1 if the file exists, 0 if not.

If **file** is not full path, searches in [these places \(246\)](#). Works like [str.searchpath \(231\)](#).

Examples

```
iff("c:\m\x.txt") mac- "Copy" ;;if file exists, execute macro "Copy"
iff-("notepad.exe") bee ;;if file doesn't exist, beep
iff(s) out s ;;if file exists, show path. Path is in variable s
iff-("$My Pictures$") end ;;if special folder doesn't exist, end macro
```

Can be used as function

```
if(iff("file1") and !iff("file2")) out "file1 exist; file2 does not exist."
```

Compress/extract files

Syntax1 - compress

```
zip zipfile files [flags] [warnings] [nozipext]
```

Syntax2 - extract

```
zip- zipfile folder [flags2] [warnings]
```

Parameters

zipfile - standard zip file.

files - file or folder to compress. Supports [wildcard characters \(271\)](#). Can also be list of files/folders. In the list, lines beginning with > character are skipped.

flags (247):

1	Minimal compression level (faster).
2	Store full paths.
4	Don't store relative paths when adding folder.
8	(QM 2.3.0) Add files but don't compress. Don't use together with flag 1.

warnings - str variable that receives list of errors ("file not found", etc). If used, files failed to add/extract are skipped. If not used or is 0, on failure is generated error.

nozipext - (QM 2.3.0) comma-separated list of filename extensions. Files of these types will be added but not compressed. Example: "mp3,rar,jpg,htm".

folder - folder where to extract files.

flags2 (247) - combination of the following values. Default: 0.

1	Overwrite read-only files.
---	----------------------------

Remarks

Syntax1: Creates new zip file **zipfile** and adds **files** to it. If the specified zip file already exists, overwrites.

Syntax2: Extracts all files from zip file **zipfile** to the specified folder.

Zip file format does not support Unicode filenames within zip files. When QM runs in ANSI mode, Unicode characters are converted to similar ANSI characters or ?, and therefore the zip file may be invalid. To partially support Unicode, when QM runs in Unicode mode, it adds/extracts UTF-8 file names. Then the zip file is always valid, although other programs will not decode UTF-8. Generally you should not use Unicode filenames with zip.

Cannot create or extract zip files bigger than 2 GB.

Example

```
str zf="$desktop$\test.zip"
str sf.getmacro("List of files to zip")
str sw

out "zip"
zip zf sf 0 sw
if(sw.len) out sw

out "unzip"
zip- zf "$desktop$\unzip" 0 sw
if(sw.len) out sw
```

Activate window, set focus

Syntax

```
act [window] [flags]
```

Can be used as function:

```
int act([window] [flags])
```

Parameters

[\(274\)](#)**window** - window or child window.

flags [\(247\)](#) (QM 2.3.2):

1 (QM 2.3.2)	Not error if activates other window of same thread. Add this flag when act activates correct window but throws error after ~1 s.
--------------	---

Remarks

If **window** is a top-level window, activates.

If **window** is a [child](#) [\(275\)](#) window, activates its parent window and sets focus.

If **window** is omitted, activates previous window, as if you would press Alt+Tab.

If used as function, **act** activates window and returns window handle.

The speed depends on [spe](#) [\(90\)](#).

Error if fails to activate.

Tips

If **act** does not work properly, try to check "Disable Windows foreground lock feature" in Options.

Examples

```
act "Notepad" ;;activate "Notepad" window
act "+IEFrame" ;;activate window with class name "IEFrame"
act ;;activate next window
act win(100 200) ;;activate window from point (100, 200 pixels of the screen)
act id(142 "Calc") ;;activate (set focus) "Calc" child with id = 142
```

Close window

Syntax

`clo` [`window`]

Parameters

[\(274\)window](#) - top-level or [child \(275\)](#) window. Default: active window.

Remarks

The window may reject the request. For example, the window may be hung, or display a "Save?" message box. QM ignores that, ie does not wait and does not generate error if the window remains open.

Error if the window does not exist. Use [err \(129\)](#) to continue the macro.

The speed depends on [spe \(90\)](#).

If the window belongs to other program or thread, `clo` sends WM_SYSCOMMAND/SC_CLOSE or/and WM_CLOSE message, depending on window type and state. With some windows it may not work well. You can send one of these messages instead of using `clo`. See examples.

To destroy a window of current thread, you can use `clo` or [DestroyWindow](#). If used `clo`, the window will receive WM_CLOSE message. If it is a dialog with Cancel button, it is equal to pressing the Cancel button. If using [DestroyWindow](#), the window will not receive WM_CLOSE. It will receive WM_DESTROY in either case. Don't use [DestroyWindow](#) to destroy modal dialogs.

See also: [shutdown \(104\)](#), [ShutDownProcess](#), [CloseWindowsOf](#).

Examples

```
clo "Notepad" ;;close "Notepad" window
clo ;;close active window
clo win("Notepad" "Notepad"); err ;;close "Notepad"; don't generate error if it does not exist

post message
int h=win("Notepad" "Notepad")
PostMessage h WM_SYSCOMMAND SC_CLOSE 0
or
PostMessage h WM_CLOSE 0 0
```

Minimize, maximize, restore window

Syntax1 - set state

```
min [window]
max [window]
res [window]
```

Syntax2 - get state

```
int min([window])
int max([window])
int res([window])
```

Parameters

[\(274\)](#)**window** - top-level or [child \(275\)](#) window. Default: active window.

Remarks

Syntax1

Minimizes ([min](#)), maximizes ([max](#)) or restores ([res](#)) window.

[res](#) works like the 'Restore' item in the window's system menu. If the window is minimized, restores it to previous state (normal or maximized). If maximized, makes normal. If need to restore only if minimized, use [if min](#) (see example) or [act \(72\)](#) (it restores if minimized).

The speed depends on [spe \(90\)](#).

Syntax2

Returns 1 if window is minimized ([min](#)), maximized ([max](#)) or normal ([res](#)). If not, returns 0.

Example

```
min "Notepad" ;;minimize "Notepad" window

restore "Notepad" window to previous state if it is minimized
int w=win("Notepad")
if(min(w)) res w
```

Change window position or size

Syntax

```
mov left top [window] [flags]
siz width height [window] [flags]
mov+ left top width height [window] [flags]
```

Parameters

left, top - new [coordinates \(241\)](#) of top-left corner.

width, height - new width and height.

[\(274\)window](#) - top-level or [child \(275\)](#) window.

flags (247):

1	Don't change left (with mov and mov+) or width (with siz).
2	Don't change top (with mov and mov+) or height (with siz).
4	Coordinates are relative to the work area. Only with top-level windows.
0x100, 0x200	QM 2.3.6. Don't change width or height when used with mov+ .

Remarks

[mov](#) changes **window** position.

[siz](#) changes **window** dimensions.

[mov+](#) changes **window** position and dimensions. Added in QM 2.3.6.

If **window** is omitted or "", uses the active window.

Coordinates must be relative to the primary monitor. If you want to use coordinates relative to some other monitor, use function [XyMonitorToNormal](#) to convert them.

If it is child window (control), coordinates must be relative to the client area of its direct parent window.

If **width** and/or **height** is a double number, it is interpreted as fraction of screen (if top-level window) or fraction of client area of direct parent window (if control). See examples. Integer numbers are pixels.

Window position and size will not be changed if it is minimized. Also you should not change it if the window is maximized. To ensure that Notepad window is not minimized and not maximized, use this code:

```
int w2=win("Notepad")
if(min(w2)) res w2
if(max(w2)) res w2
```

The speed depends on [spe \(90\)](#).

There are more functions to move or resize windows:

- [CenterWindow](#) - move to screen center, corners, certain monitor.
- [EnsureWindowInScreen](#) - ensure that whole window is in its or specified monitor.
- [MoveWindowToMonitor](#) - move window to another monitor.
- [ArrangeWindows](#) - arrange multiple windows, e.g. tile or minimize.
- [SaveMultiWinPos](#), [RestoreMultiWinPos](#) - save/restore positions of multiple windows.
- [AdjustWindowPos \(114\)](#).
- Windows API functions [MoveWindow](#), [SetWindowPos](#), [SetWindowPlacement](#). Note that with [DPI-scaled windows \(243\)](#) they use logical coordinates (not what you see), whereas QM functions use physical (what you see).

To get help, type or click function name and press F1. Note that these functions don't accept window name. Use window handle. Use function [win](#), [child](#) or [id](#) to get handle. Also, they accept only integer coordinates.

Examples

```
mov 500 0 "Notepad" ;;move "Notepad" to 500 0 pixels
mov 0.5 0.5 "Notepad" ;;move "Notepad" to the middle
siz 300 200 "Notepad" ;;change "Notepad" size
```

75. mov, siz

```
siz 1.0 0.25 "Notepad" ;;change "Notepad" size. Width will be screen width, height 0.25 of screen height  
mov+ 100 100 400 400 win("Notepad") ;;move-resize window
```

Hide or show window

Syntax1 - set state

```
hid[-] [window]
```

Syntax2 - get state

```
int hid(window)
```

Parameters

[\(274\)](#)**window** - top-level or [child \(275\)](#) window. If omitted, uses active window (with [hid](#)), or last hidden window (with [hid-](#)).

Options:

Default	hide.
-	show.

Remarks

Syntax1

Hides or shows window.

The speed depends on [spe \(90\)](#).

Syntax2

Returns 1 if window is hidden, or 0 if visible. Calls Windows API function [IsWindowVisible](#). See also [IsWindowCloaked](#).

Tips

If you hide an active window, it may stay active even if invisible. To deactivate, after [hid](#) use [act](#) to activate next window. See example.

[hid-](#) does not activate the window. To unhide and activate, use [act](#) instead.

Examples

```
hid "Notepad" ;;hide "Notepad"
hid- "Notepad" ;;unhide "Notepad"
hid- ;;unhide last hidden window
```

Hide Notepad and make next window active:

```
int h=win("Notepad")
hid h
if(h=win) act
```

Find window (get handle); compare properties

Syntax1 - specified window

```
int win(name [class] [exename] [flags] [propCSV] [matchindex|array])
int wintest(hwnd name [class] [exename] [flags] [propCSV])
```

Before QM 2.3.4 used [this](#) instead.

Instead of **propCSV** used two parameters **x y** and several additional flags.

```
int win(name [class] [exename] [flags] [x y] [matchindex|array])
int wintest(hwnd name [class] [exename] [flags] [x y])
```

x y - used for multiple purposes, depending on flags. Default: the window must contain this point in screen.

These flags can be used to change **x y** meaning:

64	x is handle of owner window.
128	x is style.
0x100	y is extended style.
0x8000	x and y used to specify a callback function and a value to pass to it.

Syntax2 - window from point

```
int win(x y [workarea])
int wintest(x y [workarea])
```

Syntax3 - window from mouse position

```
int win(mouse)
int wintest(hwnd mouse)
```

Syntax4 - the active window

```
int win
```

Parameters

name - window title.

- By default, **name** can be partial and must match case.
- Empty string ("") matches any name.

class - window class name.

- Must be full or with wildcard characters (*?). Case insensitive.
- Default: "" (any).
- Before QM 2.3.4, to use wildcard characters, need flag 0x800.
- You can see class name in QM status bar.

exename - program. Default: "" (any). Can be:

- Filename (e.g., "NOTEPAD") or full path (e.g., "\$system\$\notepad.exe").
- QM 2.2.0. Process id.

flags (247):

1	name is full or with wildcard characters (196) (*?). <ul style="list-style-type: none"> For example, if window name must end with " - Notepad", use "*" - Notepad" and flag 1. If it must be exactly "Notepad", use "Notepad" and flag 1. String "" matches windows with no name.
2	name is case insensitive.
4	Window must not be popup (275) .
8	Window must be popup.
16	name is list of names.
32	exename is owner window. Can be handle, name or +classname.
0x200	name is regular expression (198) .

0x400 Must be visible. It is default if **class** is not specified, unless used `opt hidden 1` before. It isn't default for [wintest](#).

propCSV - list of other properties in format "name1=value1[;name2=value2[;...]". It is [CSV \(116\)](#) string with separator =. Default: "".

name	value
owner	Owner window handle. Example: <code>F"owner={hwndOwner}"</code>
xy	x and y coordinates of a point (241) in screen that the window must contain. Examples: <code>"xy=100 50"</code> <code>"xy=0.9 0.1" ;;near top-right</code>
style	Style, optionally followed by a mask. Examples: <code>"style=0x54032000" ;;must have exactly this style</code> <code>F"style=-1 {WS_DISABLED}" ;;must be disabled</code> <code>F"style=0 {WS_DISABLED}" ;;must not be disabled</code>
exStyle	Extended style, optionally followed by a mask.
cClass, cText, cId, cFlags	The window must have the specified control. Parameters are as with child (83) , all optional. Example: <code>"cClass=Static[] cText=Example*[] cFlags=1"</code>
GetProp	The window must have the specified window property . Property name, optionally followed by value. If only name, value can be any nonzero. Read more . Window properties is a Windows feature that allows applications to assign named values to windows. A macro also can set window properties. Use function SetProp . Name can be any string without spaces. Value - any nonzero number. Use SetProp carefully, don't use many names, or applications may stop working. Property names are used by many applications, and the number of names is limited. Remove with RemoveProp when don't need.
callback	Address of a callback function (273) , optionally followed by a value to pass to it. Example: <code>F"callback={&Function} {aValue}"</code>
threadId (QM 2.4.2)	Thread id. Some functions to get it: GetCurrentThreadId , GetWindowThreadProcessId , GetQmThreadInfo .
processHandle (QM 2.4.2)	Process handle. Example: <code>Handle hp=run("notepad.exe")</code> <code>int w=wait(30 WC win(" " "Notepad" " " 0 F"processHandle={hp}"))</code> Note: it is not the same as process id. If you have id, pass it as the exename argument instead.

matchindex (QM 2.2.0) - 1-based index of matched window. Use when there are several windows that match other properties (**name**, **class**, etc).

array (QM 2.2.1) - variable of type [ARRAY\(int\)](#) that will receive handles of all matching windows. See also [GetMainWindows](#).

x, y (syntax2) - a [point \(241\)](#) in screen.

- The function gets the window that contains this point.

workarea (syntax2) - if nonzero, coordinates are relative to the work area (the screen area used for a maximized window).

mouse (syntax3) - literal `mouse`.

hwnd ([wintest](#)) - handle of window to test.

Remarks

[win](#) returns [top-level \(275\)](#) window [handle \(274\)](#). If window not found, returns 0.

The handle then can be passed to any function that has a window parameter.

To test window visibility, the function calls [IsWindowVisible](#), and does not call [IsWindowCloaked](#). For example, it can find inactive Windows 8 app windows and windows on inactive Windows 10 virtual desktops.

[wintest](#) compares window (**hwnd**) properties with the specified properties, and returns 1 if they match, or 0 if not. With flag 16 returns 1-based index in the list. Error if **hwnd** is 0 or invalid. See also [WinTest \(114\)](#).

Tips

To create code for [win](#), use dialog "Find window or control". Or press Ctrl+Shift+Alt+W, it shows a menu that creates code for

some window functions. Recording and various other dialogs also create code with [win](#).

You can see window class, name and exename in QM status bar. Also you can use dialog "Explore windows", it shows more window properties.

Examples

```
int h = win ;;active window
h = win("Notepad") ;;name "Notepad"
h = win("Find" "#32770" "NOTEPAD" 1|0x400) ;;name "Find", class "#32770", program "notepad", name must
match exactly, must be visible
h = win("" "Notepad" "" 0 "xy=100 100") ;;class "Notepad", must be at 100x100 pixels in screen
h = win(200 0.5) ;;window that is at 200 pixels horizontally and half of screen height vertically
act win(mouse) ;;activate window that is at the mouse position

int h=win
sel wintest(h "Visual[]Quick" "" "" 16)
_case 1 out "Visual"
_case 2 out "Quick"
_case 0 out "other"

out
ARRAY(int) a; int i; str sc sn
out "[][9]ALL VISIBLE WINDOWS"
win("" "" "" 0 0 0 a)
for(i 0 a.len)
_sc.getwinclass(a[i])
_sn.getwintext(a[i])
_out "%i '%s' '%s'" a[i] sc sn

out "[][9]ALL INVISIBLE WINDOWS"
opt hidden 1
win("" "" "" 0 0 0 a)
for(i 0 a.len)
_if(!hid(a[i])) continue ;;this window is visible
_sc.getwinclass(a[i])
_sn.getwintext(a[i])
_out "%i '%s' '%s'" a[i] sc sn

out "[][9]ALL WINDOWS OF EXPLORER"
opt hidden 1
win("" "" "explorer" 0 0 0 a)
for(i 0 a.len)
_sc.getwinclass(a[i])
_sn.getwintext(a[i])
_out "%i '%s' '%s'" a[i] sc sn
```

If window is active/focused

Syntax

```
ifa[-] window
(tab) statements
(tab) ...
[else
(tab) statements
(tab) ...]
```

Can be single line:

```
ifa[-] (window) statements
[else statements]
```

Parameters

window (274) - top-level or [child \(275\)](#) window.

Options:

-	not.
---	------

Remarks

Similar to [if \(123\)](#).

If **window** exists and is active, executes **statements** after **ifa** and skips **statements** after **else** (if used). Else skips **statements** after **ifa** and executes **statements** after **else** (if used). If child window, tests whether it has focus.

QM 2.3.3. Can be used as function. Return 1 if the window is active, 0 if inactive or not found.

Tip: For "if window exists" use [win \(77\)](#) and **if**. See the last example.

See also: [wintest \(77\)](#)

Examples

```
ifa "Notepad"
_out "'Notepad' is active"
_key Cv
else
_act "Notepad"
_bee

use as function
if(ifa("Notepad") or ifa("WordPad")) out "Notepad or WordPad window is active."

check if exists
int w=win("Notepad" "Notepad")
if w
_out "'Notepad' exists"
```

Some Windows API window functions

Here hWnd is window handle. Use function [win \(77\)](#) to get it. For controls, use [child \(83\)](#) or [id \(82\)](#).

IsWindow (hWnd)	returns 1 if the handle is valid window handle and the window is not destroyed.
IsZoomed (hWnd)	returns 1 if window is maximized. See also max .
IsIconic (hWnd)	returns 1 if window is minimized. See also min .
IsWindowVisible (hWnd)	returns 1 if window is visible. See also hid .
IsWindowEnabled (hWnd)	returns 1 if window is enabled.
EnableWindow (hWnd fEnable)	enables (fEnable 1) or disables (fEnable 0) window.
MoveWindow (hWnd x y nWidth nHeight bRepaint)	moves and resizes window. See also mov+ .
SetWindowPos (hWnd hWndInsertAfter x y cx cy wFlags)	moves and/or resizes and/or sets Z order , and/or more. For wFlags use SWP_ constants.
GetWindowRect (hWnd RECT*lpRect)	get window dimensions. See also DpiGetWindowRect .
GetParent (hWnd)	returns handle of direct parent window of child window.
GetAncestor (hWnd gaFlags)	returns handle of container window. gaFlags: 1 parent (the same as GetParent), 2 top-level parent, 3 top-level parent or owner.
WindowFromPoint (xPoint yPoint)	returns handle of window or child window from point. See also win , child .

<code>type POINT x y</code>	used to store point coordinates
<code>type RECT left top right bottom</code>	used to store rectangle coordinates

You can use functions [GetSystemMetrics](#) and [SystemParametersInfo](#) to get various system parameters. These and other Windows API functions are documented in [MSDN library \(256\)](#).

Examples

```

Toggle active window's maximized/restored state:
if(IsZoomed(win)) res; else max
Display dimensions of active window:
RECT r; DpiGetWindowRect(win &r)
out "x=%i y=%i width=%i height=%i" r.left r.top r.right-r.left r.bottom-r.top
Get the work area (the portion of the screen not obscured by the system taskbar or by application desktop toolbars):
RECT r; SystemParametersInfo(SPI_GETWORKAREA 0 &r 0)

```

Click menu item, get state

Syntax1 - click

```
men [+] menuid|menupath [window]
```

Syntax2 - get state

```
int men(menuid|menupath [window])
```

Parameters

(274)window - top-level window with menu. Default: active window.

menuid - menu item identifier. Integer.

menupath - path to menu item, like "&File\&Open".

- Menu item labels in the path can be partial, but from the beginning.
- Underlined characters must be preceded with &. To see underlined characters, show menu with keyboard, e.g. Alt.

Options:

+	system menu (menu that appears when you right click window title bar).
---	--

Remarks

Syntax1

Simulates menu click. It posts WM_COMMAND or WM_SYSCOMMAND message. Usually works even if window is inactive or hidden. Doesn't work with most nonstandard menus, especially when used **menupath**.

You can record menu commands. It is the way to know menu item id. To start recording, click menu Tools -> Record Menu. Not all menus can be recorded. If does not record, it means that **men** cannot be used with that menu.

The speed depends on [spe \(90\)](#).

Syntax2

Returns menu item state. It is combination of **MF_...** [flags \(247\)](#), eg **MF_CHECKED**, documented in [MSDN library \(256\)](#).

Often menu items are updated just before showing the menu (not immediately after you click a menu item), therefore to get correct menu item state may need to temporarily show the menu, eg with [key](#), then call **men**.

Examples

Click menu item with id=3 in "Notepad" window:

```
men 3 "Notepad"
```

Click menu "Select All" in "Notepad" window:

```
men "&Edit\Sel" "Notepad"
```

Click menu Options->Font->Large in window with class name "HH Parent":

```
men "&Options\&Font\&Large" "+HH Parent"
```

Click button, or get checked state

Syntax1 - click, check, etc

```
but [+|-|*|%] hwndbutton
```

Syntax2 - click, check, etc

```
but [+|-|*|%] id|text window
```

Syntax3 - get checked state

```
int but(hwndbutton)
```

Syntax4 - get checked state

```
int but(id|text window)
```

Parameters

hwndbutton - button handle. To get button handle, use function [id \(82\)](#) or [child \(83\)](#).

(274)window - top-level parent window. The function searches only in the first found matching window.

id - button id.

text - button label. Can be partial, but from beginning. Underlined characters must be preceded with &. The window must not have other controls with the same text.

Options:

Default	Click.
+	Check checkbox.
-	Uncheck checkbox.
*	Toggle checkbox.
%	Click if unchecked.

Remarks

Syntax1 and 2

Clicks, checks or unchecks a button, check box or option (radio) button.

Option characters usually are used with check boxes. Option + also can be used with push buttons; often it is more reliable.

Syntax3 and 4

Returns check state of check box or radio (option) button: 0 - unchecked, 1 - checked, 2 indeterminate.

How it works; using in dialog procedure

All **but** versions can be used everywhere - with windows of other applications and with your custom dialogs.

The 'click' versions (**but** and **but%**) send BM_CLICK message to the button, which sends WM_LBUTTONDOWN etc and sets focus. May not work if the window is inactive, although may activate it. It seems that Acc.DoDefaultAction also uses this message.

The 'check' versions (**but+**, **but-**, **but***) send BM_SETCHECK message to the checkbox. It does not set focus. Works with inactive windows.

but+, **but-** and **but%** do nothing if the checkbox already is in that state.

All 'click' and 'check' versions send notification (WM_COMMAND) to the parent dialog. To check a checkbox or option button in your dialog without notification, use [CheckDlgButton](#) or [CheckRadioButton](#), or assign 1 to the dialog variable before [ShowDialog](#).

Sends messages asynchronously. For example, if the button shows a dialog, **but** does not wait until the dialog is closed. Applies autodelay ([spe \(90\)](#)). If the target window belongs to current thread, sends synchronously and does not apply autodelay.

The 'get checked state' version (syntax 3 and 4) sends BM_GETCHECK message. The alternatives are [IsDlgButtonChecked](#) and BM_GETSTATE. Look in MSDN Library.

Examples

```
but child(100 100 "Calc") ;;click button at 100 100 of "Calc" window
but id(306 "Calc") ;;click button with id=306 on window "Calc"
but+ 306 "Calc" ;;check check-button with id=306 on window "Calc"
but "&Open" "Open" ;;click button "Open" on window "Open"
```

If radio button "Down" in "Find" dialog is unchecked, click it:

```
int h=child("&Down" "Button" "Find")
if(!but(h)) but h
```

Find child window (get handle)

Syntax

```
int id(id [window] [flags])
```

Parameters

id - child window id. Integer. You can see it in QM status bar.

[\(274\)window](#) - parent window. If omitted - active window. The function searches only in the first found matching window.

flags [\(247\)](#): 1 - search only direct children. Default: 0.

Remarks

Finds a [child window \(275\)](#) in the specified window, and returns its handle. Returns 0 if not found. Error if parent window does not exist.

Unlike [child \(83\)](#), at first searches direct children and does not prefer visible children.

The function is often used in [dialogs \(63\)](#), to get control handles. Also can be used with any windows.

Window id, differently than handle, usually is set by the software developer and does not change at run time. It usually is a value between 0 and 65535. A window can have more than one child window with same id. Often such child windows don't receive user input, or they are children of other children. Sometimes id may be set at run time and it may be a random value (usually then it is not in the range 0 - 65535).

See also: [child \(83\)](#) [acc \(85\)](#) [htm \(86\)](#)

Examples

```
int h = id(15 "Notepad")
act id(306 "Calculator")
```

Find child window (get handle); compare properties

Syntax1 - specified child window

```
int child(text [class] [window] [flags] [propCSV] [matchindex|array])
int childtest(hwnd text [class] [window] [flags] [propCSV])
```

Before QM 2.3.4 used this instead.

Instead of **propCSV** used two parameters **x y** and several additional flags.

```
int child(text [class] [window] [flags] [x y] [matchindex|array])
int childtest(hwnd text [class] [window] [flags] [x y])
```

Control id can be specified before text:

```
int child(id [text] [class] [window] [flags] [x y] [matchindex|array])
int childtest(hwnd id [text] [class] [window] [flags] [x y])
```

x y - used for multiple purposes, depending on flags. Default: the child window must contain this point in window.

These flags can be used to change **x y** meaning:

8	x and y are coordinates in window client area.
32	x and y are screen coordinates.
8 32	x and y are coordinates in the work area.
128	x is style.
0x100	y is extended style.
0x8000	x and y used to specify a callback function and a value to pass to it.

Syntax2 - child window from point

```
int child(x y [window] [flags2])
int childtest(hwnd x y [window] [flags2])
```

Syntax3 - child window from mouse position

```
int child(mouse) [flags2]
int childtest(hwnd mouse) [flags2]
```

Syntax4 - child window from accessible object

```
int child(acc)
int childtest(hwnd acc)
```

Syntax5 - the focused child window

```
int child
```

Parameters

text - child window text.

- By default, **text** can be partial and must match case.
- Underlined character must be preceded with &.
- Empty string ("") matches any text.

class - child window class name.

- Must be full or with wildcard characters (*?). Case insensitive.
- Default: "" (any).
- Before QM 2.3.4, to use wildcard characters, need flag 0x800.
- You can see class name in QM status bar.

(274)window - parent window.

- If omitted or "" - the active window.
- The function searches only in the first found matching window.

flags (247):

1	text is full or with wildcard characters (196) (*?). <ul style="list-style-type: none"> String "*" matches child windows with no name.
2	text case insensitive.
4	Get text using an alternative method. Use for Edit and some other controls if gets wrong text.
16	Must be direct child. If not set, the function also finds children of direct children and so on.
0x200	text is regular expression (198) .
0x400	Must be visible.

propCSV - list of other properties in format "name1=value1[]name2=value2[]...". It is [CSV \(116\)](#) string with separator =.
Default: "".

name	value
id	Control id (82) .
accName	Accessible object Name property. Must be full or with wildcard characters; case insensitive. Useful with controls that take accessible name from a sibling Static control.
wfName	Windows Forms (.NET) control name. Must be full or with wildcard characters; case insensitive. Use with controls where class name looks like "WindowsForms10.xxx.xxx". Don't use id, it is not constant. You can see wfName in QM status bar. Also can record.
xy	x and y coordinates of a point (241) in window that must be in the child window too. Relative to the client area of the window. Examples: <pre>"xy=100 50" "xy=0.9 0.1" ;;near top-right</pre>
callback	Address of a callback function (273) , optionally followed by a value to pass to it. Example: <pre>F"callback={&Function} {aValue}"</pre>
	Also supports style, exStyle, cClass, cText, cId, cFlags, GetProp. Same as with win (77) .

matchindex (QM 2.2.0) - 1-based index of matched child window in the parent window. Use when there are several child windows (in the parent window) that match other properties (**text**, **class**, etc).

array (QM 2.2.1) - variable of type ARRAY(int) that will receive handles of all matching child windows.

x, y - a [point \(241\)](#) in the parent window or screen (if **window** is 0 or used flag 32).

- The function gets the visible child window that contains this point.

flags2 (247) (syntax 2 and 3):

1	QM 2.3.4. If there is no child window, get the top-level window.
8	x and y are coordinates in client area of window.
16	Must be direct child.
32	x and y are screen coordinates.
8 32	x and y are coordinates in the work area.

- QM 2.3.4. **flags2** also can be used with syntax3 (mouse). Only 1 and 16.

mouse (syntax 3) - literal `mouse`.

accobj (syntax 4) - [accessible object \(85\)](#) (an **Acc** or **IAccessible** variable).

hwnd ([childtest](#)) - handle of child window to test.

Remarks

Finds a [child window \(275\)](#) (control) in the specified window, and returns its handle. Returns 0 if not found. Error if parent window does not exist.

Syntax1: Finds child window whose text and other properties match the specified.

Syntax2: Gets handle of child window from the specified point in its parent window. If **window** is omitted or "" - active window. If **window** is literal 0, parent can be any window (**x** and **y** are screen coordinates).

Syntax3: Gets handle of child window (except invisible) from mouse pointer position.

Syntax4: Gets handle of child or top-level window that matches or contains accessible object **accobj**.

Syntax5: Gets handle of child window that has focus. To get focused window of current thread, use [GetFocus](#) instead.

Function [childtest](#) compares child window (**hwnd**) properties with the specified, and returns 1 if they match, or 0 if not. If

window is omitted or literal 0, does not compare it. See also [WinTest \(114\)](#).

See also: [id \(82\)](#) [acc \(85\)](#) [htm \(86\)](#)

Examples

```
act child("Hex" "Button" "Calculator")
if child("The text*" "Static" "Notepad" 1)
  _bee
int h = child(100 100 "Notepad")
h = child(mouse)
if(childtest(h "OK" "Button")) but h
```

Accessible objects

See also: [acc \(85\)](#)

You can work not only with windows, but also with user interface (UI) objects that are in windows. Most of them are child windows ("controls") and can be identified by handle. You can use function [id \(82\)](#) or [child \(83\)](#) to retrieve child window handle. But not all objects are child windows. For example, links and other objects in web pages are not child windows, and functions [id](#) and [child](#) cannot find them. The alternative is accessible objects.

Accessible objects and Acc class

Accessible objects are user interface objects (buttons, links, etc) that can be manipulated (find, click, etc) through [COM \(162\)](#) interface [IAccessible](#). The service is provided by MSAA (Microsoft® Active Accessibility®).

To store a reference to an accessible object, QM uses class [Acc](#).

```
class Acc IAccessible 'a elem
```

a - pointer to object's [IAccessible](#) interface.

elem - child element id. With most objects it is 0. Used with *simple element* objects, such as list items.

A variable of class [Acc](#) fully identifies an accessible object, regardless of whether it is full object or simple element.

Functions

To find an object and initialize an [Acc](#) variable, use function [Acc.Find](#) or [acc \(85\)](#). To create code for it, use dialog 'Find Accessible Object'.

The [Acc](#) class has functions to manipulate an object. To create code for them, you can use dialog 'Accessible Object'.

The common syntax is:

```
[result = ]a.Function([parameters])
```

Example:

```
----
int w=win("QM TOOLBAR" "QM_toolbar")
Acc a.Find(w "PUSHBUTTON" "Windows, controls" "class=ToolbarWindow32[id=9999" 0x1005)
err ret ;;return if not found. If without flag 0x1000, use this instead: if(!a.a) ret
str name=a.Name
a.DoDefaultAction
...
```

On failure, the functions throw error, therefore you should understand [error handling \(129\)](#). Note that different kinds of accessible objects support different sets of functions. You should call only supported functions, or be prepared to handle error.

Also you can use functions of [IAccessible](#) interface. Call them through member **a** of [Acc](#) variable. Use **elem** where needed. Example:

```
Acc a.Find("Quick Macros" "MENUITEM" "Tools" "class=ToolbarWindow32")
out a.a.KeyboardShortcut(a.elem)
```

'Find Accessible Object' dialog tips

There are 2 ways to capture an object:

1. Drag the "Drag" tool, and drop on an object. While dragging, you can right-click to switch to another window.
2. Right click the "Drag" tool, click "Capture when Shift pressed", move the mouse to an object, and press Shift key.

After capturing, click the Test button. If it says "Object not found", try to change some values in the dialog.

84. Accessible objects

If it finds wrong object, try to capture an adjacent object. Test it, check Navigate, and enter post-navigation string. Read more about **navig** parameter of [acc \(85\)](#). Example: pa n2 c15 f. Explanation: get parent object, get next object two times, get 15-th child object, get first child object.

If the search time is too long, try to check 'in reverse order' and/or 'as Firefox node'.

The coordinates (x, y) are in the client area of the window.

At the bottom of the dialog - object tree. It shows accessible objects that are in the window. Shows their properties and relationships. By default, it does not show invisible and useless objects. Invisible objects are gray.

To show the dialog you also can use the Ctrl+Alt+Shift+W menu. You can change the hotkey in Options. If the currently active window has always-on-top style, the dialog also will be always-on-top.

To capture objects in Windows 8 full-screen windows, make the dialog always-on-top (open with Ctrl+Alt+Shift+W or right click the "Drag" tool). QM should be running as Administrator or uiAccess, not as User; you can set it in Options. Or capture with Shift.

Using in web pages

Accessible objects in web pages are supported in Internet Explorer, Firefox, Chrome and Opera.

In Chrome and Opera, accessible objects are disabled by default. QM tries to enable when need. If it fails, use one of:

- In QM Options dialog check "Enable Chrome acc when it starts". It works only if Chrome process starts while QM is running.
- Start Chrome with command line --force-renderer-accessibility
- You cannot use the first option in exe where QM is not running. But you can [implement it in exe](#).

Firefox accessible objects are disabled by default. Firefox enables it when some program tries to get its accessible objects. May need to restart Firefox, or in Firefox Options disable multi-process mode.

Note: enabling web browser's accessible objects can make large web pages load slower.

Note: From time to time, web browsers introduce new features or bugs that break some QM functions. If something stopped working, you can: download new QM version; post a bug report in the [forum](#). When it is fixed in QM, it is not automatically fixed in QM-created exe files; need to rebuild them with the new QM version.

Web browsers (and other apps) have bugs in their accessible object implementations. For example, can give incorrect locations of some objects. Or you cannot capture some objects because they are "behind" bigger objects; then try to capture two times: at first capture normally, then check 'Capture smallest' in the Drag tool's right-click menu, and capture again.

There are two alternatives, often faster:

- With Internet Explorer and IE-based browsers you can use function [htm \(86\)](#) (find html element).
- With Firefox, check 'as Firefox node' in the 'Find accessible object' dialog. Chrome and Opera also support it, if is installed Firefox of same 32/64 bit.

When none of the above works, use [scan](#) (find image).

In current Opera versions everything is the same as in Chrome.

Java windows

QM 2.4.2. [acc](#) and [Acc](#) class functions support accessible objects in applications written in Java that don't use Windows controls, including OpenOffice, LibreOffice, NetBeans, jEdit and Java in Control Panel.

Prerequisites (except OpenOffice and LibreOffice):

- Java version 7.6 or later.
- Enable the Java Access Bridge (JAB). It is disabled by default.
- For older Java versions (<7.6) would need to download JAB separately. Installing it is not easy.

To enable JAB, run function [JavaEnableJAB](#) (once on a computer/account). Or run jabswitch.exe -enable. Or enable in Control Panel -> Ease of access (not on XP). Then restart Java applications. When installing Java, may need to restart QM too.

Current OpenOffice and LibreOffice versions don't use JAB. For older versions need JAB, and also check: Options ->

Accessibility -> Support assistive technology tools.

If only 64-bit Java is installed, JAB does not work by default because of missing file C:\Windows\SysWOW64\WindowsAccessBridge-32.dll. To fix it, also install 32-bit Java (it coexists with 64-bit Java). Or somewhere get WindowsAccessBridge-32.dll and put in SysWOW64 or QM folder. Only JAB 2.0.2 (old version) can be downloaded separately from Java.

Some JAB features are unstable. Getting focused object often stops working. Getting object at x y does not work in some windows. To capture objects in such windows, in dialog 'Find accessible object' right click the Drag image and check 'Capture smallest'.

QM does not implement accessible object triggers in Java windows that use JAB.

Find [accessible object \(84\)](#); compare properties

Note: In QM 2.3.3 and later, there are functions of Acc class that replace acc/acctest. They are simpler to use. Use acc/acctest if you want to make the macro compatible with older QM versions. Also, here you can find more info than in Acc.Find help (it calls acc).

Syntax1 - get specified object

```
Acc acc(name [role] [window] [class] [value] [flags] [x y] [navig] [waitS] [matchindex])
int acctest(object name [role] [window] [class] [value] [flags] [x y])
```

Syntax2 - get object from point

```
Acc acc(x y [window] [flags2])
int acctest(object x y [window] [flags2])
```

Syntax3 - get object from mouse pointer position

```
Acc acc(mouse)
int acctest(object mouse)
```

Syntax4 - get object from window handle

```
Acc acc(hwnd)
int acctest(object hwnd)
```

Syntax5 - get focused object

```
Acc acc
int acctest(object)
```

Syntax6 - get adjacent object

```
Acc acc(object navig)
```

Syntax7 - get object from html element

```
Acc acc(html element)
```

Parameters

name - object name.

- By default, **name** can be partial. Case insensitive.
- Empty string ("") matches any name.
- Usually it is text that is displayed in the object or before the object. Not all objects have name.

role - object type. Supported are all role types that are returned by [Acc.Role](#):

- Standard role constant (numeric), eg `ROLE_SYSTEM_PUSHBUTTON`.
- String that matches the part that follows `ROLE_SYSTEM_`, eg `"PUSHBUTTON"`.
- Role string like `"push button"`.
- Default: 0 or "" (any).

[\(274\)window](#) - container window (top-level or child).

- If omitted or "" - the active window.
- Searches only in the first found matching top-level window.
- With [acc](#) it also can be an accessible object (variable of type [Acc](#) or [IAccessible](#)).

class - class name of direct container window. Usually it is a [child \(275\)](#) window of the **window**.

- Must be full or with wildcard characters (QM 2.3.4). Case insensitive.
- Also can include id. Full syntax is `"[id=digits] [classname]"`, where digits is control id.
- Default: "" (any).
- Not used if **window** is accessible object.

value - object value.

- For example, value of an edit control is it's text; value of a link is its URL; value of an outline (treeview) item is it's level within the hierarchy. Most objects don't have a value.
- By default, **value** can be partial. Case insensitive.
- Empty string ("") matches any value.

[flags \(247\)](#):

1	name is full or with wildcard characters (196) (*?). <ul style="list-style-type: none"> String "" matches objects without name. Without this flag, name can be partial. This flag cannot be used with flag 2.
2	name is regular expression (198) . <ul style="list-style-type: none"> Case sensitive, unless contains (?i).
4	value is full or with wildcard characters. <ul style="list-style-type: none"> String "" matches objects without value.
8	value is regular expression. <ul style="list-style-type: none"> Case sensitive, unless contains (?i).
16	Search invisible objects too (and their descendants).
32	Search useless objects too (and their descendants): scrollbar, grip, invisible titlebar, separator.
64	Search only direct children. <ul style="list-style-type: none"> Can be useful when window is accessible object.
128	Search in reverse order (starting from bottom). <ul style="list-style-type: none"> Applied only to objects in the class child window. Finds the child window not in reverse order. This flag is not used with acctest.
0x1000	Error if object not found. <ul style="list-style-type: none"> This flag is not used with acctest.
0x2000	QM 2.3.3. Search only in web page, the active tab. <ul style="list-style-type: none"> Makes acc faster. class is ignored. window must be the browser window. Cannot be a control or accessible object. This flag can be used with IE, IE-based browsers, Firefox, Chrome, Thunderbird. This flag is not used with acctest.
0x4000	QM 2.3.3. value is CSV containing various properties. Documented in function Acc.Find .
0x10000	QM 2.4.3. This flag removes a 1-2 s delay in some web pages in Firefox. <ul style="list-style-type: none"> Use with pages where even finding DOCUMENT takes 1-2 s. It happens if the DOCUMENT always has "busy" state. Only with flag 0x2000. For reliability also use waitS (see below).

Obsolete flags

These flags are not used with [Acc](#) class functions. Now these properties can be specified in CSV (see flag 0x4000).

0x100	x and y are coordinates in window client area. <ul style="list-style-type: none"> Ignored if window is accessible object. Then must be screen coordinates.
0x200	x and y are coordinates in the screen.
0x300	x and y are coordinates in the work area of the screen.
0x400	value is description.
0x800	Check object's state. <ul style="list-style-type: none"> x must be state. y must be state mask (state flags that must match must be 1, others - 0). If y is -1 (0xFFFFFFFF), state must match exactly. Object state constants are documented in MSDN (256). Example: STATE_SYSTEM_READONLY.
0x8000	Use callback function (273) .

x, y - the object must be at this [point \(241\)](#) in the window. Default: 0 0 (any).

- For syntax1, it must be top-left corner coordinates. For syntax2, [acc](#) gets object that contains the point.
- Obsolete with syntax1. Now can be specified in CSV (see flag 0x4000).

navig - post-navigation string. Read in Remarks.

waitS - time (seconds) to wait for the object in the window.

matchindex (QM 2.2.0) - 1-based index of matched object in the window. Use when there are several objects (in the window) that match other properties (**name**, **role**, **class**, **value**, **flags**, **x**, **y**).

[flags2 \(247\)](#) - combination of values listed below. Default: 0.

1	x and y are coordinates in client area of window.
2	x and y are coordinates in the screen.
3	x and y are coordinates in the work area of the screen.

hwnd - handle of some window (child or top-level).

mouse - literal `mouse`.

object - accessible object. Variable of **Acc** type.

htmlelement - [html element \(86\)](#). Variable of **Htm** or **IHTMLElement** type.

Remarks

Syntax1: Finds accessible object whose name and other properties match the specified properties.

Syntax2: Gets object from the specified point in the window. If **window** is omitted or "" - active window. If **window** is literal 0 - any window (**x** and **y** are screen coordinates).

Syntax3: Gets object from mouse pointer position.

Syntax4: Gets accessible object of window (**hwnd**) itself. Window can be child or top-level. To get window handle from accessible object, use [child \(83\)](#) function.

Syntax5: Gets the focused accessible object.

Syntax6: Gets adjacent object.

Syntax7: Gets accessible object that matches html element **htmlelement**.

acc should be assigned to a variable of type [Acc \(84\)](#). If object not found, the variable will be empty. You can use code like `if (a.a=0) out "not found"`. Or use flag 0x1000 (error if not found) and [err \(129\)](#). Always error if window does not exist.

acctest compares accessible object properties with the specified, and returns 1 if they match, or 0 if not. If **window** is omitted or literal 0, does not compare it.

To capture accessible objects and create code, use the "Find Accessible Object" dialog. Starting from QM 2.3.3, it inserts code like `Acc a.Find(...)`. The Find function calls **acc**, therefore all its parameters are similar.

The **navig** string can be used to get an adjacent object. Let's call it *post-navigation*. After **acc** finds the specified object, it navigates from the found object to another object according to the **navig** string. In the string you can use one or more of the following words or abbreviations:

<i>up, down, left, right</i>	Spatial navigation. Most objects don't support it.
<i>next, previous</i>	Next or previous sibling.
<i>parent</i>	Parent.
<i>first, last</i>	First or last child.
<i>child</i>	Child x. To specify child object, append 1-based child index.

Each word can be followed by a number that specifies the number of navigation operations in the specified direction. For example, use "next3" instead of "next next next".

Post-navigation is useful when the wanted object cannot be uniquely identified (does not have name, etc). Then you can find an adjacent object, and use **navig** to get the object you actually need. Post-navigation also can be used to reduce search time. To view relationships between objects, use the "Find accessible object" dialog. Navigation does not always work in all directions. Sometimes, invisible objects are skipped (except with *parent* and *child*). Example of **navig** string: "pa n2 c15 f" (get parent, get next object two times, get 15-th child, get first child).

acc is much slower than similar functions **id** and **child**. If window or web page has stable structure (hierarchy), post-navigation can significantly reduce search time. You can specify some object in top of hierarchy (e.g., PANE or DOCUMENT object in web page), and use post-navigation to navigate to the object you need.

acc is mostly used with web pages. There are two alternatives that often are faster. With Internet Explorer and IE-based browsers you can use function [htm \(86\)](#) (find html element). With Firefox and Chrome try to check 'as Firefox node' in the 'Find accessible object' dialog.

Example

```
Acc a=acc("Google Search" "PUSHBUTTON" " Internet Explorer" "Internet Explorer_Server" ""  
0x1001)  
a.DoDefaultAction
```

Find html element, or get document interface

Syntax1 - find html element

```
IDispatch htm(tag name [html] [window] [frame] [index] [flags] [waitS] [navig])
```

Syntax2 - get IHTMLDocument2 of web page

```
IDispatch htm(hwnd)
```

Syntax3 - get html element from accessible object

```
IDispatch htm(acc)
```

Parameters

tag - tag. Example: "A".

name - text. Depending on flags, it also can be some attribute. Can be "".

html - outer html. Example: "Name". Default: "" (any).

window - window with web page ("Internet Explorer_Server" control) that contains this element.

- If omitted or 0, searches in first (in the [Z order](#)) visible Internet Explorer window (class "IEFrame") or other IE-compatible browser window (class must be specified with [RtOptions \(114\)](#)).

frame - 1-based index of frame/iframe. Or path to it, like "2/3".

- If "0", the element must be not in frame/iframe.
- Default: "" (the element can be anywhere).
- In multi-frame pages, **frame** makes **htm** faster.

index - index of this element in collection of elements of **tag** type.

- **index** makes **htm** much faster.
- **index** may not match exactly (e.g., after page structure has changed), but then **htm** is slower.
- If **tag** is "", **index** is index in collection of all elements.
- Default: 0.

flags (247):

1	name is full or with wildcard characters (196) (*?). <ul style="list-style-type: none"> • "" matches elements with no name. • Without this flag, name can be partial. • This flag cannot be used with flag 2.
2	name is regular expression (198) .
4	html is full or with wildcard characters. This flag cannot be used with flag 8.
8	html is regular expression.
16	window is handle of "Internet Explorer_Server" child-window. Default: main window. If not set, the function searches only in the first visible "Internet Explorer_Server" control.
32	Error if not found.
0x100-0xA00	name is attribute: 0x100 - id, 0x200 - name, 0x300 - alt, 0x400 - value, 0x500 - type, 0x600 - title, 0x700 - href, 0x800 - onclick, 0x900 - src, 0xA00 - classid.

[\(274\)](#) **waitS** - time (seconds) to wait for the element in the window.

navig - a positive or negative number that can be used to get an adjacent element.

hwnd, acc - read in Remarks.

Remarks

Finds a html element.

Returns pointer to MSHTML.IHTMLElement interface (except Syntax2), or 0 if not found. The return value can be assigned to a variable of type Htm. The variable represents a html element, and is used to manipulate it (click, get text, set text, get other interface, etc).

A html element is an object in a web page (link, button, text, etc). Htm elements is faster and more precise alternative to accessible objects. However, this function works only with Internet Explorer and other windows that display web page in "Internet Explorer_Server" control.

Syntax1

Finds html element.

Searches only in the first found matching window, and only in the first found visible "Internet Explorer_Server" control that displays a web page. To search in other windows or controls (for example in hidden tabs), find the window ([win \(77\)](#)) or control ([child \(83\)](#)) and pass the window handle to [htm](#) as **window**.

Syntax2

Returns pointer to MSHTML.IHTMLDocument2 interface of web page in window **hwnd**. If **hwnd** is literal 0 - in first matching window. The interface represents web page, which is container of html elements.

Tips: To get html document from html element, call element's *document* property. To get MSHTML.IHTMLWindow2, call document's *parentWindow* property.

Syntax3

Gets html element that corresponds to accessible object **acc**.

Tip: Function [acc \(85\)](#) can be used for backward conversion.

Tip: To get html element from certain coordinates, use [acc \(85\)](#): `Acc am=acc(x y hwnd); Htm hm=htm(am)`. It works with some elements (e.g. links, images, input) but does not work with others (e.g. simple text).

Dialogs

To insert the [htm](#) function, use the Find Html Element dialog. Drag the picture and drop onto an object in a web page. This fills the dialog with values that are optimized for best performance.

If the web page is frequently updated, element's index may change, and for some elements this may cause the function to find another similar element. To minimize this possibility, you should test (press the Test button) with different *Index*. Test with index 0 and index 10000. If then will be found different element, try to change some parameters, or find an adjacent element and use *Navigate* (positive or negative number) to navigate to the desired element. Also, if index does not match, the search time is longer. It is as longer as the count of elements in the *Tag* collection is larger, so you may consider to find an element that belongs to a tag with less number of elements, and then navigate.

In the dialog, the *Index in all* field is index of element in collection of all elements in page or frame. Normally, it isn't used with [htm](#) function. Click arrows to view adjacent elements. Or, type another index and press ! (update).

To manipulate a html element (click, get/set text, etc), use the Html Element dialog. Alternatively, type `.` after variable name, and select a function from the drop-down list.

Example

```
Htm el=htm("INPUT" "id" "" "Internet Explorer" 0 0 0x221)
el.Click
or
el.el.click
or
MSHTML.IHTMLElement el=htm("INPUT" "id" "" "Internet Explorer" 0 0 0x221)
el.click
```

Find image on screen

Syntax1 - find

```
int scan(image [window] [rect] [flags] [colorDiff] [matchIndex|array])
```

Syntax2 - wait

```
wait timeS [-]S image [window] [rect] [flags] [colorDiff] [matchIndex|array]
```

Parameters

image - image to find. Can be:

- QM 2.4.0. Image [resource \(261\)](#) name. Format: "image:name" or "resource:<macro>image:name". Read more in Remarks.
- Bitmap [file \(246\)](#). Must end with ".bmp".
- Icon file. Any file path that does not end with ".bmp". Icon index can be specified, like "shell32.dll,5".
- QM 2.3.2. [Color \(240\)](#) of a single pixel. Format: "color:0xBBGGRR". To use a variable: F"color:{variable}".
- QM 2.4.3. Bitmap or icon handle. Format: "hbitmap:handle", "hicon:handle". To use a variable: F"hbitmap:{variable}".
- QM 2.4.3. Can be a list of images or colors, like "image:image1[]image:image2[]image:image3". Read more in Remarks.
- [Other options, now obsolete.](#)
 - QM 2.3.2. Macro containing bmp file data. Format: "macro:name". Obsolete, use macro resources instead.
 - Bitmap or icon handle. Now instead you can use F"hbitmap:{h}" or F"hicon:{h}".

[\(274\)window](#) - top-level or child window where to search.

- If omitted or literal 0, searches in whole screen, all monitors.
- QM 2.4.3. Can be an accessible object (an [Acc](#) or [IAccessible](#) variable).
- See also flag 0x200.

rect - variable of type [RECT](#). Default: 0.

- In the variable [scan](#) stores image coordinates in screen (relative to the top-left of the primary monitor).
- Also used to limit the search area, unless the variable is empty or used flag 128. Read more in Remarks.

flags (247):

1	Move mouse to the image. This flag not used with flag 0x200 and -S.
2	Error if image not found. This flag not used with wait .
4	Use the top-left pixel color of the image as transparent color.
16	Search only in the client area of the window. This flag is not used when searching in whole screen or bitmap. When window is an accessible object, this flag is useful with flags 0x100 and 0x1000.
32	When image is icon file, use large icon (32x32). Default - small icon (16x16).
64	If image is handle or exe resource id, use this flag to specify that it is icon.
128	QM 2.3.2. Use rect for results, but not to limit the search area.
0x100	QM 2.3.2. Get pixel colors not from screen but directly from window, with function PrintWindow . Notes: <ul style="list-style-type: none"> • Works even if the window is in background (covered by other windows) or offscreen, but not when minimized or hidden. • The window may flicker, especially when waiting. The speed depends on window. • With this flag scan cannot find images in some windows (including Windows store apps), and in some window parts (nonclient, glass). Error if the window is of a higher integrity level (277) process. • This flag is ignored: 1. If the window is DPI-scaled (243). 2. With flag 0x1000 if Aero is enabled.
0x200	QM 2.3.2. Search in bitmap stored in memory. Then window must be bitmap handle. If rect used, it receives coordinates of the found image in the bitmap; it is not used to limit the search area.
0x400	QM 2.3.2. When found, wait until mouse buttons released. It is more useful with wait and flag 1 (move mouse).
0x800	QM 2.4.3. Find all images specified in images . Return 1 if all found, else return 0 (or error, if flag 2).
0x1000	QM 2.4.3. Use a fast and flicker-free method to get pixel colors directly from window. Notes: <ul style="list-style-type: none"> • Works only if Windows Aero theme is enabled. Else this flag is ignored. Also it is ignored if the window is DPI-scaled (243). • Works even if the window is in background (covered by other windows) or offscreen, but not when

minimized or hidden.

- This flag makes **scan** faster when Aero is enabled (Aero makes it slower). The speed depends on window and control.
- With this flag **scan** cannot find images in some windows (including Windows store apps), and in some window parts (nonclient, glass).
- On Windows XP Aero is unavailable. On Windows Vista/7 it can be enabled or disabled. On Windows 8/10 it is always enabled (now it is not called "Aero" but is used the same technology).

colorDiff (QM 2.3.2) - maximal allowed color/brightness difference of the on-screen image. Allows to find images that don't match exactly. Can be 0-255, but should be as small as possible.

matchIndex (QM 2.3.2) - 1-based index of matched image. 0 is same as 1. Use when there are several matching images in the search area and you need not the first one.

array (QM 2.3.3) - variable of type **ARRAY(RECT)** that will receive coordinates of all matching images.

timeS - max time (seconds) to wait. Error on timeout. If ≤ 0 , waits infinitely.

S - literal S.

- If -S, waits until image disappears.
- If -S, and **image** is "", waits until something changes in the window or rectangle.

Remarks

scan searches for the specified image on the screen. Returns 1 if found, 0 if not (or throws error if flag 2 is set).

wait waits for the specified image on the screen. Almost everything is the same as with **scan**. In the following text, "**scan**" or "the function" means "**scan** and **wait**".

To capture/save an image and create code, use the 'Find image' dialog. The captured image can be saved in macro resources or in a .bmp file.

By default the dialog saves the captured image as a **resource (261)** of current macro, with name like "image:hFBDB67A2" or "image:button1" (if button1 specified in the dialog). The resource name is used in macro, like `scan "image:button1" 0 0 1|2`. When the macro runs, **scan** gets the image from resources of its macro, or from resources of a caller in the function call stack.

You can also select an existing resource. Supported are resources with names that begin with "image:" or end with ".bmp", ".png" or ".ico". You can use resources of other macros too, like `scan "resource:<macro1>image:button1" 0 0 1|2`.

When creating **exe (51)**, QM adds macro resources to exe resources, and **scan** in exe uses these exe resources. If in exe you use image files instead, the files must be available where exe runs. Or you can [add files to exe resources](#).

You can add bitmap and icon files to exe resources:

1. In "Make exe" dialog check "Auto add files..."
2. In code use resource id in file path.

Assume there are 3 scan with 2 images:

```
scan "A.bmp" 0 0 1|2
...
scan "B.bmp" 0 0 1|2
...
scan "B.bmp" 0 0 1|2
```

Add resource id:

```
scan ":1 A.bmp" 0 0 1|2
...
scan ":2 B.bmp" 0 0 1|2
...
scan ":2 B.bmp" 0 0 1|2
```

Then A.bmp and B.bmp will be added to exe resources. In exe will be used the resource images. If the macro runs in QM, it uses file (ignores the ":1 " part).

QM 2.4.3. Multiple images or colors can be specified. Then **scan** will search for any or all of them:

- Without flag 0x800 searches for any image and returns 1-based index of the first found image. If finds the first image,

returns 1, else if finds the second, returns 2, and so on. If all not found, returns 0 (or error, if flag 2). If **array** used, stores coordinates of all found instances of all images in it. The array may not match the specified list because for each image may find 0, 1 or more instances.

- With flag 0x800 searches for all images and returns 1 if all found, else 0 (or error). If **array** used, stores coordinates of each first found image in it (the array will match the specified list).
- Example: `scan "image:h43312EF0[]image:h65F5BD95[]image:h2FC19D26" w 0 1 | 2 | 16`

You can make `scan` faster by limiting the search area. Always specify **window**. Even better if it is a control or accessible object. To measure the search time, press Test in the 'Find Image' dialog.

To limit the search area you also can use **rect** (variable of type **RECT**). The rectangle specified in **rect** must be relative to the search area defined by **window** (screen, window/control, window/control client area (flag 16) or accessible object).

If you use **rect** for results but don't want to limit the search area, the **RECT** variable must be initially empty (all members 0). Or use flag 128.

The speed also depends on computer hardware, Windows version, whether Aero is enabled and whether the correct video driver is installed. With flags 0x100 and 0x1000 depends on window.

The function can find only images that are visible on the screen, unless used flag 0x100, 0x1000 or 0x200. If **window** used, make sure that the window is not covered by other windows. The function does not test whether the image belongs to the window. You can use `act` to activate the window.

The function can only find on-screen images that exactly match the specified image. If you use **colorDiff**, it can find images with slightly different colors and brightness. It cannot find images with different shapes.

The function may fail to find image after you change Windows theme, color scheme, skin, etc, because the background color may change. To make the function independent from such changes, try flag 4. Then it does not compare pixels that have the same color as the top-left pixel of the captured image. Usually it is background color, so if background will change in the future, the function will not fail. The function also may fail after changing: display color resolution; [DPI \(243\)](#) (text size); font smoothing (if the image contains text). Also be aware of window shadows.

With icons you should always use **colorDiff**. Recommended 8. Because icons often are displayed with slightly different colors than in the icon file.

image also can be a bitmap or icon handle, ie when the bitmap or icon is loaded in memory. These functions can be used to get handle: [LoadImage \(256\)](#), [LoadPictureFile \(114\)](#), [GetFileIcon \(114\)](#), [GetWindowIcon \(114\)](#), [CaptureImageOnScreen](#), [CaptureImageOrColor](#). Later use [DeleteObject](#) for bitmap or [DestroyIcon](#) for icon.

QM 2.3.2: If flag 0x200 used, the function searches in bitmap whose handle is passed as argument 2 (**window**). The bitmap can be loaded using [LoadPictureFile \(114\)](#) or other function (see above), and later must be deleted using [DeleteObject](#). Must not be selected into a device context. Searching in bitmap is faster because the slowest part usually is getting pixels from screen or window.

Error if:

- file not found, or resource does not exist, or image cannot be extracted.
- window not found.
- object (image) not found. Only if flag 2 is set.
- failed (an unexpected error, such as invalid handle or not enough memory).
- timeout (with `wait`).

The **RECT** type is used to specify a rectangle.

```
type RECT left top right bottom
```

left and **top** - top-left corner coordinates.

right and **bottom** - bottom-right corner coordinates. This point actually is outside the rectangle. Rectangle width is **right-left**. Height is **bottom-top**.

See also: [wait for image or color \(89\)](#), [pixel \(105\)](#), [id \(82\)](#), [child \(83\)](#), [acc \(85\)](#), [htm \(86\)](#).

Examples

Find image (stored in resource) in "Abc" window. If found, click, else error:

```
scan "image:example" "Abc" 0 1 | 2
```

```
lef
```

Wait for image (stored in file) max 10 s:

```
wait 10 S "example.bmp" "Abc"
```

Find image in window, and get its location:

```
RECT r
if(scan("image:example" "Abc" r))
_out "x=%i y=%i width=%i height=%i" r.left r.top r.right-r.left r.bottom-r.top
else out "not found"
```

Find image in specified rectangle area, and get its location:

```
RECT r; r.left=100; r.top=100; r.right=300; r.bottom=300
if(scan("image:example" 0 r))
_out "x=%i y=%i width=%i height=%i" r.left r.top r.right-r.left r.bottom-r.top
else out "not found"
```

Wait

See also: [wait for \(89\)](#)

Syntax

```
[wait ]timeS
```

Parameters

timeS - time (seconds) to wait. Can be integer (like `10`) or double (like `0.1`). Special values:

0 or omitted	Just process messages that possibly are waiting in thread's message queue, regardless of opt waitmsg. (97)
-1	Infinite.
-2	Autodelay (90) . The wait time will be equal to the speed of the macro or function that directly called (151) the current function. If there is no direct caller - default macro speed.

Remarks

Waits **timeS** seconds.

If **timeS** is simple digits, or begins with a digit, the `wait` keyword can be omitted. See examples.

The wait time precision is about 2 milliseconds (0.002 s). For example, `wait 0.001` will probably wait 2 ms.

Note: If the thread has windows/dialogs or uses COM events or some Windows API functions that use Windows messages, you should avoid wait commands in it. Or use small time (<0.1). By default, messages are not processed while waiting, and your thread may temporarily hang. To process messages while waiting, use [opt waitmsg \(97\)](#).

Note: Don't use Windows API wait functions, because then QM cannot properly end the thread on user request. QM then terminates thread, which causes a big memory leak and possibly more bad things. Also, `opt waitmsg` is not applied.

QM 2.4.0. `wait` uses alertable waiting. For example, [QueueUserAPC](#) can be used to run a function in the waiting thread.

Examples

```
Wait 0.5 seconds:
wait 0.5
or:
0.5

Wait 15 seconds; the wait time is variable wt:
int wt=15
wait wt

Wait 2 seconds:
double F=0.5
4*F
```

Wait for

See also: [wait \(88\)](#)

Events

[Wait for window](#)
[Wait for window name](#)
[Wait for key](#)
[Wait for mouse](#)
[Wait for CPU](#)
[Wait for handle](#)
[Wait for multiple handles](#)
[Wait for variable](#)
[Wait for web page](#)
[Wait for cursor \(mouse pointer\)](#)
[Wait for color](#)
[Wait for image on screen](#)
[Wait for other events](#)

Syntax (common to all events)

```
[wait /timeS event [...]]
```

Can be used as function:

```
int wait(timeS event [...])
```

Parameters

timeS - max time (seconds) to wait. Integer (e.g. 5) or double (e.g. 0.5).

event - one of literals described below (WA, etc).

... - more parameters. Depends on **event**.

Remarks

If **timeS** ≤ 0, waits infinitely.

If **timeS** > 0, waits maximum **timeS** seconds. Error if the event does not occur within **timeS**. To continue macro, use [err \(129\)](#).

The wait time precision depends on event. For key, mouse and H/HM events, it is 0 - 10 ms. For others - from 20 to 300 ms.

Can be used as function, e.g. `variable=wait(...)`. The return value depends on event. If not specified, returns 1. If [opt waitmsg \(97\)](#) is set, returns 0 if the thread receives WM_QUIT message.

Note: If the thread has windows/dialogs or uses COM events or some Windows API functions that use Windows messages, you should avoid wait commands in it. Or use small time (<0.1). By default, messages are not processed while waiting, and your thread may hang. To process messages while waiting, use [opt waitmsg \(97\)](#).

Note: Don't use Windows API wait functions, because then QM cannot properly end the thread on user request. QM then terminates thread, which causes a big memory leak and possibly more bad things. Also, opt waitmsg is not applied.

QM 2.4.0. [wait](#) uses alertable waiting. For example, [QueueUserAPC](#) can be used to run a function in the waiting thread.

Wait for window

```
[wait /timeS event window
```

event:

WA	Wait until window is active, visible and not minimized. <ul style="list-style-type: none"> • If window does not exist, at first waits until it is created. • Returns window handle. • WA is optional. • See also: run (64).
----	---

-WA or WN	Wait until window is deactivated or destroyed. <ul style="list-style-type: none"> Does not wait if window is already inactive or does not exist. Returns handle of the active window.
WC	Wait until window is created. <ul style="list-style-type: none"> Does not wait if window already exists. Returns window handle.
-WC or WD	Wait until window is destroyed. <ul style="list-style-type: none"> Does not wait if window does not exist. Returns handle of the active window.
WP	Wait until window's program exits. <ul style="list-style-type: none"> Does not wait if window does not exist. Returns handle of the active window. See also: run (64).
WV	Wait until window is visible. <ul style="list-style-type: none"> If window does not exist, at first waits until it is created. Returns window handle.
-WV	Wait until window is hidden or destroyed. <ul style="list-style-type: none"> Does not wait if window does not exist. Returns window handle. If window does not exist, returns 1.
WE	Wait until window is enabled. <ul style="list-style-type: none"> If window does not exist, at first waits until it is created. Returns window handle.
-WE	Wait until window is disabled or destroyed. <ul style="list-style-type: none"> Does not wait if window does not exist. Returns window handle. If window does not exist, returns 1.
WAI	Wait until one of windows is active, visible and not minimized. <ul style="list-style-type: none"> window is list of windows, like <code>"Notepad[]WordPad"</code>. Returns 1-based window index in the list.

window - window. Can be child window.

Examples

Wait for window "Notepad" max. 5 seconds; on timeout error:

```
wait 5 WA "Notepad"
or (WA can be omitted):
wait 5 "Notepad"
or (wait can be omitted):
5 WA "Notepad"
or (WA and wait can be omitted):
5 "Notepad"
```

Wait for window "Notepad" max. 5 seconds; on timeout continue:

```
wait WA 5 "Notepad"; err
```

Wait max 5 s for window "Notepad"; on timeout run macro "Open":

```
wait 5 WA "Notepad"
err
_mac- "Open"
```

Wait until window "Notepad" is closed:

```
wait 0 -WC "Notepad"
```

Wait until window with name "Calc" and class "SciCalc" is created:

```
wait 0 WC win("Calc" "SciCalc")
```

Wait until control with id 1500 in window "Window" becomes enabled:

```
wait 0 WE id(1500 "Window")
```

Wait for window "Notepad" max. 5 seconds; store window handle into variable:

```
int hwnd = wait(5 WA "Notepad"); err
```

Wait until currently active window is deactivated (another window becomes active); return handle of another window:

```
int hwnd = wait(5 -WA "")
```

Wait for message "Error" or "Warning" that belongs to "APP" program:

```
int hwnd = wait(5 win("Error[]Warning" "#32770" "APP" 16))
```

Wait for window name

```
[wait] timeS event handle name [flags]
```

event:

WT	Wait until window name is name .
-WT	Wait until window name is other than name .

handle - window handle.

name - name to wait for. Can be partial.

flags (247):

1	name is full or with wildcard characters (196) (*?).
2	name is regular expression (198) .
4	name case insensitive.

Added in QM 2.3.4.

Wait for key

```
[wait] timeS event [keycode]
```

event:

K	Wait for key up event.
KF	Wait for key down event. <ul style="list-style-type: none"> Unlike K, it "eats" the keyboard event. The active window will not receive it. Not available in exe (51).

keycode - single QM key code. Optional.

- If **keycode** is omitted, waits for any key.
- keycode** also can be string variable in parentheses (QM key code), or numeric value in parentheses ([virtual-key code \(270\)](#)).

Returns virtual-key code.

On [Vista/7/8/10 \(277\)](#), does not work if QM runs as standard user and the active window has higher integrity level.

Examples

Wait for key F12:

```
wait 0 K F12
```

Wait for any key, eat the key, and display virtual-key code:

```
int vk = wait(0 KF)
out vk
```

Wait for mouse

```
[wait] timeS event
```

event:

ML	Wait for left mouse button up event. • Returns 1.
MR	Wait for right mouse button up event. • Returns 2.
MM	Wait for middle mouse button up event. • Returns 4.
M	Wait for any mouse button up event. • Returns mouse button code: 1 left, 2 right, 4 middle, 5 X1, 6 X2.

Does not "eat" the mouse event. The active window will receive it.

On [Vista/7/8/10 \(277\)](#), does not work if QM runs as standard user and the active window has higher integrity level.

Wait for CPU

```
[wait ]timeS event [threshold]
```

event:

P	Wait until CPU usage is less than threshold . • threshold must be 1 to 100.
-P	Wait until CPU usage is more than threshold . • threshold must be 0 to 99.

threshold - % CPU usage. Default: 20 % or as set with [RtOptions \(114\)](#).

Use carefully. Some programs constantly use CPU, even when they are ready for input.

See also: [GetCPU \(114\)](#).

Example

```
Run program and max. 15 seconds wait until CPU usage is less than 20%:  
run "app.exe"; wait 15 P 20
```

Wait for handle

```
[wait ]timeS H handle
```

handle - [synchronization object](#) handle.

Waits until the object becomes signaled. For example, can wait until another thread exits (see examples).

Returns 1.

Returns -1 if a handle is invalid.

wait H/HM/HMA uses Windows API wait functions ([\[Msg\]WaitForMultipleObjectsEx](#)) that modify object's state on success. For example, on successful wait for an auto-reset event, **wait** resets it; on successful wait for a mutex, the thread owns the mutex; if the mutex was abandoned, **wait** calls [SetLastError\(128\)](#). Don't use Windows API wait functions directly (unless with 0 or very small timeout) because then QM cannot properly end the thread on user request, and also cannot apply opt waitmsg.

Tip: Use variables of type [__Handle](#) to store handles to be automatically closed with [CloseHandle](#) when the variable dies. Don't close QM thread handles.

Tip: To wait for a mutex, see also [lock \(112\)](#).

Examples

Run notepad and wait until it exits (just example of wait 0 H; use run with flag 0x400 instead):

```
__Handle hProcess=run("notepad.exe") ;;the __Handle variable will close the handle
wait 0 H hProcess
```

Run function "Function" in separate thread and wait until it exits:

```
int hThread=mac("Function") ;;note: don't use __Handle variables with QM thread handles
wait 0 H hThread
```

Create event and wait until another thread sets it (SetEvent):

```
__Thread1
__Handle+ g_hevent; if(!g_hevent) g_hevent=CreateEvent(0 0 0 0)
wait 60 H g_hevent
__Thread2
SetEvent(g_hevent)
```

Wait for multiple handles (QM 2.2.0)

```
[wait ]timeS HM ha
or
[wait ]timeS HM h1 h2 [h3] [h4]
```

ha - variable of type **ARRAY(int)**, containing up to 60 handles of [synchronization objects](#).

- QM 2.4.0. Also can be **ARRAY(__Handle)**.

h1-h4 - handles of synchronization objects. Can be of type **int** or **__Handle**.

If HM used, waits until one of the objects becomes signaled. Returns 1-based index of the first signaled handle.

If HMA used (added in QM 2.4.0), waits until ALL the objects are signaled. Returns 1.

Returns -1 if a handle is invalid.

Tip: Use variables of type **__Handle** or **ARRAY(__Handle)** to store handles to be automatically closed with [CloseHandle](#) when the variable dies. Don't close QM thread handles.

Examples

Create two event objects and wait until one of them is set (SetEvent)

```
__Handle+ g_e1 g_e2
if(!g_e1) g_e1=CreateEvent(0 0 0 0)
if(!g_e2) g_e2=CreateEvent(0 0 0 0)
int i=wait(0 HM g_e1 g_e2)
out i
```

Start three notepad processes and wait until all closed

```
ARRAY(__Handle) a
rep 3
_a[]=run("notepad.exe")
wait 0 HMA a
```

Wait for variable

```
[wait ]timeS event variable
```

event:

V	Wait until the variable becomes nonzero.
-V	Wait until the variable becomes 0.

variable - variable of type int.

Returns the final value of the variable.

Wait for web page

```
[wait] timeS I [url]
```

url - wait for this URL. If omitted, waits while browser is busy.

Waits until web page is finished loading and web browser isn't busy.

Works only in Internet Explorer and IE-based web browsers.

Note: Waits only in single window. If, while waiting, the page is opened in a new window, ignores the new window.

See also: [web \(94\)](#).

Wait for cursor (mouse pointer)

```
[wait] timeS event cursor
```

event:

CU	Wait for the cursor.
-CU	Wait until the cursor disappears.

cursor - one of:

- Standard cursor identifier, such as IDC_ARROW.
- QM 2.3.2. Custom cursor identifier, recorded in the Wait dialog.

Wait for color

```
[wait] timeS event color x y [window] [flags]
```

event:

C	Wait for the specified pixel color (240) in the specified point (241) in window or screen (if window is 0 or omitted).
-C	Wait until pixel color will change from the specified color.

x y - [coordinates \(241\)](#).

[\(274\)window](#) - top-level or child window. If omitted or literal 0, **x** and **y** are screen coordinates.

flags (247):

1	x y are relative to the top-left corner of window's client area or of the work area.
2	Activate window . Restore if minimized. Error if point x y does not belong to window or its top-level parent window. No error with flag 0x1000.
0x1000	QM 2.4.3. Get pixel color directly from window , not from screen. The same as scan (87) flag 0x1000. This flag is ignored if window not used or if Windows Aero theme is not enabled or window is DPI-scaled (243) .

See also: [pixel \(105\)](#), [scan \(87\)](#).

Wait for image on screen

```
[wait] timeS event file [window] [rect] [flags] [colorDiff] [matchIndex]
```

event:

S	Wait for image.
-S	Wait until image disappears

file, window, rect, flags, colorDiff, matchIndex - same as with [scan \(87\)](#).

QM 2.3.2. Added more features.

Tips

Waiting for image (S, -S) in large region (whole screen or window) consumes significant amount of CPU time. If during that time other application performs some processing, QM may slow down it. If processing speed is important, you can view CPU usage of qm.exe, and, if it is too high, minimize it by minimizing the search region ([read more \(87\)](#)). While waiting, QM periodically searches for the image. If speed ([spe](#)) is 0, the rate is much higher.

To see CPU usage, use Windows Task Manager or perfmon.exe. You can also use [GetCPU \(114\)](#) function.

Wait for other events

Wait for user input: [mes \(62\)](#) (message box), [inp \(60\)](#) (input box), [ShowDialog](#) (custom dialog), [ListDialog](#) and other dialogs.
 Wait for accessible object (text, link, button, etc): [acc \(85\)](#). Specify the wait time in the 'Find Accessible Object' dialog.
 Wait for html element (web page objects in Internet Explorer): [htm \(86\)](#). Specify the wait time in the 'Find Html Element' dialog.
 Wait for image: [scan \(87\)](#). Select 'Wait for' in the 'Find Image' dialog.
 Run program and wait for: [run \(64\)](#). Use the 'Run Program' dialog.
 Open web page and wait for: [web \(94\)](#). Use the 'Open Web Page' dialog.

Waiting for other events can be implemented with [rep \(126\)](#). To create new "wait for" user-defined functions, you can use [WaitFor](#) template (menu File -> New -> Templates).

Note: Don't use Windows API wait functions, because then QM cannot properly end the thread on user request. QM then terminates thread, which causes a big memory leak and possibly more bad things. Also, opt waitmsg is not applied.

Examples

Wait for some condition that is not supported by wait:

```
rep() if(condition) break; else 0.1
```

Wait for mouse left button down:

```
rep() 0.01; ifk((1)) break
```

wait for one of two files

```
str file1="$personal$\file1.txt"
str file2="$personal$\file2.txt"
rep
  if(FileExists(file1)) break
  if(FileExists(file2)) break
0.5
```

Set macro speed (autodelay)

Syntax1 - set speed

```
spe [timeMS]
```

Syntax2 - get speed

```
int spe ([context])
```

Parameters

timeMS - autodelay time in milliseconds. Can be 0 to 60000. Default: 0.

- If -1, inherits from caller. If there is no [direct caller \(151\)](#), uses default autodelay for macros (read in remarks).
- If -2, uses default autodelay for macros.

context - if omitted or 0, returns current macro/function speed. If -1 or -2, returns caller's or global speed (see above).

Remarks

Syntax1

Changes autodelay for subsequent commands.

Autodelay is a small wait time automatically added after macro commands. It makes the macro more reliable, because then the target window has more time to process input. It makes the macro slower.

Autodelay is applied to:

lef, rig, mid, dou, mou (/4), key, paste/outp, str.setsel, run, act, clo, min, max, res, siz, mov, hid, men, but, web, wait WT, MouseWheel, MouseButtonX, ArrangeWindows, RestoreMultiWinPos, CloseWindowsOf, Acc.Mouse, Acc.CbSelect, Htm.Mouse, Htm.ClickAsync, WindowText.Mouse.

In macros and menu/toolbar/autotext items, default (initial) autodelay is 100 ms, or as set with [RtOptions \(114\)](#). In functions and member functions, initial autodelay is 0.

spe changes autodelay only for current function or macro, not for functions that it calls. If a user defined function wants to use caller's speed, insert this at the beginning of the function: `spe -1`.

Tips: To add autodelay feature for a user-defined function, add `wait -2` at the end.

Syntax2: Returns current autodelay (does not change). Alternatively, use [getopt \(98\)](#) function.

See also: [opt \(97\)](#), [getopt \(98\)](#), [window \(274\)](#).

Examples

Set macro speed (wait 50 ms after macro commands):

```
spe 50
```

Set maximal speed:

```
spe
```

Get precise time (performance counter)

Syntax

`long perf`

Remarks

Returns the number of microseconds since Windows startup. Useful to measure code speed.

When high precision is not necessary, you can instead use [GetTickCount](#) (~15 ms precision) or [timeGetTime](#) (2 ms precision). They return the number of milliseconds since Windows startup.

When measuring code speed, it is a good idea to repeat it several times. Your thread may be interrupted to give CPU time to other programs and threads that need it. For example, when you move the mouse, OS needs some time to move the mouse pointer. Also, code speed depends on various CPU and other hardware features.

Note: Depending on computer speed, [perf](#) itself may get 1 or several microseconds.

Note: The type of the return value is long (64-bit), not int (32-bit). Be careful when displaying the results formatted: either at first assign the difference to an int variable, or in the format-string use %I64i instead of %i, or use F-string.

See also: [PerfFirst](#), [PerfNet](#), [PerfOut \(114\)](#)

Examples

```
measure how long runs code
long t1=perf ;;get time before
int i j
for i 0 1000
  j=5
long t2=perf ;;get time after
out t2-t1 ;;display the difference
```

Timer

Syntax

```
tim [timeS] [function] [flags]
```

Parameters

timeS - timer period, seconds. Default: 0. Error if > 2147483.647.

function - name of a user-defined function. Default: this function (the caller).

- The function does not receive arguments.

flags (247):

1	Allow multiple instances of function running simultaneously.
2	QM 2.3.3. One-shot timer. The function will run once. Without this flag, the timer is periodic.

Remarks

Sets timer that executes **function** every **timeS** seconds.

If **timeS** is 0 or omitted, stops the timer.

If timer for that function is already exists, and **timeS** is not 0, then changes or just resets its period. Next time the function will run after **timeS** seconds.

If the timer function is disabled, it does not run, however the timer runs anyway.

The timer function runs in new thread each time. It is not recommended to use **tim** with small period (<1 s).

Usually it is better to use **rep** and **wait** instead of **tim**. In dialogs procedures and other window procedures, use **SetTimer**, **KillTimer** and **WM_TIMER** instead. See the examples.

Tips

To make one-shot timer that will work in all QM versions, use `tim 0` (stop timer) in that timer function.

You can see currently active timers in the Running Items list (menu Run -> View Active Items). You can right click an item and stop the timer. Does not show timers started by macros running in separate process.

Examples

Start timer to run function Func every 30 seconds:

```
tim 30 Func
```

Stop it:

```
tim 0 Func
```

Run function Func every 2 seconds:

```
rep
wait 2
Func
```

Use timer in dialog procedure:

```
sel message
case WM_INITDIALOG
SetTimer hDlg 35 1000 0 ;;35 is timer id, 1000 is period in ms
-
case WM_TIMER
sel wParam
case 35
KillTimer hDlg wParam
out "timer 35"
```

Variable type DATE

Stores date and time.

Note: In most cases use **DateTime** type instead. Its purpose is the same as of **DATE**. It has more functions and supports milliseconds. Use **DATE** only with functions that need it, for example COM functions.

The **DATE** type is implemented using a floating-point number (**double**). Days are represented by whole number increments starting with 30 December 1899, midnight as time zero. Time values are expressed as fractional part. Time precision is 1 second. Also can be used to represent date-only (the fractional part is 0) or time-only (the whole number part is 0).

To remove the time part (leave only the date part) from a **DATE** variable, you can assign it to an **int** variable. However, when converting **DATE** to **int**, result may be rounded up. To prevent it, use the member variable *date*. See examples.

DATE supports operator = (assign). Use it to convert string (**str**, **lpstr**, **BSTR**, **VARIANT**) containing date to **DATE** and vice versa.

To format date/time string, use [str.timeformat \(236\)](#).

To store date and time, also often are used types **SYSTEMTIME** and **FILETIME**:

```
type SYSTEMTIME @wYear @wMonth @wDayOfWeek @wDay @wHour @wMinute @wSecond @wMilliseconds
type FILETIME dwLowDateTime dwHighDateTime
```

Functions

Here **var** is a variable of type **DATE**. Where the return type is not specified, the function returns **var** itself.

```
var.getclock
```

Gets current time.

```
var.fromsystemtime(st)
var.tosystemtime(st)
```

Converts from **SYSTEMTIME** to **DATE** type and vice versa. Here **st** is a variable of type **SYSTEMTIME**.

```
var.fromfiletime(ft)
var.tofiletime(ft)
```

Converts from **FILETIME** to **DATE** type and vice versa. Here **ft** is a variable of type **FILETIME**.

Obsolete functions.

Use **DateTime** class instead. If need **DATE** type, you can convert it to/from **DateTime** with its functions **FromDate** and **ToDate**. Also you can use other functions from the time category, for example **TimeSpanGetPartsTotal**.

```
var.add(diff)
```

Adds **diff** to **var**. Here **diff** is a date span variable. With **DATE** functions, date span (difference between two dates) is a variable of type **SYSTEMTIME**.

Use this function when date span is quite complex. To add/subtract days, you can use operator + or -. For example, `var=var-2` subtracts 2 days; `var=var+(4/24.0)` adds 4 hours.

```
var.sub(diff)
```

Subtracts **diff** from **var**.

```
double var.diff(date2 [diff] [part])
```

Returns difference between **var** and **date2**. The return value is negative if **var** is less than **date2**, 0 if **var** is equal to **date2**, positive if **var** is greater than **date2**.

diff can be 0 or a variable of type **SYSTEMTIME**. It receives date span (always positive).

By default, returns difference in days. If **part** is 1 then returns difference in hours, 2 - minutes, 3 - seconds.

Always returns whole number. To get exact difference between two dates, use operator -. Example: `diff=var-date2`.

Examples

```
DATE d="4/1/2003"
add 2 days
d=d+2
out d

remove the time part
DATE d=10.55
int i=d
out i ;;11 (rounded up)
i=d.date
out i ;;10 (ok)
```

Open web page

Syntax

```
web url [flags] [window] [url2] [geturl2] [gethwnd]
```


Parameters

url - web page address. Can be any string that you can type in web browser's address bar ("http://...", "javascript:...", etc). Also, can be "Back", "Forward", "Home", "Refresh", "Search", "Stop", "Quit", a local file or an Internet shortcut. If **url** is "", nothing is opened, but other features (wait, etc) are applied to current page.

flags (247):

1	Wait until web page is finished loading and browser isn't busy.
2	Open in existing window, or generate error if a compatible window is not found. Without this flag, if window not found, starts new instance of web browser. If window is specified, this flag has no effect (the specified window must exist).
4	Open in new popup window.
8	Open in new instance of Internet Explorer. Use url "" to open "about:blank".
12 (4 8)	QM 2.2.0. Open in new tab (Internet Explorer 7 and later).
16	url2 (final URL) is full or with wildcard characters (196) (*?).
32	Get "Internet Explore_Server" child window handle instead of main window handle.
64	QM 2.2.0. If new window or tab is opened, don't activate it.
128	QM 2.2.0. Vista. If new IE window must be opened, open it with protected mode off. Use this if QM fails to determine whether protected mode must be off, for example when using <code>web (" " 8)</code> . In exe (51) running as administrator or Low, this flag is ignored, and IE is launched with the same privileges as of exe.
high-order word (250)	Max wait time (s). Default: infinite.

[\(274\) window](#) - open web page in this window. If omitted or "", opens in default browser. If your default browser is not Internet Explorer but is IE-compatible (has "Internet Explorer_Server" control), its window class should be specified with [RtOptions \(114\)](#).

url2 - final URL must contain this string. If flag 1 is set, waits while browser is busy, then checks URL, and, if URL does not contain **url2**, generates error. If flag 1 isn't set, waits for **url2** even if browser isn't busy. You can use  to specify that final URL must be same as **url**. URL is taken from document, not from browser. Sometimes document's URL differs from browser's URL, for example, when error page is displayed. Default: "" (any).

geturl2 - str variable that receives final URL. Default: 0.

gethwnd - int variable that receives main window handle. If flag 32 is set - "Internet Explore_Server" child window handle. Default: 0.

Remarks

Opens web page. Navigates directly from current page that is displayed in web browser. Gives more control than [run \(64\)](#): can wait for web page to finish loading, check/get final url, run "javascript:..." on current page, navigate back, etc, open web page in certain window, always navigate in current window, always open in new window, access web browser's object model.

This function works only with Internet Explorer and other windows that display web page in "Internet Explorer_Server" control. However, if only **url** is used, web page is opened in default web browser, whatever it is (otherwise, opens in Internet Explorer). Some IE-compatible browsers (other than Internet Explorer) do not support some special strings (e.g., "Quit").

If used as function, **web** returns [IDispatch](#) interface for web browser control, and can be assigned to variable of type [SHDocVw.WebBrowser](#) (or [IWebBrowser2](#)).

The speed depends on [spe \(90\)](#).

Other ways to open web page:

1. `run url` - opens web page in default web browser.
2. `run browser url` - opens web page in the specified browser.
3. Find a link or button in a web page (use [htm \(86\)](#) or [acc \(85\)](#)), and programmatically click it.
4. Use functions of [WebBrowser](#) or [IHTMLWindow2](#) interface. For example, the [Navigate](#) function of [WebBrowser](#) can open web page in certain frame.

5. Use functions from System\Functions\Internet folder to download web page without opening it in web browser.

On Windows Vista and later, if IE protected mode is on, something may not work well, especially in exe. Also, if you use `_create` to create `WebBrowser` object, it opens two IE7 instances, etc. Use `web` instead. If you use `WebBrowser` functions to navigate to an URL in different zone, another IE7 instance is launched. IE8 works better. In IE8 tabs belong to different process(es) than the main IE frame window.

Tip: To insert `web` function, drag and drop a link or Internet shortcut onto the macro text or onto a toolbar.

Examples

```
web "http://msdn.microsoft.com/en-us/library/default.aspx" 1

SHDocVw.WebBrowser b=web("")
out b.LocationURL
```

Comments

Syntax1

(space) **comments**

Syntax2

statements **;; comments**

Remarks

Lines that begin with space are disabled. They can be used as comments.

Syntax2 can be used to place comments after commands in the same line.

Syntax1 also can be used as label with [goto](#).

Lines that begin with /, \ or ; also are disabled.

Tips

Easiest way to disable/enable a line (or several selected lines) - right click the selection bar.

Examples

This is comment

key Y ;;Enter

Sound

Syntax1 - play one of standard sounds

bee [sound]

Syntax2 - play wave file

bee [+] wavefile

Syntax3 - PC speaker

bee frequency duration

Parameters

sound - one of standard Windows sounds that are specified in Control Panel.

0 (default)	Default Beep.
1	Critical Stop (error).
2	Question. In latest Windows versions this sound is disabled by default.
3	Exclamation (warning).
4	Asterisk (information).
-1	A simple beep. If the sound card is not available, the sound is generated using the speaker.

wavefile - sound file [path. \(246\)](#)

frequency - frequency, 37 - 32767 Hz.

duration - duration, ms.

Options:

Default	Play asynchronously (continue macro immediately).
+	Play synchronously (wait until sound stops).

Remarks

Syntax2: Supported are only .wav files. If **wavefile** not found, plays default sound. If **wavefile** is "", stops currently playing sound.

Supports [macro resources \(261\)](#) (QM 2.4.1) and [exe \(51\)](#) resources (QM 2.3.4).

Note: By default **bee** does not wait until finished playing. In exe the sound stops as soon as exe process ends.

Syntax3: On Windows 7 uses soundcard instead. On Vista does not work. On older Windows, if there is no PC speaker, uses soundcard or does not work.

See also: [Play](#) (play audio file of any format).

Examples

```
bee ;;default sound
bee 1 ;;play one of default sounds (0 to 4)
bee "c:\m\moo.wav" ;;play file
bee "$windows$\Media\chimes.wav" ;;play file in Windows media folder
bee "" ;;stop sounds
```

Set local run-time options

Syntax1 - set option

`opt option [value]`

Syntax2 - save/restore options

`opt save|restore`

Parameters

option - one of words specified in the table.

value - one of values specified in the table.

- To inherit the option from the caller (macro or function that [directly called \(151\)](#) the current function), use **value -1**.
- Error if the specified value is not supported.
- QM 2.4.1. **value** is optional, default 1.

option	When QM uses it	value	
		0	other
hidden	When in macro commands is specified only window name (class is not specified).	Search only visible windows.	1 - Search all windows. Hidden windows are always searched when class is specified, also with hid- .
clip	Clipboard commands (paste/oupt , str.getsel , str.setsel).	Preserve (restore) clipboard text.	1 - Do not preserve clipboard text.
err	On run-time error.	End macro. Show error.	1 - Continue macro. Don't show error.
waitmsg	When waiting (wait , autodelay , etc). Read more in remarks.	Completely block thread execution.	1 - Allow to receive Windows messages and COM events. 2 - The same as 1, but sets this option for current thread (not only for current function).
waitcpu	Autodelay (90) .	Apply only simple autodelay.	1 - Also wait for CPU. See RtOptions (114) waitcpu_time and waitcpu_threshold . Read more in remarks.
slowmouse	Mouse commands (mou , lef , rig , mid , dou).	Move mouse immediately.	1 - move mouse slowly. The speed depends on macro speed (spe) and on the distance.
slowkeys	key	Don't wait between keys.	1 - Wait a little after each key event. The speed depends on macro speed (spe).
keymark (QM 2.2.0)	key . Read more in remarks.	Use correct scancodes.	1 - Use modified scancodes. QM 2.3.3. This is default in autotext lists.
keysync (QM 2.2.1)	key . Use to make it either faster or more reliable.	Use default synchronization method. The macro waits when sent keys are actually received by the target window. Since there is no	1 - Don't use synchronization. The keys are sent without waiting until the window actually receives them. Fastest. Should not be used if not using low level hooks (13) (unless in exe), because it can create problems with keyboard triggers.

		<p>97. opt completely reliable way to know this, it waits only for a limited time period. It also waits while the window is busy. This method is fast and reliable in most cases. However in certain conditions it may be slower.</p> <p>In exe (51), uses method 2 instead. Waiting until the target window actually receives sent keys is not supported in exe, because it is too expensive (uses hooks in dll).</p> <p>Here "in exe" actually means "in exe, if QM is not running". If QM is running, all synchronization methods work in exe like in QM.</p> <p>A special synchronization method is used if the target window belongs to the same thread.</p>	<p>2 - Use minimal synchronization. This method is used by default in exe. The macro only waits while the target window is busy. The synchronization does not work well (keys may be sent too fast) with some windows and in stress conditions (low memory, busy CPU, etc). Slowest in most cases (about 20-100 %). Should not be used if not using low level hooks (unless in exe), because it can create problems with keyboard triggers.</p> <p>3 - Use maximal synchronization. The macro waits until the key is received by the target window. Since there is no completely reliable way to know this, in some cases the macro can wait infinitely. Therefore this method should not be used in macros designed to work in any window. Exe uses method 2 instead.</p>
hungwindow (QM 2.3.0)	With key and clipboard functions, when the active window is hung.	While the window is hung, waits and shows a message box with Abort and Ignore buttons. On Abort the macro would end (error). On Ignore the macro would continue (don't wait).	<p>Don't show a message box.</p> <p>1 - wait while the window is hung.</p> <p>2 - abort. Same as if the user would click Abort.</p> <p>3 - ignore. Same as if the user would click Ignore.</p>
keychar (QM 2.3.3)	With key (56) , when using "text" or string variable. Use to type exact text.	Send normal keys. The active window receives WM_KEYx messages with usual virtual-key codes (270) . In some rare conditions the typed text may be not exactly the same as in macro, because it depends on keyboard state.	Send keys as characters. The active window receives WM_KEYx messages where virtual-key code is VK_PACKET. The typed text will always be as in macro. However in some windows it may not work.
nowarnings (QM 2.3.3)	On run-time warning. Warnings can be generated by some QM functions and user-defined functions (end (131)).	Show warnings.	1 - don't show warnings.
nowarningshere (QM 2.3.5)	On run-time warning.	Show warnings as usually.	1 - pass warnings to the caller. In warning text will be a link to the caller, not to this function.
noerrorshere (QM 2.3.5)	On run-time error.	Show errors as usually.	<p>1 - pass errors to the caller. Unhandled errors will be in the caller, not in current function.</p> <p>This option works only if there is direct caller (151).</p> <p>Other way to pass errors to the caller - handle errors with err (129) and rethrow with end (131): <code>err+ end _error</code>. Can be used in all QM versions.</p>

Remarks

Syntax1: Changes a run-time option locally (only for current instance of running function or macro).

If used as function, `opt` is identical to [getopt \(98\)](#) (returns current value).

Initially all options are 0, except keymark in autotexts. Some initial options can be changed with [RtOptions \(114\)](#).

`opt waitmsg 1` should be used if you use wait commands (explicit or implicit, e.g. autodelay or clipboard commands) in thread (running macro) that has a window (or dialog). While waiting, window must process messages. Without this option, it cannot process messages. This causes various anomalies, such as window not responding, etc. This option also is often necessary when working with COM, especially with events.

Macros that contain `opt waitcpu 1` sometimes may run very slowly, because some programs constantly use CPU. To see CPU usage, use Windows Task Manager or perfmon.exe. You can also use [GetCPU \(114\)](#) function.

About `opt keymark 1`: For keyboard triggers and some other features, QM uses keyboard hooks to intercept keyboard events. QM processes QM-pressed (*injected*) keys differently than user-pressed (*real*) keys. For example, a macro cannot trigger a hotkey macro using the `key` command. However, when the user presses or releases a key simultaneously with the `key` command, QM processes the key event like generated by the `key` command (there is no reliable way to distinguish real and injected keys, unless you use low level hook). If it is a trigger-key, the trigger will not work. If it is a wait-key (`wait 0 K`), the macro will not stop waiting. Also, user-pressed keys may be inserted (typed) between QM-pressed keys. There are two ways to avoid all these problems: 1. Use [low level keyboard hook \(13\)](#). 2. Insert `opt keymark 1` before `key`. However then in some windows some keys may not work as expected. When using low level keyboard hook, `opt keymark 1` does not change anything (everything works well with or without it).

QM 2.4.1. Added **Syntax2** for saving/restoring current `opt` and `spe (90)` settings:

- `opt save` remembers current settings (adds to an internal stack).
- `opt restore` restores the saved settings (gets/removes from the stack).

QM 2.4.3. Disabled opt end. Not error if used, but now it does nothing.

See also: [getopt \(98\)](#), [RtOptions \(114\)](#), [err \(129\)](#), [end \(131\)](#), [Errors \(48\)](#), [wait \(88\)](#), [wait for \(89\)](#), [key \(56\)](#), [paste \(58\)](#), [str.getsel \(216\)](#).

Examples

```
set local run-time option "search hidden windows"
opt hidden 1

save/change/restore local run-time options
opt save
opt slowkeys 1; spe 10
out "%i %i" getopt(slowkeys) spe
opt restore
out "%i %i" getopt(slowkeys) spe
```

Get run-time options

Syntax

```
int|lpstr getopt(option [context])
```

Parameters

option - one of options that can be set with [opt \(97\)](#), or one of options listed below.

speed	macro speed (autodelay) in milliseconds. Speed can be set locally (spe (90)) and globally (RtOptions (114)).
itemid	QM tem id (107) .
itemname	QM item name. Unlike other options, the return type is lpstr (string), not int. See also context 6 (below).
nargs	number of arguments passed. Can be used in a function with optional parameters.
nthreads	number of threads currently running. Does not include special threads (49) . If context is 4, returns total number of running threads. Otherwise, returns number of threads started by function defined by context . See also: IsThreadRunning (114)

context - one of values listed below. Default: 0.

0	Get option of current QM item (from which is called this function).
1	Get option of direct caller (151) (if current item is function and it is called from other item).
2	Get option of callback entry function or thread entry function.
3	Get option of thread entry function (macro or function that started execution).
4	Get global option. Valid only for <i>speed</i> and <i>nthreads</i> .
5	QM 2.3.3. Get option of this function or of the nearest function in the call stack where the option is not 0. Return 0 if all 0. Example: There are 3 functions, calling each other like F1->F2->F3. In F3 getopt returns 1 if the option is 1 in F3, F2 or F1.
6	QM 2.4.1. Get option of parent item of current sub-function (182) . If not in a sub-function - same as 0.
<0	QM 2.3.5. Get option of -context caller in the function call stack. For example, if context is -2, gets option of caller's caller. Returns 0 if there is no caller at this level. See also: GetCallStack (114)

Remarks

Returns 0 if the option is unavailable in the specified context.

QM 2.2.1. Added slowmouse, slowkeys, keymark, keysync.

QM 2.3.0. Added hungwindow.

QM 2.3.3. Added nowarnings.

QM 2.3.5. Added nowarningshere, noerrorswhere.

QM 2.4.1. Added itemname.

Example

```
show number of passed arguments
int na=getopt(nargs)
out na
```

Set debug mode

Not available in [exe \(51\)](#).

Syntax

`deb [+ | -] [speedMS]`

Parameters

speedMS - time, in milliseconds, to wait before executing a statement. Default: -1 (infinite).

Options:

Default	Start debug/step mode.
+ (QM 2.3.1)	Start debug mode and run until a breakpoint, then start step mode.
-	End step mode and run until a breakpoint. Does not start/end debug mode. Does not use speedMS .

Remarks

Sets debug mode for current [thread \(49\)](#) (running macro).

Also you can run current macro in debug mode from the Debug menu. Then don't need [deb](#).

In debug mode you can execute statements in steps, see variables, etc. While a thread is running in debug mode, several buttons are added to the toolbar, and optionally Debug window is shown.

In debug mode, the thread can run in step mode or not. In step mode, before each statement the thread waits **speedMS** milliseconds. If **speedMS** is omitted or -1, waits until you click one of Debug menu items or toolbar buttons.

[deb](#) sets step mode. [deb-](#) ends step mode. [deb+](#) also ends step mode. To set step mode also can be used breakpoints and menu Debug -> Options -> Break on error. Read more below. To start/end step mode, also can be used Debug menu/toolbar items. Using all these features, you can execute some parts of macro in step mode, and other parts not in step mode.

Statements in not expanded folders are executed not in step mode. This makes debugging easier because skips System functions and other low-level functions, unless you expand the folder where the function is (or use "Step Into" button).

Menu/toolbar items:

Step - executes the statement that is marked with a yellow arrow, and stops at the next statement. Then marks the executed statement with a green marker. If the statement calls a user-defined function, opens the function. Skips functions that are not in expanded folders.

Step Into - the same as above, but does not skip functions that are not in expanded folders.

Step Out - continues not in step mode, but stops when the function returns. Usually stops at the next statement in the caller function.

Run to Cursor - runs the macro (or continues) not in step mode, but stops (starts step mode) at the statement that contains the text cursor (caret). Also stops at breakpoints.

Run to Breakpoint - runs the macro (or continues) not in step mode, until a breakpoint or until the end. The same as [deb+](#) or [deb-](#).

End - end threads that are being debugged, and turn off the debug mode.

If **speedMS** is ≥ 0 , before each statement the thread waits for minimum **speedMS** milliseconds.

While the thread is waiting before executing a statement, it is not completely blocked. It allows to call callback functions, process dialog messages, COM events, etc. These callback functions then run not in step mode, even if they call [deb](#) or contain breakpoints.

Also you can use breakpoints. To add or remove a breakpoint, middle click the selection bar or use the Debug menu. Breakpoints are marked with brown circle markers in the selection bar. A breakpoint is similar to [deb](#). However breakpoints are activated only in debug mode, ie if the macro executed [deb](#) or [deb+](#) before, or you started it from the Debug menu. Breakpoints disappear when you close the macro or exit QM.

You should not put breakpoints on some flow-control statements ([err](#), empty [case](#), etc), because there they may behave not as you expect. Breakpoints in lines that are not executed at run time (comments, declarations, directives, etc) break at the next statement.

In debug mode, the debug hotkeys (F5, etc) are temporarily registered as global hotkeys and work regardless whether QM is active. Hotkeys F5 (Step), Ctrl+F5 (Run to Breakpoint) and Shift+F5 (Run to Cursor) are always available in QM window. For example, you can press F5 to run the current macro in debug step mode, or Ctrl+F5 to run it in debug mode and stop at the first breakpoint.

In menu Debug -> Options you can set some options that are active in debug mode. If 'Break' checked, stops (starts step mode) on run-time errors. If 'Display' checked, displays [handled \(129\)](#) errors in QM output.

If menu Debug -> Options -> Show Debug Window is checked, QM will show Debug window when starting debug mode. Read more below.

Tips

You can place [deb](#) in several places in macro. It allows you to debug parts of macro in different speeds or turn on/off step mode.

If you want to debug only certain message in dialog box procedure, don't forget to place [deb-](#) when leaving the code.

See also: [other debug functions \(47\)](#).

The Debug window

Function call stack

At the top is the current function. At the bottom is the [thread \(49\)](#) entry function.

You can click a function to see its variables. Double click to open.

Variables

Background colors:

blue - [member variables \(157\)](#) (this);

green - [parameters \(152\)](#);

wheat - [thread variables \(142\)](#);

yellow - variables inherited from [sub-function \(182\)](#) parent or [application folder \(11\)](#) main function.

Text colors:

purple - changed in previous step;

gray - hidden.

Variables are shown only in step mode (and not if **speedMS** used).

Examples

```
deb ;;set debug mode; stop before each statement
deb 500 ;;set debug mode; wait before each statement for minimum 500 ms
deb 0 ;;set debug mode; don't wait before statements but mark them while they are executed
deb- ;;continue; ignored if not in debug mode
deb+ ;;set debug mode and immediately continue
```

Run macro

Syntax

```
mac [+|-] [macro] [command] [a1 ...]
```

Can be used as function:

```
int mac([macro] [command] [a1 ...])
```

Parameters

macro - [QM item \(19\)](#) name (full, case insensitive) or [id \(107\)](#) (integer).

- If omitted or "" - QM item that is currently open in the code editor.
- Can be QM item path. For example, "\\Mouse\\Next" is macro "Next" in folder "Mouse".
- Can be [sub-function \(182\)](#), like "sub.SubFunctionName". Then cannot be variable.

command - some string or number. How it is interpreted depends on item type. Default: "".

If macro is	command is interpreted as
Macro or function	Something to be stored into the _command (144) variable.
Toolbar (23)	Window to be associated with the toolbar. Can be window name or handle. If the window does not exist, not error, and the toolbar will not be created.
Menu	Menu item label. Instead of showing menu, will execute that item.
Other	Not used.

a1 ... - arguments for **macro**. It receives them through [function \(152\)](#) statement.

Options:

Default	Run macro .
+	Open macro in editor (don't run).
-	End this thread (49) and run macro .

Remarks

Launches **macro**. Unlike [function](#) call, **macro** runs asynchronously, in separate [thread \(49\)](#).

Some features not available in [exe \(51\)](#).

Error if **macro** does not exist when executing the **mac** statement. If **macro** is a [sub-function \(182\)](#) - when compiling.

Supported [QM item \(19\)](#) types: all except folders and member functions.

mac can be used as function. The return value depends on item type.

If macro is	mac returns
Macro or function	Thread handle. You can use it, for example, to wait (89) until the thread ends (see example). Don't close the handle.
Toolbar	Toolbar window handle. Read more (23) .
Menu	If used as function (like <code>variable=mac("menu")</code>), mac waits until the menu is closed. It then returns a nonzero value if an item clicked, or 0 if not. If used like <code>mac "menu"</code> , it does not wait. Read more (22) .
File link	Thread handle of function that has "file link run" trigger for that file type.
Other	0

If **macro** and current thread both are [macros \(20\)](#) (not functions etc) without option 'Run simultaneously', **macro** starts when this thread ends. Then **mac** returns 0.

Note: **mac** does not wait until the new thread ends. In exe, when the main thread ends, other threads also are terminated. You can use `wait 0 H` or [WaitForThreads](#) to wait for them.

Tips

To quickly insert `mac` statement, you can drag and drop.

See also: [functions](#), [net \(101\)](#), [EndThread](#), [RunTextAsFunction](#), [wait for thread \(handle\) \(89\)](#), [find macro \(107\)](#), [create macro \(108\)](#).

Examples

```
mac "Macro5" ;;run macro "Macro5"
mac "\Folder\Macro5" ;;run "Macro5" that is in "Folder" folder
mac- "Macro5" "a" ;;end current macro and run "Macro5". Also, send string "a" as command.
mac "Func" "" "some string" 1 55.5 ;;run function "Func" and send three arguments
mac "sub.Sub1" ;;run sub-function Sub1 in separate thread
```

Get id of "Macro5" and run it:

```
int i = qmitem("Macro5")
mac i
```

If window "Notepad" exists, create toolbar "TB" and attach it to the window:

```
mac "TB" win("Notepad")
```

Run function "Function1" in separate thread and wait until it ends:

```
int hThread=mac("Function1")
wait 0 H hThread
```

Call function Function1 (unlike mac, it does not create new thread):

```
Function1
```

Run macro on other computer

Syntax

```
int net(computer password macro [retval] [a1 ...])
```

Parameters

computer - name or IP address (e.g. "125.45.245.1") of the remote computer.

- QM port of remote computer can be specified using syntax computer:port, like "125.45.245.1:8177".

password - password that is required to run macros on the remote computer.

- It is the password that is set in remote computer's QM Options -> Network.
- Can be "" for macros that the remote computer allows to run without a password.
- Can be encrypted (in Options -> Security, use "net" as function name).

macro - name of a macro or function on remote computer.

- Can be QM item path. For example, "\\Mouse\\Next" is macro "Next" in folder "Mouse".

retval - str variable that receives the return value of the macro. The macro can return a numeric or string value.

a1 ... - arguments for **macro**. It receives them through [function \(152\)](#) statement.

Remarks

Launches a macro or function on other computer in network. If **retval** is used, waits until the macro ends.

The return value is 0 or an error code:

0	The macro was started successfully. If retval is used, the macro ran successfully, unless it was terminated by the user, in which case the retval variable is empty or "0".
1	Computer not found.
2	Could not connect. Possible reasons: The remote computer is not running; not connected to the network; QM is not running; QM does not allow to run macros from other computers (Options -> Network -> Allow ... is unchecked on remote computer); Firewall problems; Different port is specified; The port is in use; Problems with network connection (try to disable/enable or repair).
3	Password incorrect, or, if password is "", this macro is not allowed to run without a password.
4	Macro not found.
5	Could not start the macro. For example, it contains errors, or other macro is running.
6	Could not send arguments. The macro did not run.
7	Could not retrieve the return value. The macro ran successfully.
8	Run-time error in the macro.

The `_command` variable of the macro that runs on the remote computer contains caller computer's IP address.

Initially, accessing QM from other computers is disabled. You can enable it in Options -> Network. You don't have to enable it on your computer if you only want to run macros on other computers (to use [net](#)).

You can see your computer's name and IP addresses in Options -> Network. You can change computer name in My Computer's Properties.

Both computers must use the same port. QM port number is displayed and can be changed in Options -> Network. Initially it is 8177. If it conflicts with other applications, etc, you can change it. Recommended values are between 5000 and 65000. If QM on remote computer uses different port, you can specify the port number in **computer**, using syntax computer:port, like "125.45.245.1:8178".

Although [net](#) normally is used in local network, it also can work on the Internet. But several problems exist, such as firewall and unknown IP.

If your Internet connection uses a firewall or other security features, QM may be inaccessible from the Internet or even from your network. You would have to configure a firewall to allow incoming connections on QM port. To configure Windows XP (prior to SP2) Internet Connection Firewall, right-click your connection in Control Panel -> Network Connections, click Properties, Advanced, Settings, Add, type QM as service description, enter your computer name or 127.0.0.1, and enter QM port. If you have Windows XP SP2 or later, it is done automatically if you click Unblock in the Windows Security Alert dialog when QM network features are enabled first time. If you click Keep Blocking, QM cannot be accessed from network too. You can configure firewall to allow accessing QM only from your network.

To identify a computer on the Internet, is used different IP address than in local network. You can see IP addresses in Options -> Network -> Computer Info. An IP can be static (computer always have the same IP) or dynamic (computer's IP changes).

To send a local macro to a remote computer, you can use the [NetSendMacro](#) function, which runs the [NewItem](#) function (same as [newitem \(108\)](#)) on the remote computer. You can also send and receive files. See the examples.

[net](#) may be slow when using computer name. If you call it repeatedly, better use IP. To get IP of a computer, you can use function [GetIpAddress](#) (it is also slow but you call it only 1 time, and then multiple times call [net](#) with IP).

To control a remote computer visually, use Windows XP Remote Desktop or other remote control software.

See also: [network setup \(258\)](#), [shared files \(17\)](#), [mac \(100\)](#), [sharing files and text between computers](#)

Examples

```
run function NetworkMessage on computer John, and pass "Hello!" as argument
net "John" "passw555" "NetworkMessage" 0 "Hello!"

function NetworkMessage (on computer John)
function $message
mes message



---



get file from another computer
int r
str s
r=net("123.234.35.1" "pppassw" "net_GetFile" s "$personal$\test.txt")
if(r or !s.len) end "failed" ;;the remote macro net_GetFile did not run or did not return the requested file
s.setfile("$personal$\test.txt") ;;save it on this computer

function net_GetFile
function' str $file_
str s.getfile(file_)
ret s



---



send file to another computer
int r
str data.getfile("$personal$\test.txt")
str sr
r=net("123.234.35.1" "pppassw" "net_SetFile" sr "$personal$\test.txt" data)
if(r or sr!="1") end "failed" ;;the remote macro net_SetFile did not run or did not return 1

function net_SetFile
function# $file_ str' data
data.setfile(file_)
ret 1



---



send and run function Function7 on Computer3
str c("Computer3") p("p765") m("Function7")
if(NetSendMacro(c p m)) end "could not send"
if(net(c p m)) end "could not launch"



---



This macro upgrades QM on several computers.

str setupfile="$desktop$\quickmac.exe" ;;QM setup program. Macro will send and run it on each computer.
str computers="computer1[]computer2[]computer3" ;;list of computers
str password="passw" ;;assume, all computers use same password (Options -> Network)
str cmdline="/silent" ;;install without the setup wizard, using the same settings as when installing last time

str c sd.getfile(setupfile)
int r
```

```
str sm=
function str'filedata $cmdline
str temp="$temp qm$\quickmac.exe"
filedata.setfile(temp)
run temp cmdline

foreach c computers
_r=net(c password "NewItem" 0 "NetUpgradeQm" sm "Function" "" "\User\Temp" 17) ;;send macro
that receives and runs setup file
if(!r) r=net(c password "NetUpgradeQm" 0 sd cmdline) ;;if ok, run the macro
if(r) out "Cannot upgrade QM on %s (error %i)" c r
else out "QM upgraded on %s" c
```

Disable/enable triggers

Not available in [exe \(51\)](#).

Syntax1 - set state

```
dis [+|-] [macro] [flags]
```

Syntax2 - get state

```
int dis([macro] [flags])
```

Parameters

macro - [QM item \(19\)](#) name (full, case insensitive) or [id \(107\)](#) (integer).

- Can be QM item path. For example, "\\Mouse\\Next" is macro "Next" in folder "Mouse".
- Can be list of items (multiple lines).
- If [sub-function \(182\)](#), QM uses its parent function.

flags:

0	macro is not folder.
1	macro is folder.
2	macro is folder or not.

Options:

Default	toggle.
+	disable.
-	enable.

Remarks

Syntax1

Disables or enables **macro** trigger.

If **macro** is omitted, disables/enables QM (same as menu [Run -> Disable Triggers \(8\)](#)).

If **macro** is "", disables/enables current file.

Syntax2

If used as function, just returns disabled/enabled state. For items, returns 0 if enabled, 1 if disabled, 2 if enabled but is in disabled folder or file. For QM, returns combination of flags indicating which trigger types are disabled: 1 - keyboard, 2 - mouse, 4 - window, 8 - command line, 16 - user-defined, 32 - file, 64 - event log, 128 - QM is in "disabled" state (gray icon), 0x100 - process, 0x200 - accessible object.

If **macro** is "", returns 1 if current file is disabled, or 0 if enabled.

Examples

```
dis ;;disable or enable QM
dis+ "Macro" ;;disable macro
dis+ "\\Folder\\Macro"
dis "Folder1[]Folder2" 1 ;;disable or enable two folders

int i = qmitem("Macro139")
dis i
if(dis(i)) out "disabled"; else out "enabled"

if(dis&128) ret ;;if QM is disabled, exit
if(dis&16) ret ;;user-defined triggers are disabled, exit
```

Register function to run when macro ends

Syntax

```
atend function [argument]
```

Parameters

function - name of a user-defined or dll function. Can be [sub-function \(182\)](#).

argument - some value to pass to the function. The function can optionally have one parameter of type int or pointer. Default: 0.

Remarks

Sometimes it is important to execute some code when macro ends. Macro can, for example, end on an error. This statement registers **function** to run when current macro ends.

function runs in [thread \(49\)](#) that registered it. When it runs, local variables of current thread's entry function still exist. The [_error \(129\)](#) variable is empty if the macro finished successfully, or contains error info if the macro failed.

Registered functions run in LIFO (last-in, first-out) order. `atend` for same **function** can be called more than once in thread, but it does not register **function** again if **arg** value is the same.

Do not use `atend` in functions that run in special [threads \(49\)](#) and threads created by dll functions.

In **function** don't use `end` (except to generate warning) and avoid unhandled run-time errors. [More info \(131\)](#).

Examples

```
atend RestoreCapsLock
```

```
int* p=malloc(10); atend free p  
...
```

Shutdown Windows, QM, macro, etc

Syntax

`shutdown` *action* [*flags*] [*computer|command|threadname*] [*wait_|threadid*] [*message*]

Parameters

action - one of values below.

0	Log off Windows.
1	Shut down Windows.
2	Restart Windows (reboot).
3	Shut down and power off.
4	Hibernate.
5	Suspend (sleep, standby).
6	Lock computer or switch user.
-1	Exit Quick Macros. In exe (51) - exit the exe process (QM 2.2.0).
-2	Restart Quick Macros.
-3	Hide Quick Macros.
-4	Show Quick Macros.
-5	Reload current QM file (17) .
-6	End currently running macro, or specified thread (49) . Read more below.
-7	Softly end current thread. Read more below.

Actions -2, -3, -4 and -5 not available in exe. RT error if used.

flags - see below. Depends on **action**. With other actions should be 0 or omitted. Default: 0.

action	flags	
0 - 5	0	Applications are allowed to cancel the operation. For example, if an application shows a "Save?" message box, you can click Cancel to stop the shutdown.
0 - 5	1	Does not allow to cancel. When used with actions 0 - 3, forces all applications to terminate, which can cause them to lose data. This value also should be used if the computer is locked.
0 - 3	2	Forces to terminate only hung applications.
-6	1-8	Read in Remarks.
-7	1	End thread immediately. Read more in Remarks.

These three parameters are used with action 1 and 2:

computer - computer name. Default: "" - this computer.

- To shut down other computer, your user account must have administrator rights on that computer.

wait_ - time (seconds) to wait. A shutdown dialog box or other notification is displayed. Default: 0 (no dialog box).

- Tip: To cancel the shutdown, use function [AbortSystemShutdown](#).

message - text to display in the shutdown dialog. Default: "".

command - [QM command line \(18\)](#). Used with action -2 (restart QM). For example, it can be `"v"` to restart visible.

threadname - QM item name or [id \(107\)](#). Used with action -6 (end threads). Read more in Remarks.

threadid (QM 2.2.0) - thread handle, id or unique id (depends on **flags**). Used with action -6 (end threads). Read more in Remarks.

Remarks

This command is asynchronous. It tells QM to perform the specified action but does not wait until the action is finished. With many actions, it should be the last command in the macro.

Use action -6 to end a [thread \(49\)](#). [Read more.](#)

QM 2.3.3. You can instead use function [EndThread](#). It works like [shutdown](#) -6.

flags:

0	QM 2.2.0. threadid is thread handle. Read more below.
1	QM 2.2.0. threadid is thread id.
2	QM 2.2.0. threadid is thread unique id.
4	QM 2.3.2. If a function has "End thread" trigger for the thread, run it.
8	QM 2.3.2. Synchronous. Wait until the thread is ended.

If **threadname** and **threadid** are not used (omitted, 0 or ""), ends the currently running macro. If only **threadname** used, ends all threads (all running instances of the function). If **threadid** used, ends only that instance (**threadname**, if used, also must match). Not error if the thread is not running. Cannot end [special threads \(49\)](#). Cannot close toolbars (use [clo](#) instead). Don't use this to end current thread; instead use [ret](#), [end](#), [shutdown](#) -7, etc.

To get thread handle, id or unique id, use [mac \(100\)](#), [EnumQmThreads \(114\)](#) or [GetQmThreadInfo \(114\)](#). Don't use [GetCurrentThread](#) or [DuplicateHandle](#). A thread handle identifies thread only while it is running. Later (after > 3 seconds) the same value can be reused (assigned to a new thread). Thread id also can be reused. Unique thread id is not reused.

If [sub-function \(182\)](#), **threadname** can be like "ParentName:SubName".

Use action -7 to end current thread when you don't know if it has windows. Differently than [end \(131\)](#), it closes all thread's windows, giving them chance to free allocated memory, etc. When [shutdown](#) returns, windows are already destroyed. If **flags** is 1, ends thread immediately after destroying windows. If 0 - gives 0.5 s (or [wait_](#) ms, if used) to finish naturally. This action for example can be used in a function that doesn't know how to properly end current thread.

Depending on operating system, hardware and security settings, some features may not work.

Examples

```
shutdown 3 ;;shut down; allow to cancel
shutdown 2 1 ;;reboot; don't allow to cancel
shutdown 1 0 "" 30 "QM" ;;shut down after 30 s
shutdown -1 ;;exit QM
shutdown -6 ;;end currently running macro
shutdown -6 0 "Func" ;;end all threads (running instances) of function Func
```

Get point color

Syntax

```
int pixel(x y [window] [flags])
```

Parameters

x y - [coordinates \(241\)](#).

[\(274\)](#)**window** - top-level or child window. If omitted or literal 0, **x** and **y** are screen coordinates.

flags [\(247\)](#):

1	x y are relative to the top-left corner of window's client area or of the work area.
2	Don't activate window . Not error if point x y does not belong to window or its top-level parent window.
0x1000	QM 2.4.3. Get pixel color directly from window , not from screen. The same as scan (87) flag 0x1000. It is faster. Does not activate window . Not error if x y does not belong to window ; then returns -1 if the point is not in the window or client area rectangle. This flag is ignored if window not used or if Windows Aero theme is not enabled or window is DPI-scaled (243) .

Remarks

Returns pixel [color \(240\)](#).

When **window** is used:

- Restores minimized window.
- Activates inactive window (depends on **flags** and window style).
- Error if the point belongs to another top-level window (depends on **flags**).

Don't use this function to search for a pixel, because it is slow. Instead use [scan \(87\)](#) (it can search for one or more pixel colors or images).

See also: [wait for color \(89\)](#), [scan \(87\)](#).

Examples

```
int w=win("Quick Macros" "QM Editor")
int color=pixel(100 100 w 1|0x1000)
int r g b
ColorToRGB(color &r &g &b)
out F"color=0x{color} red={r} green={g} blue={b}"
```

Store color from mouse pointer into the clipboard:

```
str s=F"0x{pixel(xm ym)}"
s.setclip
```

Registry and ini file functions

Registry functions

The registry is a database where Windows and applications store settings. Macros also can store their settings there.

Syntax

```
int rset(data [name] [key] [hive] [options|datatype])
int rget(variable [name] [key] [hive] [default] [datatype])
```

Parameters

data - data to be stored in the registry. Can be variable or other expression.

variable - variable that receives data stored in the registry.

name - registry value name. Use "" for "(Default)".

key - registry key.

- Default (or ""): "Software\GinDi\QM2\User".
- If begins with \, it is interpreted as subkey of default key.
- Must not begin with "HKEY_CURRENT_USER\" or similar.
- Also can be an open key handle (QM 2.3.0).

hive - one of HKEY_... constants. Integer.

- Default (or 0): HKEY_CURRENT_USER.
- Add flag HKEY_64BIT (eg `HKEY_LOCAL_MACHINE|HKEY_64BIT`) if you need 64-bit key.

default - default value to be used if the value does not exist in the registry. Default: "" for strings, 0 for numeric variables. Cannot be used with user-defined types.

options:

0 (default)	write data.
-1	delete value. <ul style="list-style-type: none"> • data is not used and can be 0 or "".
-2	delete subkey. <ul style="list-style-type: none"> • data - not used and can be 0 or "". • name - subkey name. • key - parent key. • QM 2.3.5. Can be used only key, like <code>rset "" "" "ParentKey\KeyToDelete" 0 -2</code>
-3	delete subkey, but only if it does not contain subkeys.
-4 (QM 2.2.0)	delete subkey, but only if it is empty (does not contain subkeys and values).

datatype - registry data type. Can be used only with strings. Can be one of REG_... constants, for example REG_BINARY. Default: REG_SZ.

Remarks

rset writes **data** to the registry. If the specified key does not exist, creates it. Also used to delete values and keys.

rget reads data from the registry and stores it into the **variable**.

Each of these functions returns a nonzero value on success. For strings, it is string length+1. For data of other type, it is registry data length. If fails (e.g., key or value does not exist, access denied, etc), returns 0. You can get error description using [str.dllerror \(207\)](#).

When deleting, **rset** returns 1 if successful, 0 if failed. Returns 0 if the value or key did not exist; then

`if(GetLastError=ERROR_FILE_NOT_FOUND)` will be true, unless it failed for some other reason, for example if access is denied (ERROR_ACCESS_DENIED).

Warning: You should not use **rset** if you are not sure that this will not damage something. You can safely use **rget**. It is quite safe to use **rset** if **key** is omitted, "", or begins with \, because then you deal only with the key dedicated for QM macros.

Hive HKEY_CURRENT_USER (default) is used for current user's settings. Other hives are used for settings common to all users.

For security reasons, **rset** usually cannot write to hives other than HKEY_CURRENT_USER on non-administrator user accounts. On [Windows Vista/7/8/10 \(277\)](#), it fails to write to these hives even on the administrator account if QM is running

not as administrator.

Registry data format:

expression/variable type	registry data format
str, lpstr	REG_SZ or datatype
int, byte, word	REG_DWORD
other	REG_BINARY

QM 2.3.0. If **variable** is a str variable, and **datatype** is REG_SZ or 0 or omitted, **rget** succeeds even if data type in registry is not of REG_SZ type. REG_EXPAND_SZ strings are automatically expanded. REG_MULTI_SZ strings are correctly retrieved too. REG_DWORD and REG_QWORD values are converted to string. Data of other nonstring types is copied to the str variable unmodified. In older QM versions, **rget** would fail if registry data type is not a string type. String types are REG_SZ, REG_MULTI_SZ and REG_EXPAND_SZ.

If **variable** is a str variable, and **datatype** other than 0 or REG_SZ is specified, **rget** fails if data type of the value in the registry is other. With variables of other types, data type in the registry must match too.

Variables of composite types except str cannot be used. These are ARRAY, BSTR, VARIANT, interface pointer, and types containing these types or str. With **rget** also cannot be used pointers and lpstr. Types containing str can be used if you convert them from/to string. See [str.getstruct/str.setstruct \(222\)](#).

rset and **rget** can be used with single argument. Then the argument must be a variable, in simplest form (without ., [], etc). The variable name is used for the registry value name. For example, `rset v` is the same as `rset v "v"`.

Tips

If your macro has many settings, try class **__Settings**. It makes easier to manage them. Displays in grid control, where user can change them.

Examples

```

retrieves value "Logon User Name" from HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer
str s
if(rget(s "Logon User Name" "Software\Microsoft\Windows\CurrentVersion\Explorer"))
_out s
else
_out "the value or the key does not exist"

retrieves the default value from HKEY_CLASSES_ROOT\mp3file\shell\play\command
str s
if(rget(s "" "mp3file\shell\play\command" HKEY_CLASSES_ROOT))
_out s

sets value "test" in HKEY_CURRENT_USER\software\gindi\qm2\user\MyFunctionSettings
rset "data" "test" "\MyFunctionSettings"

retrieves value "test" from HKEY_CURRENT_USER\software\gindi\qm2\user\MyFunctionSettings
str s
rget s "test" "\MyFunctionSettings"
out s

sets value "test" in HKEY_CURRENT_USER\software\gindi\qm2\user
int test = 100
rset test

retrieves value "test" from HKEY_CURRENT_USER\software\gindi\qm2\user
int test
rget test
out test

deletes value "test" in HKEY_CURRENT_USER\software\gindi\qm2\user\MyFunctionSettings
rset "" "test" "\MyFunctionSettings" 0 -1

deletes key "MyFunctionSettings" in HKEY_CURRENT_USER\software\gindi\qm2\user
rset "" "MyFunctionSettings" "" 0 -2

```

Ini file functions

Ini files can be used to store settings. However it is not recommended because is too limited. No Unicode, not all characters allowed, no hierarchy, etc. To store small-size settings, instead use the registry. For bigger-size settings and other data you can use [XML \(117\)](#), [CSV \(116\)](#) or [Sqlite](#).

Syntax

```
int rset(data name section inifile [options])
int rget(variable name section inifile [default])
```

Parameters

data - data to be stored in the ini file. Can be variable or other expression.

variable - variable that receives data stored in the ini file.

name - ini value name (also called key). Everything is the same as with the registry.

section - ini section name. In ini files section names are in [].

inifile - ini [file \(246\)](#). Default: "\$my qm\$User.ini".

default - default value to be used if the value does not exist. Everything is the same as with the registry.

options:

0 (default)	write data.
-1	delete value. <ul style="list-style-type: none"> • data is not used and can be "".
-2	delete section. <ul style="list-style-type: none"> • data and name are not used and can be "".

Remarks

[rset](#) writes **data** to the ini file. If the file or section does not exist, creates.

[rget](#) reads data from the ini file and stores it into the **variable**.

Each of these functions returns 1 on success, 0 on failure. If [rset](#) failed, you can get error description using [str.dllerror \(207\)](#).

Example

```
write to ini file
rset "data" "a" "section" "$desktop$\test.ini"
```

QM items: find; enumerate; get properties

Syntax

```
int qmitem([name] [flags] [pi] [mask])
or
int qmitem(iid [flags] [pi] [mask])
```

Parameters

name - [QM item \(19\)](#) name. Full, case insensitive.

- If omitted or "", gets item that is currently open in code editor.
- Can be QM item path. For example, "\\Mouse\\Next" is macro "Next" in folder "Mouse".
- QM 2.4.0. Can be QM item [GUID \(249\)](#) string. Unavailable in exe.

iid - QM item id. Integer.

- If [sub-function \(182\)](#), QM uses its parent function. To get name or id of current sub-function, use [getopt \(98\)](#).

flags (247):

1	Skip folders.
2	Skip items in shared folders (17) (eg System).
4	Skip encrypted items.
8	Skip disabled items.
16	Skip items with no trigger.
32	Skip member functions.
64	QM 2.3.0. Skip file links.

pi - variable of type QMITEM (see Remarks). Function fills it with information about found item.

mask - defines which members of QMITEM must be retrieved. It can be combination of values listed below. Other members are always retrieved. Default: 0.

1	Name.
2	Trigger (without programs).
4	Programs. Begins with / (only) or \ (not).
8	Text.
16	Parent folder id. Obsolete, used before QM 2.4.0.
32	Filter function.
64	Trigger description (short).
128	Date modified.
256	QM 2.3.0. File link target (file path).

Remarks

Each macro and other QM item, including folders, has an integer identifier (id). It can be used with [mac \(100\)](#), [dis \(102\)](#), [str.getmacro \(219\)](#), [qmitem](#) and some other functions. It changes when the file is opened next time.

[qmitem](#) returns QM item id. It can be 1 to 65535. Returns 0 if not found.

To get id of current or related function, instead use [getopt itemid \(98\)](#), it's faster and don't need to specify name.

This function can be used for 3 purposes:

1. Get item id when you know its name, path or GUID.

- If not found, [qmitem](#) returns 0.
- **flags** can be used to filter results (e.g., skip folders).

2. Enumerate QM items.

- Use **iid**. It must be ≤ 0 .
- [qmitem](#) returns next matching item. It starts to search from item whose id is absolute value of **iid** + 1. For example, if **iid** is -2, [qmitem](#) returns item that has id 3 or more (it can skip items that match **flags**).
- To enumerate all items, at first call [qmitem](#) with **iid**=0. Then repeatedly call it and use negative form of **iid** that was returned in previous call. Stop when [qmitem](#) returns 0.

3. Get item properties.

- **pi** must be variable of type QMITEM.
- **qmitem** always fills its first 9 members and **folderid**. To fill other members, use **mask**.
- The first argument can be either **name** or **iid** (<=0 when enumerating).

```
type QMITEM !itype !ttype !tkey !tmod !tkey2 []!tmon []!tth flags htvi ~name ~trigger
~triggerdescr ~programs ~filter ~text ~linktarget folderid DATE'datemod
```

itype - item type: 0 macro, 1 function, 2 pop-up menu, 3 toolbar, 4 autotext list, 5 folder, 6 member function, 7 file link.

ttype - trigger type: 0 none, 1 keyboard, 2 mouse, 3 window, 4 user-defined, 5 QM events, 6 file, 7 event log, 8 process, 9 accessible object, 10 autotext.

tkey - trigger key. For keyboard triggers it is [virtual-key code \(270\)](#). For mouse triggers: 1 wheel forward, 2 wheel backward, 4 X1 button, 5 X2 button, 6 left button, 7 right button, 8 middle button, 9-20 screen edges, 21-32 other movements. For other triggers - undocumented.

tmod - modifier-keys (keyboard and mouse triggers). Combination of the following values: 1 Shift, 2 Ctrl, 4 Alt, 8 Win.

tkey2 - next key (keyboard triggers).

tmon - monitor (mouse movement triggers). 0 primary, 31 all.

tth - [hit test code \(40\)](#) (mouse click and wheel triggers). 0 any.

flags (247) - item properties: 2 is in a [shared folder \(17\)](#), 4 encrypted, 8 disabled, 16 is in a read-only folder.

htvi - handle of item in TreeView control.

name - item name.

trigger - [encoded trigger string \(264\)](#) (without programs and filter function).

triggerdescr - trigger description (e.g., Ctrl+E instead of Ce).

programs - trigger scope (programs). Starts with / (Only) or \ (Not).

filter - [filter function \(40\)](#)

text - item text.

linktarget - QM 2.3.0. If the item is file link, contains file path.

folderid - parent folder id.

datemod - date modified. Precision: 1 s. Time zone: local. This member is 0 for items modified before QM 2.1.5. Modification date is updated when applying changes in text.

In [exe \(51\)](#) can be used only to get item id.

See also: [str.getmacro \(219\)](#), [getopt itemid \(98\)](#), [GetQmItemsInFolder \(114\)](#)

Examples

Find item "LED":

```
int i=qmitem("LED" 1)
if(i) out i; else out "item not found"
```

Find item "LED" and display properties:

```
QMITEM q
int i=qmitem("LED" 1 &q 31)
if(i) out "i=%i itype=%i ttype=%i tkey=%i tshift=%i flags=%i name=%s trigger=%s programs=%s
folder=%s text=[]%s" i q.itype q.ttype q.tkey q.tmod q.flags q.name q.trigger q.programs
iif(q.folderid _s.getmacro(q.folderid 1) "") q.text
```

List all items that have triggers:

```
QMITEM q; int i
rep
  i=qmitem(-i 1|16 &q 1|2|4)
  if(i=0) break
  out "%-30s %-30s %s" q.name q.trigger q.programs
```

Create, modify, copy and delete QM items

Not available in [exe \(51\)](#).

Syntax

```
int newitem([name] [text] [type|template] [trigger] [folder] [flags])
```

Parameters

name - name.

- Also can be [id \(107\)](#) (integer) of existing item (for example, when deleting or replacing).
- Error if contains invalid filename characters or is invalid name for function or member function.
- If "", uses name of type or template.

text - item text.

- For file link item - file path.
- If "", copies text of template.

type - [QM item type \(19\)](#). Can be: "Macro", "Function", "Menu", "Toolbar", "Autotext", "Member", "Folder", "File Link".

- If omitted or "", creates macro.
- Instead of "Autotext" can be "T.S. Menu" or "TS Menu", for compatibility with QM versions < 2.3.5.

template - name or id of an existing item. Its type and other properties will be applied to the new item.

- Must not be folder. QM 2.3.3: can be folder if using id.
- QM 2.3.3. Can be file link.

trigger - [encoded trigger string \(264\)](#).

- QM 2.4.0. If omitted or "", copies trigger of template.
- QM 2.3.3. Can be used with folders. Sets scope (programs). Example: "/IEXPLORE,Firefox".

folder - name or id of folder where new item must be created.

- Also creates the folder if does not exist.
- Can be full path, like "\\Folder\\Subfolder".
- If omitted or "", creates not in a folder.

flags (247):

0	If item with name exists, create new item with unique name, e.g., "Macro" -> "Macro2". <ul style="list-style-type: none"> • Flags 0 - 3 cannot be used together. • QM 2.3.3. Does not make unique name if it is folder.
1	If item with name exists, replace it. <ul style="list-style-type: none"> • Cannot replace folder.
2	If item with name exists, error.
3	If item with name exists, replace its text. <ul style="list-style-type: none"> • The user can Undo if the item is currently open (is in the "Open items" list). • Does not replace other properties. • Alternatively you can use str.setmacro (220).
4	Select new item.
8	Let user edit name of new item (if flag 4, and QM window is active).
16	Create at the bottom.
32	Delete item. <ul style="list-style-type: none"> • Other flags (except 64), text and trigger are not used. • Not error if item not found. Then the function returns -1.
64	Confirm (with flag 32). Error if user clicks Cancel.
128 (QM 2.3.0)	Temporary. Temporary items disappear when closing file. <ul style="list-style-type: none"> • QM 2.4.0. Creates in the Temp folder, regardless of folder.

Remarks

Creates new [QM item \(19\)](#), or replaces/modifies/deletes existing item. Returns [item id \(107\)](#).

Cannot create/replace/modify/delete items in read-only folders. Cannot replace (flag 1), modify (flag 3) or delete (flag 32) existing item if it is in other folder or has other type, unless item id is used.

QM 2.2.0. Can be used encrypted templates. Then the new item also will be encrypted. Also can replace (flag 1), replace text (flag 3) and delete (flag 32) encrypted items. If flag 3 used, and the item already exists, it also must be encrypted.

Example

Execute QM macro code that is stored in file "test.txt". This will create temporary macro with text of file and execute it:

```
str s.getfile("$desktop$\test.txt")
mac newitem("temp_file_text" s "" "" "\Temp\files" 1)
```

Get size of a type

Syntax

```
int sizeof(typeName)
```

Parameters

typeName - name of user-defined type or intrinsic type.

Remarks

Returns size (number of bytes) of variables of **typeName**. It is compile-time function.

Example

```
type POINT x y  
int ts = sizeof(POINT)  
now ts is 8
```

Get GUID of interface or coclass

Syntax

*GUID** **uuidof**(*identifier*)

Parameters

identifier - one of:

- Name of interface or coclass.
- Interface pointer variable.
- GUID in form "{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}".
- ProgId in form "Application.Class".

Remarks

Returns pointer to [globally unique identifier \(249\)](#) which is associated with **identifier**. It is a compile-time function.

Example

```
isfd.BindToObject(pidl 0 uuidof(IShellFolder) &isf)
```

Shared memory

This function removed in QM 2.3.4. Instead use `__ProcessMemory` class. If you have code from QM forum with this function, look [here](#) for new version.

Not available in [exe \(51\)](#).

Syntax

```
int* share ([window])
```

Parameters

[\(274\)](#)**window** - top-level or [child \(275\)](#) window that belongs to the application with which will be used shared memory.

Remarks

If you try to use [SendMessage](#) function to get or set data in other application (other than QM) using pointer or string, in most cases it will not work, and may crash the application or damage its data. It is because the pointer is in address space of QM, and is invalid in address space of other application. The solution is to use shared memory.

QM provides 1 KB of shared memory that macros can use. Function [share](#) returns its address. It returns different value for each application (although physical memory is the same, but applications use different address to access it). Call [share](#) twice: Call it without **window** argument to get shared memory address in QM context. Call it with **window** argument to get shared memory address in context of the application to which belongs the window. Use the first address (possibly with some offset) to get and/or set data. Pass the second address (possibly with some offset) to that application (usually as IParam of [SendMessage](#)).

Always use `lock _share` when using shared memory. It prevents using shared memory by multiple simultaneously running [threads \(49\)](#). See examples. Read about [lock \(112\)](#).

Examples

Pass address of a variable of a user-defined type

```
lock _share
VARIABLETYPE* variable=+share
memset(variable 0 sizeof(VARIABLETYPE))
variable.member1=value1
variable.member2=value2
SendMessage(hwnd message wparam share(hwnd))
out variable.member1
lock- _share
```

Set status bar text

```
int hwnd=child(" " "msctls_statusbar32" "Notepad" 0x1)
str text="text from QM"
lock _share
strcpy(+share text)
SendMessage(hwnd SB_SETTEXT 0 share(hwnd))
lock- _share
```

Get toolbar button text

```
int hwnd=child("Notification Area" "ToolbarWindow32" "+Shell_TrayWnd" 0x1)
int buttonID=0
lock _share
SendMessage(hwnd TB_GETBUTTONTEXT buttonID share(hwnd))
str s.get(+share 0)
lock- _share
out s
```

Get SysListView32 control item text

```
int hwnd=child("FolderView" "SysListView32" "My QM" 0x1)
int item=0
lock _share
LVITEM* lip=share ;;in QM context
```

```
LVITEM* lip2=share(hwnd) ;;in hlv context
memset(lip 0 sizeof(LVITEM))
int stringoffset=sizeof(LVITEM)+20
lip.pszText+=(lip2+stringoffset) ;;in hlv context
lip.cchTextMax=260
lip.mask=LVIF_TEXT
SendMessage(hwnd LVM_GETITEMTEXT item lip2)
str s.get(+lip stringoffset)
lock- _share
out s
```

Prevent simultaneous execution of code by multiple threads

Syntax1 - lock

`lock` [name] [mutex] [timeoutMS]

Syntax2 - unlock

`lock-` [name]

Parameters

name - name of this lock. Must be like a [variable name \(257\)](#), without quotes. Used to identify the same lock in multiple functions (or macros, etc). Can be omitted or literal 0 if this lock is used only in current function.

mutex - mutex name. String. Case sensitive. Can contain any characters except \. Can have "Global\" or "Local\" prefix. Read more in remarks.

timeoutMS - max time (milliseconds) to wait in case another thread is executing locked code. Integer. On timeout throws error that can be handled by [err \(129\)](#). If omitted or -1, waits infinitely. To just check whether the code is locked, use 0.

Options:

-	unlock.
---	---------

Remarks

Added in QM 2.2.0.

A [thread \(49\)](#) is a running function or macro. Multiple threads can run simultaneously. It means that two or more threads can execute the same code block at the same time. Sometimes it is important to prevent this. For example, if one thread is writing to a file (e.g. using `str.setfile`), another thread would fail to open the file at that time. Or, if one thread is modifying a global variable, the value of the variable may become incorrect if another thread also modifies it at the same time. Use `lock` to avoid this.

Syntax1

Locks the following block of code (code from `lock` to `lock-` or to the end of the function). It means that the code cannot be executed by more than one thread at a time. If thread2 wants to execute the code while thread1 is executing it, thread2 waits until thread1 finishes executing the code.

Syntax2

Unlocks the code block that was locked by `lock`. Unlocking in many cases is not necessary. QM automatically unlocks the code when the function exits.

If **name** was used with `lock`, use the same name with `lock-`.

`lock-` unlocks the code only if `lock-` was actually executed. For example, if you `goto` somewhere without executing `lock-`, the code remains locked.

You can use the same named lock (the same **name**) in multiple functions. For example, while one thread is executing locked code in function1, other threads also will not be able to execute locked code in function2 if the code is locked with the same named lock.

Don't use multiple `lock` without `lock-`. The lock count is incremented (by 1) by `lock` and decremented (by 1) by `lock-` and when the function exits. If the lock count is more than 1 when the function exits, all code locked by that lock object remains locked. To reset, you can execute `lock-` somewhere else, or restart QM.

Be careful using `lock`. Two threads can lock each other, causing both to wait forever. Example: Thread1 is executing locked code and sends message to a window of thread2. Thread2 receives the message and tries to execute the same code (or other code locked by the same named lock). Result: Thread1 waits in [SendMessage](#) (because thread2 waits in `lock`), and thread2 waits in `lock` (because thread1 waits in [SendMessage](#)). To reset, restart QM.

While waiting in `lock`, the thread does not process messages, even if `opt waitmsg 1` used.

If **mutex** is omitted or "", the lock synchronizes only threads in current process (running program, ie QM or exe). With mutex it can synchronize threads in multiple processes. It can be useful, for example, if 'Run in separate process' is checked (see [exe \(51\)](#)). **lock** creates or opens the mutex. The mutex name must be an unique string, like "QM_mutex_macroname_8469". Error if a kernel object of other type with the same name exists in the system. A mutex with "Global\" prefix can be used by processes in multiple user sessions. With mutex **lock** is about 8 times slower. You can read more about mutexes and other kernel synchronization objects in [MSDN library](#).

Examples

```
lock
str s="test"
s.setfile("$desktop$\test.txt")
lock-

lock 0 "QM_mutex_test1"
...
lock-

lock 0 "" 1000; err end "the code is locked more than 1 s"
...
lock-

lock shared_lock_object
...
lock- shared_lock_object
```

Extensions, dll, categories

In QM you can use several kinds of functions:

- QM intrinsic functions. Listed in the [reference \(52\)](#) topic.
- QM [extensions \(50\)](#). Many are in the System folder in QM. You can create or [download](#) more.
- Functions [exported by qm \(114\)](#) as dll functions, COM interfaces and controls.
- [Windows API \(160\)](#) and other dll functions, [COM \(163\)](#) classes/interfaces and controls.

All QM functions and some API/COM functions are added to [categories](#):

mouse	Mouse functions.	
keytext	Keys, text, clipboard. Added in QM 2.3.2.	
dialog	Message box, other dialogs, popup menu. <ul style="list-style-type: none">• See also: custom dialogs (63).	
file	Run, files, folders, database, XML, CSV.	
window	Windows, controls. <ul style="list-style-type: none">• See also: some Windows API window functions (79).	
control	Controls, other UI objects, menu, Excel, find image.	
time	Wait, date/time.	
internet	Internet functions.	
string	String functions and operators.	
math	Math and some related functions. <ul style="list-style-type: none">• See also: msvcrt math functions (120).	
multimedia	Sound.	
qm	QM items, threads, misc info, etc.	
sys	Various system functions: registry, shutdown, tray icon, processes, environment variables, etc. <ul style="list-style-type: none">• See also: Services (121).	
sysinfo	Get/set system settings and info: version, display, user, etc.	
script	Other languages that can be used in QM: VBScript, C, C#, etc.	
flow	Flow control: if, repeat, select, etc.	
_debug	Debug, errors, optimize, log. Added in QM 2.3.2.	
_operator	Operators. Added in QM 2.3.2.	
_other	Several subcategories. All added in QM 2.3.2.	
	__declaration	Declaration of functions, types, etc.
	__directive	Compiler directives.
	__programming_misc	Misc. QM functions for programming.
	__qm_dll	All dll functions and COM interfaces exported by qm.
	__in_dlgproc	Functions that can be used in dialog procedures.
	__variable_examples	Examples of declaring and initializing variables.
	__array_examples	Examples of working with arrays.
	__rt_option	Run-time options.

To get help for a function, click its name in code editor and press [F1 \(245\)](#). Also look in status bar.

Dll functions exported by QM

These dll functions and interfaces are provided by qm.exe.

In some function descriptions is specified QM version when the function has been added. If not specified - added before QM 2.2.0.

Almost all functions support [Unicode \(238\)](#). If a function does not support Unicode, it is specified in its description.

All file functions support special folders.

Interfaces

[IStringMap \(115\)](#), [CreateStringMap](#)

String map object. A string map is an array of string pairs.

[ICsv \(116\)](#), [CreateCsv](#)

CSV object. CSV is a text file format used to save tables.

[IXml, IXmlNode \(117\)](#), [CreateXml](#)

XML object. XML is a text file format used to save settings and data.

[QM file management functions \(260\)](#)

Functions

String: [StrCompare](#), [StrCompareN](#), [MemCompare](#), [q_stricmp](#), [q_strnicmp](#), [q_memicmp](#), [StrCompareEx](#), [q_sort](#), [DetectStringEncoding](#)

Character type: [isdigit](#), [isalnum](#), [iscsym](#)

Memory: [q_malloc](#), [q_calloc](#), [q_realloc](#), [q_free](#), [q_msize](#), [q_strdup](#)

Keys: [QmKeyCodeToVK](#), [QmKeyCodeFromVK](#), [RealGetKeyState](#), [GetMod](#), [FormatKeyString](#)

QM items: [SilentImport](#), [SilentExport](#), [GetQmItemsInFolder](#)

QM threads: [IsThreadRunning](#), [EnumQmThreads](#), [GetQmThreadInfo](#)

QM menus and toolbars: [GetToolbarOwner](#), [GetLastSelectedItem](#)

Standard menu: [MenuSetString](#), [MenuGetString](#)

Window: [SubclassWindow](#), [SetWinStyle](#), [IsWindow64Bit](#), [AdjustWindowPos](#), [WinTest](#)

Multiple monitors: [MonitorFromIndex](#), [MonitorIndex](#),

DPI-scaled windows: [DpiIsWindowScaled](#), [DpiGetWindowRect](#), [DpiClientToScreen](#), [DpiScreenToClient](#), [DpiMapWindowPoints](#), [DpiScale](#), [DpiGetDPI \(243\)](#)

Rich edit: [RichEditLoad](#), [RichEditSave](#)

Performance - system: [GetCPU](#), [GetDiskUsage](#)

Performance - code: [PerfFirst](#), [PerfNext](#), [PerfOut](#)

Session: [IsLoggedIn](#), [EnsureLoggedIn](#)

System/user info: [IsUserAdmin](#), [GetUserInfo](#)

Process info: [GetProcessUacInfo](#), [GetProcessIntegrityLevel](#), [EnumProcessesEx](#), [ProcessNameTold](#), [ProcessHandleTold](#), [GetProcessExename](#)

Run program: [StartProcess](#), [AllowActivateWindows](#)

File system: [FileExists](#), [GetFileOrFolderSize](#), [GetFullPath](#), [PidToStr](#), [PidFromStr](#)

Shortcut: [CreateShortcutEx](#), [GetShortcutInfoEx](#)

Icon, image: [GetFileIcon](#), [SaveBitmap](#), [LoadPictureFile](#)

Drag and drop: [QmRegisterDropTarget](#), [QmUnregisterDropTarget](#)

Exe: [GetExeResHandle](#), [ExeExtractFile](#), [ExeGetResourceData](#)

Math: [Crc32](#), [Round](#), [RandomNumber](#), [RandomInt](#), [ConvertSignedUnsigned](#)

Tools: [GetQmCodeEditor](#), [InsertStatement](#), [HtmlHelp](#), [RecGetWindowName](#), [EditReplaceSel](#), [QmCodeToHtml](#)

Debug: [CompileAllItems](#), [OutWinMsg](#), [Statement](#), [GetCallStack](#), [q_printf](#)

Other: [SetPrivilege](#), [RegisterComComponent](#), [UnloadDll](#), [RtOptions](#), [QmSetWindowClassFlags](#), [IsValidCallback](#), [RedirectQmOutput](#), [InitWindowsDll](#)

Note that the colored code lines below are not function calling examples. They are copied from declarations and used here to show function name, arguments etc.

Where the return value is not documented, it means that either the return value is not used or the function returns 1 if succeeded, 0 if failed.

Tip: For pointer parameters you can pass variables without operator & (QM 2.4.1).

```
#StrCompare $s1 $s2 [insens]
```

QM 2.3.0.

Compares two strings.

```
#StrCompareN $s1 $s2 n [insens]
```

QM 2.3.0.

Compares maximum **n** characters (bytes) of two strings.

```
#MemCompare $s1 $s2 n [insens]
```

QM 2.3.0.

Compares **n** bytes of two memory blocks. Unlike the StrX functions, can compare binary data. The StrX functions never compare memory after the [terminating null character \(183\)](#).

s1, s2 - strings. Can be str or lpstr variables or string constants. With [MemCompare](#), also can be pointers of any type.
n - number of bytes to compare.
insens - case insensitive if nonzero.

These functions return:

0	s1 and s2 are equal. Null (183) and "" are considered equal.
1	s1 is > s2 . It means that s1 would be below s2 in a sorted list.
-1	s1 is < s2 . It means that s1 would be above s2 in a sorted list.

These functions should be used instead of similar Windows API and msvcrt.dll functions, because:

1. Support [Unicode \(267\)](#) (when QM is running in Unicode mode).
2. Case insensitive comparison is much faster, especially when the strings are ASCII.
3. Don't afraid null strings. Null and "" are considered equal.
4. The return value is strictly 0 or 1 or -1.

To compare strings, you can also use [matchw](#), [findrx](#), [sel](#), [SelStr](#), some str functions and operators. However they cannot be used in sorting.

To compare strings also can be used [CompareString](#), [lstrcmp](#), other Windows API and msvcrt.dll functions, all documented in [MSDN Library \(256\)](#). Note that the case insensitive ANSI versions cannot compare UTF-8 text containing non ASCII characters. In QM, text normally is UTF-8 in Unicode mode.

The following 3 functions exist for backward compatibility. Use the above functions instead.

```
#q_stricmp $s1 $s2
#q_strnicmp $s1 $s2 count
#q_memicmp $s1 $s2 count
```

```
#StrCompareEx $s1 $s2 compare ;;compare: 0 simple, 1 insens, 2 ling, 3 ling/insens, 4 number/ling/insens, 128 date
```

QM 2.3.2.

Compares two strings. Same as [StrCompare](#) but has more options.

compare:

0	Simple, case sensitive. Uses StrCompare to compare strings.
1	Simple, case insensitive. Uses StrCompare to compare strings.
2	Linguistic, case sensitive. Uses StrCmp to compare strings.
3	Linguistic, case insensitive. Uses StrCmpl to compare strings.
4	Number, linguistic, case insensitive. Uses StrCmpLogicalW to compare strings. It compares numbers in strings as number values, not as strings.
128 (flag)	QM 2.3.3. Date. Converts strings to DATE and compares. If cannot convert both strings, compares like without this flag.

```
#q_sort !*base num width func [!*param]
```

QM 2.3.2.

Same as [qsort](#), but allows you to pass some value to the callback function. Read more about [qsort](#) in MSDN. The callback function must begin with:

```
function[c]# param TYPE&a TYPE&b
```

Here TYPE is type of array elements.

A template is available in menu -> File -> New -> Templates.

See also: [ARRAY.sort \(146\)](#), [sub-functions \(182\)](#).

```
#DetectStringEncoding $s ;;returns: 0 ASCII, 1 UTF8, 2 other
```

QM 2.3.2.

Scans the string to determine its character encoding.

Returns:

0	all characters are ASCII (character codes <=127).
1	found UTF-8 (267) characters or BOM, and the string can be considered UTF-8.
2	found characters in range 128-255 that are not valid UTF-8 characters.

```
#isdigit char
```

QM 2.3.0.

Returns a nonzero value if **char** is a digit '0' to '9'. This function replaces the msvcrt.dll function because the later also would return nonzero for superscript characters.

```
#isalnum char
```

QM 2.3.0.

Returns a nonzero value if **char** is a digit '0' to '9' or a letter (including non ASCII). This function replaces the msvcrt.dll function because the later also would return nonzero for superscript characters.

The return value consists of flags: C1_DIGIT (4), C1_ALPHA (0x100), C1_UPPER (1), C1_LOWER (2).

```
#iscsym char ;;ASCII letter, digit or _
```

QM 2.3.2.

Returns 1 if **char** is a valid character for a QM or C++ [identifier \(257\)](#), or 0 if not. This function replaces the msvcrt.dll function because the later also would return nonzero for some non ASCII characters.

For other character types, can be used similar msvcrt.dll functions, for example [isalpha](#), [isprint](#). Also can be used Windows API functions, for example [IsCharAlpha](#), [GetStringType](#). All documented in [MSDN Library \(256\)](#).

These functions don't support [Unicode \(267\)](#). To get Unicode UTF-16 character type, use W versions of Windows API functions, for example [IsCharAlphaW](#).

```
!*q_malloc size ;;QM malloc
```

Allocates **size** bytes of memory. Returns pointer to the allocated memory block.

```
!*q_calloc elemcount elemsize ;;QM calloc
```

Allocates **elemcount*elemsize** bytes of memory and fills with 0. Returns pointer to the allocated memory block.

```
!*q_realloc !*mem size ;;QM realloc
```

Allocates more or less memory. **mem** can be memory allocated with one of these functions, or 0. Returns pointer to the allocated memory block. It can be different or the same as **mem**.

```
q_free !*mem ;;QM free
```

Frees memory allocated with one of these functions. **mem** can be 0.

```
#q_msize !*mem ;;QM _msize
```

Gets size of memory allocated with one of these functions.

```
$q_strdup $s ;;QM_strdup
```

Allocates duplicate string. **s** can be any string, not necessary allocated with one of these functions. The function allocates memory using [q_malloc](#) and copies **s** there.

These functions also can be used with memory that is allocated or will be freed by [str variables \(234\)](#). For this purpose don't use [malloc](#) and other functions from msvcrt.dll, because QM uses other memory allocation functions.

See also: [other memory allocation functions \(148\)](#)

```
#QmKeyCodeToVK $qmkey int&vk ;;Returns number of characters eaten.
```

Converts from [QM key code \(251\)](#) (string) to [Windows virtual-key code \(270\)](#) (integer). **qmkey** may contain more than one QM key code. The function stores virtual-key code of the first QM key code into variable **vk**, and returns number of parsed characters (eg 3 for F12). If **qmkey** is invalid, **vk** receives 0.

QM 2.2.1: Instead can be used [key \(56\)](#). It converts multiple key codes, variables, etc. However then the QM key codes cannot be a variable string.

```
QmKeyCodeFromVK virtualkey str&qmkey
```

Converts virtual-key code to QM key code.

```
#RealGetKeyState vk ;;Returns 1 if the key is pressed. If vk contains flag 0x100 - if toggled. vk is virtual-key code.
```

QM 2.2.1.

Checks if the specified key or mouse button is pressed or toggled. Can be used instead of [ifk \(59\)](#). More reliable than [GetKeyState](#) and [GetAsyncKeyState](#).

Returns 1 if the key is pressed, 0 if not. If **vk** contains flag 0x100 (eg VK_CAPITAL|0x100), returns 1 if the key is toggled, 0 if not.

```
vk - virtual-key code \(270\) 1 to 255. Add flag 0x100 to check toggled state.
```

```
#GetMod ;;Returns: 1 Shift, 2 Ctrl, 4 Alt, 8 Win.
```

Checks whether modifier keys are pressed. The return value is combination of the following values: 1 - Shift is pressed, 2 - Ctrl is pressed, 4 - Alt is pressed, 8 Win is pressed. For example, if Ctrl and Alt are pressed, the return value is 6.

Example

```
if (GetMod&2) out "Ctrl pressed"
```

```
FormatKeyString !vk !mod str&s ;;mod: 1 Shift, 2 Ctrl, 4 Alt, 8 Win.
```

Gets key name for a key and/or modifiers. The text will be like "Ctrl+Page Up".

vk - [virtual-key code \(270\)](#).

mod - modifiers. See [GetMod](#).

s - variable that receives the text. If the variable initially is not empty, appends to the end.

```
#SilentImport $_file [flags] ;;_file: qml file to import; flags: 1 add shared, 2 at bottom. Returns 1 if successful.
```

Imports a [QM file \(17\)](#). Same as menu File -> Import, but without the File Viewer.

_file - full path of .qml file.

flags - see above.

Not available in [exe \(51\)](#).

```
#SilentExport $item $_file [flags] ;;item: qm item or folder name or +id; _file: folder or qml file; flags: 1 no import, 2 no open, 4 no shared, 8 no partial/renamed, 16 read-only, 0x100 make zip. Returns 1 if successful.
```

Exports a [QM item or folder \(19\)](#) to a [QM file \(17\)](#). Same as menu File -> Export, but without the Export dialog.

item - QM item or folder. Can be [name](#), [path](#), [GUID](#) or [+id \(107\)](#).

- QM 2.4.3.8: can be a list of QM items or/and folders, like "Func1[]Func2[]Functions\Public".

_file - full path of .qml file to create. If does not end with ".qml" - folder where to create .qml file.

flags - see above. The same as the checkboxes in the Export dialog.

Not available in [exe \(51\)](#).

```
#GetQmItemsInFolder $folder ARRAY (QMITEMIDLEVEL) &a [flags] ;;1 only direct children
```

QM 2.4.1.

Gets array of [QM items \(19\)](#) in a folder in the list of QM items.

Returns 0 if folder not found. Else returns 1.

folder - folder name, path or +id. Gets all if **folder** is "".

a - variable that receives id and level of QM items that are in **folder** and its subfolders. The order will be the same as in the list of QM items. Levels are 0-based, relative to **folder**. To get item properties, use [qmitem \(107\)](#) or [str.getmacro \(219\)](#).

Not available in [exe \(51\)](#).

Example

```
out
ARRAY (QMITEMIDLEVEL) a; int i
if (!GetQmItemsInFolder("\System\Functions" &a)) end "failed"
for i 0 a.len
  out "%. *m%s" a[i].level 9 _s.getmacro(a[i].id 1)
```

```
#IsThreadRunning $threadName ;;Returns number of threads. name can be QM item name or +id.
```

Returns the number of [threads \(49\)](#) (running instances) of the function or macro.

threadName - QM item name or [id \(107\)](#). If [sub-function \(182\)](#), can be like "ParentName:SubName".

To get the number of threads of current function, instead use [getopt nthreads \(98\)](#), it's faster and don't need to specify name.

Example

```
mac "FunctionX" ;;run function "FunctionX" in separate thread
wait 5
if(IsThreadRunning("FunctionX"))
_out "FunctionX is running"
```

```
#EnumQmThreads [QMTHREAD*arr] [nelem] [flags] [$threadName] ;;flags: must be 0
```

Gets information about currently running QM [threads \(49\)](#), except special threads. Returns the number of threads.

arr - caller-allocated array of **nelem** elements of **QMTHREAD** type. Can be 0. See example.

nelem - number of elements in **arr**.

flags - currently not used and must be 0.

threadName (2.2.0) - QM item name or [id \(107\)](#). Use 0 or "" if need all threads. If [sub-function \(182\)](#), can be like "ParentName:SubName".

This type is used with [EnumQmThreads](#) and [GetQmThreadInfo](#):

```
type QMTHREAD qmitemid threadid threadhandle flags tuid
```

qmitemid - QM item id. Can be used to display item name ([str.getmacro](#) or [qmitem](#)) or end thread ([shutdown](#)).

threadid - thread id. Can be used with [shutdown -6](#) (end thread) and with Windows API functions.

threadhandle - thread handle. Can be used with [wait](#) (wait until thread exits), [shutdown -6](#), and with Windows API functions.

flags - currently is not used.

tuid (QM 2.2.0) - unique thread id. Can be used with [shutdown -6](#). Cannot be used with Windows API functions. Unlike thread id and handle, the same unique id value is not reused for new threads.

Examples

```
Display all threads
int i n=EnumQmThreads(0 0 0 0)
ARRAY(QMTHREAD) a.create(n)
for i 0 EnumQmThreads(&a[0] n 0 0)
_out _s.getmacro(a[i].qmitemid 1)

Is Function55 running?
if(EnumQmThreads(0 0 0 "Function55")) out "running"
```

```
#GetQmThreadInfo handle QMTHREAD&qt ;;Returns 1 on success, 0 on failure. handle=0 - current thread.
```

QM 2.2.0.

Gets information about a QM [thread \(49\)](#). It must not be a special thread.

handle - thread handle. For example, [mac \(100\)](#) returns thread handle when you use it to run a function. If 0 - current thread.

qt - variable of type **QMTHREAD** (described above) that receives the info.

```
#GetToolbarOwner hwndTb ;;Returns handle of a QM toolbar's owner window
```

QM 2.2.0.

Returns handle of the window to which the [toolbar \(23\)](#) is attached. Returns 0 if the toolbar is not attached to a window. Use this function instead of [GetWindow](#) or other Windows API, because QM toolbars are not owned as Windows interprets it.

hwndTb - toolbar window handle. Note that it is handle of toolbar, not of its owner window.

Example (in a toolbar)

```
out my owner :int w=GetToolbarOwner(TriggerWindow); outw w
```

```
GetLastSelectedItem str&label [str&command] [flags] ;;flags: 1 get pidl always
```

QM 2.2.0.

Yo can call this function after showing a [popup menu \(22\)](#) to get clicked item string. To show menu, use [mac \(100\)](#) as function. If [mac](#) returns a nonzero value, call this function. See example.

label - variable that receives selected menu item label string. Can be 0.

command - variable that receives selected menu item code string. With an expand-folder menu - selected file path or [ITEMIDLIST string \(246\)](#). Can be 0.

Not available in [exe \(51\)](#).

Example

```
Show menu "Drives" and get selected item info
if(mac("Drives"))
  str s
  GetLastSelectedItem 0 s 0
  mes s
```

```
MenuSetString hMenu item $s ;;item: > 0 id, <=0 -position
```

QM 2.3.0.

Changes menu item text.

```
#MenuGetString hMenu item str&s ;;item: > 0 id, <=0 -position. Returns 1 on success, 0 on failure.
```

QM 2.3.0.

Gets menu item text.

hMenu - menu handle.

item - menu item id or negative position.

s - sets or receives menu item text.

The menu cannot be a QM user-defined popup menu.

```
#SubclassWindow hwnd newWndProc
```

QM 2.3.0.

Replaces window procedure. Returns address of old procedure. Read more about subclassing in [MSDN Library \(256\)](#).

hwnd - window handle. The window must belong to qm/exe process.

newproc - address of new window procedure (a user-defined function).

- The function must begin with `function hwnd message wParam lParam`.
- The function must end with `ret CallWindowProcW(oldproc hwnd message wParam lParam)`.

Note: If the window is Unicode ([IsWindowUnicode](#) returns 1), the new window procedure receives Unicode UTF-16 strings in standard text messages such as WM_SETTEXT. Most window classes are Unicode. If you instead use

[SetWindowLong/CallWindowProc](#) (ANSI versions), Windows automatically converts Unicode/ANSI, however it is not recommended.

QM 2.3.5. You should instead use [SetWindowSubclass](#). It is safer and easier to use. It is a Windows API, documented in MSDN. A template window procedure is available in menu File -> New.

```
SetWinStyle hwnd style [flags] ;;flags: 0 set, 1 add, 2 remove, 4 exstyle, 8 update nonclient, 16 update client
```

Changes window style or extended style.

hwnd - window handle. Can be top-level or child window.

style - style or extended style.

- Window styles are documented in [MSDN Library \(256\)](#).
- To view available styles and extended styles, type `.WS` .

flags:

0	set.
1	add.
2	remove.
4	style is extended style.
8	update nonclient area.
16	update client area (QM 2.2.1).

See also: [GetWinStyle](#)

Example

```
remove Notepad's title bar:
int h=win("Notepad")
SetWinStyle h WS_CAPTION 2|8
```

```
#IsWindow64Bit hwnd [flags] ;;flags: 1 hwnd is process id
```

QM 2.2.0.

Returns 1 if the window belongs to a 64-bit process.

Before QM 2.4.3 this function returned -1 if failed. Now returns 0.

hwnd - window handle. If **flags** includes 1 - process id.

See also: [_win64 \(144\)](#)

```
#WinTest hwnd $cls
```

QM 2.4.3.

Gets window class name and compares with the specified string. Returns 1 if matches, 0 if not. Returns 0 if **hwnd** is 0 or invalid.

hwnd - window handle. Can be any window (top-level or child).

cls - a string to compare with window class name.

- Can contain [wildcard characters \(271\)](#) *?.
- Can be a list, like `"Class1[]Class2"`. Then returns 1-based index of matching string.

In programming often need just to compare window class name. Then [WinTest](#) is easier to use than [wintest \(77\)](#), which has more parameters and throws error when the handle is invalid.

```
AdjustWindowPos hwnd RECT&r [flags] [monitor] ;;flags: 1 work area, 2 actually move, 4 raw x y, 8 only move
into screen, 16 can resize, 32 monitor is hmonitor. monitor: 0 primary, 1-30 index, -1 mouse, -2 active window, -3 primary,
```

or window handle

QM 2.2.1.

Adjusts window or rectangle coordinates like [mes](#) and other QM functions do. If negative, adjusts so that the window would be at the right or bottom of the screen. If 0, adjusts so that the window would be at the center of the screen. Ensures that whole window is in the screen. Ensures that the window will be in the specified monitor.

If the window is minimized, does not restore it but ensures that the window will be in correct position when restored. If the window is maximized, does not change its position and size, except when need to move to another monitor, but ensures that the window will be in correct position when restored to normal. All this is if **hwnd** is nonzero and flag 2 used. If **hwnd** is nonzero, the function always returns rectangle of normal (not minimized or maximized) window, regardless of its current state.

hwnd - window handle. Can be 0 if you only need to adjust **r**.

r - a RECT variable or 0.

- On input, the variable must contain requested window coordinates in **r.left** and **r.top**. If **hwnd** is 0, **r.right** and **r.bottom** also must be specified, else they are not used. The coordinates are relative to the monitor, except when flag 8 used.
- On output, the variable will contain normal coordinates, ie relative to the primary monitor.
- Can be 0 if flag 8 used or to use **r.left**= 0 and **r.top**=0.

flags:

1	when calculating new coordinates, use work area instead of whole screen.
2	actually move the window. By default, only modifies r .
4	use raw x y relative to the monitor. Negative and 0 x y don't have special meaning.
8	only move the window or rectangle to the nearest position in the work area, if it is not completely there. If hwnd is not 0, r is not used and can be 0. Else r must contain normal coordinates. If monitor is 0, uses the monitor that contains the biggest part of the window or rectangle, or is nearest.
16	resize if needed. By default the function can only move. Should be used only with flag 8.
32	monitor is monitor handle.

monitor - monitor. The same as with [MonitorFromIndex](#) and [_monitor \(144\)](#).

```
#MonitorFromIndex [monitor] [flags] [RECT&r] ;;flags: 1 work area, 32 monitor is hmonitor. monitor: 0
primary, 1-30 index, -1 mouse, -2 active window, -3 primary, or window handle
```

QM 2.2.1.

Returns handle of monitor that is specified by index or some other property. Also can get monitor coordinates.

monitor - monitor. Like with [_monitor \(144\)](#), it can be monitor index 1-30, or 0 (primary), -1 (mouse), -2 (active window), -3 (primary), or window handle.

flags:

1	get work area coordinates.
32	monitor is monitor handle.

r - variable that receives monitor coordinates, relative to the primary monitor. Can be 0 if don't need.

Other functions that can be used to get monitor handle: [MonitorFromWindow](#), [MonitorFromPoint](#), [MonitorFromRect](#). Documented in [MSDN Library \(256\)](#).

```
#MonitorIndex hmonitor ;;Returns 1-based monitor index.
```

QM 2.2.1.

Returns 1-based monitor index. Returns 0 if the handle is invalid.

hmonitor - monitor handle.

```
#RichEditLoad hwndRE $ _file ;;Returns 1 if successful.
```

Loads a file to a rich edit control.

```
#RichEditSave hwndRE $ _file ;;Returns 1 if successful.
```

Saves rich edit control text to a file.

hwndRE - control handle.

_file - file. If ends with .rtf, uses RTF format, else text format.

Rich edit control's class usually is RichEdit20A or RichEdit20W. The control must belong to qm/exe process.

Supports [macro resources \(261\)](#) (QM 2.4.1) and [exe \(51\)](#) resources.

```
#GetCPU ;;Returns % CPU usage, 0 to 100.
```

Returns % of processor usage since last call. When calling first time in thread, result is undefined.

```
#GetDiskUsage ;;Returns % disk usage, 0 to 100.
```

Returns % of disk usage (all physical disks) since last call. That is, % of time spent in disk read/write mode. When calling first time in thread, result is undefined. Should not be called too frequently (with < 100 ms period), or results may be incorrect.

This function should work only for administrators and users in the Performance Log Users group. In my tests it was true on Windows Vista, but the function always works on Windows 7 and 8.

```
PerfFirst
```

```
PerfNext
```

```
PerfOut [flags] [str&sOut] ;;flags: 1 begin collect, 2 collect, 3 out collected
```

QM 2.3.5.

Measures and shows time spent executing parts of macro code. Unit - microseconds.

[PerfFirst](#) and [PerfNext](#) get current time and store in internal global variables.

[PerfOut](#) displays times between [PerfFirst](#) and max 10 following [PerfNext](#).

flags: 1 - 3 can be used to measure time spent in parts of repeatedly executed code. Instead of displaying time in each loop, [PerfOut 2](#) collects times, and finally [PerfOut 3](#) displays sums. Insert [PerfOut 1](#) at the very beginning.

sOut - variable that receives string containing times. If omitted or 0, the function displays the string in QM output.

See also: [perf \(91\)](#)

Examples

```
simple
PerfFirst
rep 1000
  _find("one two three four" "four")
PerfNext
rep 1000
  _findrx("one two three four" "four")
PerfNext
PerfOut

with collecting times
PerfOut 1 ;;start collecting
rep 10
  _PerfFirst
  win("" "Shell_TrayWnd")
  _PerfNext
  win("not found")
  _PerfNext
  _PerfOut 2 ;;collect. Does not show times now.
```

```
_0.001 ;;can be more code, its time will not be included
PerfOut 3 ;;show collected
```

#IsLoggedIn ;;Returns: 0 not logged on, 1 normal, 2 locked, 3 switched off, 4 custom desktop.

Checks whether QM is running on interactive desktop. Returns 0 if not logged on, 1 if normal (runs on interactive desktop), 2 if locked (directly or by a secure screen saver), 3 if switched off, 4 custom desktop.

#EnsureLoggedIn [unlock] ;;Returns: 0 cannot ensure, 1 normal, 2 successfully unlocked.

Checks whether QM is running on interactive desktop. If computer is locked, and **unlock** is nonzero, attempts to unlock. Returns 0 if cannot ensure interactive desktop, 1 if normal, 2 if successfully unlocked. The unlock method and settings are the same as specified in the [Unlock Computer \(268\)](#) dialog.

Unlocking is unavailable in exe and in portable QM. Then **unlock** value is ignored.

Some triggers (scheduler, event log, etc) may launch the macro while QM is not running on interactive desktop (e.g., while computer is locked). Then the macro runs in the background. However, many commands (keyboard, mouse, etc) then don't work, because background users cannot access keyboard, mouse and monitor. To ensure that the macro will not run in the background, you can check the checkboxes in Properties -> Macro properties or use [EnsureLoggedIn](#) or [IsLoggedIn](#) in the macro.

Example

```
If QM is not running on interactive desktop, try to unlock computer; if it fails, exit
if(!EnsureLoggedIn(1)) ret
```

#GetUserInfo str&s [flags] ;;flags: 0 user name, 1 computer name

QM 2.3.0.

Gets user name or computer name.

s - variable that receives it.

#IsUserAdmin ;;Returns 1 if QM (or exe) is running as administrator

QM 2.2.0.

Returns 1 if current process (QM or exe) is running as administrator.

#GetProcessUacInfo hwnd [flags] ;;returns: 0 XP or failed, 1 UAC off or non-admin account, 2 admin, 3 uiAccess, 4 user; flags: 1 hwnd is process id (0 = current process)

QM 2.2.0.

Returns some UAC ([Vista/7/8/10 \(277\)](#)) info of a process (running program):

Return values:

0	Older OS (not Vista/7/8/10), or failed.
1	UAC turned off, or the process is running as standard user on standard user account.
2	The process is running as administrator.
3	The process is running not as administrator but has uiAccess privileges.
4	The process is running on administrator account but not as administrator.

hwnd - handle of a window that belongs to the process. If **flags** includes 1, then **hwnd** must be process id. You can use process id 0 to specify current process.

See also: [IsUserAdmin](#) (see above)

```
#GetProcessIntegrityLevel hwnd [flags] ;;returns: 0 XP or failed, 1 Low, 2 Medium, 3 High, 4 System; flags: 1
hwnd is process id (0 = current process)
```

QM 2.2.0.

Returns **integrity level (277)** of a process:

0	Older OS (not Vista/7/8/10), or failed.
1	Low.
2	Medium.
3	Administrator or uiAccess.
4	System.

hwnd - handle of a window that belongs to the process. If **flags** includes 1, then **hwnd** must be process id. You can use process id 0 to specify current process.

```
EnumProcessesEx ARRAY(int)&pids [ARRAY(str)&names] [flags] ;;flags: 1 full path, 2 current user session
only
```

Gets names and/or process ids of all processes (running programs).

pids - variable that receives process ids. Can be 0 if don't need.

names - variable that receives process names. Can be 0 if don't need.

flags:

1	get full paths. <ul style="list-style-type: none"> Note: cannot get full paths of processes running in other user sessions if QM is running not as administrator.
2 (QM 2.2.0)	enumerate only processes running in current user session (fast user switching).

Example

```
ARRAY(str) a
EnumProcessesEx 0 a 0
for(_i 0 a.len) out a[_i]
```

```
#ProcessNameToId $exename [ARRAY(int)&allPids] [flags] ;;allpids can be 0; flags: 1 current user session
only
```

Returns process id.

Returns 0 if the process does not exist (the program is not running).

QM 2.2.0. Added parameter **flags**.

exename - program name or full path.

allPids - variable that receives process ids of all matching processes. Can be 0 if don't need.

flags:

1	find only processes running in current user session.
---	--

```
#ProcessHandleToId hProcess
```

QM 2.4.2.

Gets process id from process handle.

Example

```
_Handle hp=run("notepad.exe")
int pid=ProcessHandleToId(hp)
```

```
#GetProcessExename pid str&exename flags ;;flags: 1 full path. Returns: 1 success, 0 failed.
```

QM 2.2.1.

Gets name or full path of the executable file that started specified process (running program).

pid - process id.

exename - variable that receives the name.

The process here must be identified using process id. To use window instead, can be used [str_getwinexe \(224\)](#).

See also: [_qmdir \(144\)](#)

```
#StartProcess flags $path [$cl] [$defDir] [int&hProcess] ;;flags: 0 as QM, 1 medium, 2 high, 3 high
with consent, 4 low, 5 as QM -uiAccess, 16 get process id, hibernate showcmd
```

QM 2.2.0.

Runs a program with specified privileges (administrator, user, etc). Mostly useful on [Vista and later \(277\)](#), but can be used on all OS.

flags - integrity level (IL) that you need, and other flags, see above. The IL values are described in the [Make Exe topic \(51\)](#).

path - executable file path.

cl - command line.

defDir - default directory.

hProcess - variable that receives process handle. Later close it with [CloseHandle](#). With flag 16, gets process id instead; don't close it.

QM 2.2.1. The highest byte of **flags** can contain suggested window state (maximized, hidden, etc), like with [run \(64\)](#). To store a value to the highest byte, use `value<<24`. For example, to run the program hidden, use `flags | (16<<24)`.

Not available in [exe \(51\)](#).

There are some limitations in [portable QM \(259\)](#).

```
#AllowActivateWindows ;;Returns 1 if successful
```

By default, Windows does not allow background programs to activate windows. QM is usually running in the background. Use this function to temporarily allow QM to activate windows.

Tip: After calling this function, you also can call [AllowSetForegroundWindow -1](#) to allow other processes activate windows.

```
#FileExists $_file [flags] ;;flags: 0 file, 1 folder, 2 file or folder. Returns: 0 not found, 1 file, 2 folder.
```

QM 2.4.0.

Checks if the specified file, folder or drive exists.

0	Does not exist, or failed.
1	Exists, and is file.
2	Exists, and is folder or drive.

_file - full or relative path of a file, folder or drive.

flags:

0	Must be file. Return 0 if exists but is folder or drive.
---	--

(default)	
1	Must be folder or drive. Return 0 if exists but is file.
2	Can be file, folder or drive.

Internally uses Windows API function [GetFileAttributes](#). Returns 0 if it fails for any reason, eg when the file exists but cannot be accessed because of security settings. Does not support wildcard characters, use [Dir](#) class instead (dialog "Get file info").

```
%GetFileOrFolderSize $file
```

QM 2.2.0.

Returns size of the specified file, folder or drive.

If folder, calculates sum of sizes of all files in the folder and subfolders. It is slow.

If drive, gets used size. It is fast. Uses Windows API function [GetDiskFreeSpaceEx](#), except for network drives.

```
GetFullPath $sAny $str&$sFull
```

QM 2.3.0.

Gets full path of a file or folder. Expands special folders, replaces "..", etc. Does not check whether the file or folder exists.

\$sAny - full or relative path. If it is relative path, prepends current directory. Can contain ".." (parent folder), "." (same folder). If it is a drive letter without "\", appends "\". Read more: [File names and paths](#).
\$sFull - variable that receives full path.

This function can be used to create full path from user-entered file name. For example, if user entered "..\f3\file.txt", and your default folder is "c:\f1\f2\", join them ("c:\f1\f2\..\f3\file.txt") and pass to [GetFullPath](#). Result will be "c:\f1\f3\file.txt".

To know whether a file path is full or relative, can be used function [PathIsRelative](#).

```
#PidlToStr ITEMIDLIST*pidl $str&$ flags ;;Returns 1 if successful. flags: 1 must be path, 2 must be "::ITEMIDLIST", 4 path can be URL, 8 path can be "::{CLSID}", 16 path must be display name
```

Converts **ITEMIDLIST** to string - file system path, ":: ITEMIDLIST", URL, "::{CLSID}" or display name. Variable **s** receives the string.

[Read more \(246\)](#).

QM 2.4.5. Added flags 4, 8, 16.

If used flag 2, gets ":: ITEMIDLIST".

Else if used flag 16, gets display name, eg "qm.exe" or "Control Panel".

Else if **pidl** is of a file system object, gets its path.

Else if **pidl** is of a URL and used flag 4, gets the URL.

Else if **pidl** is of a virtual object that has a CLSID and used flag 8, gets "::{CLSID}".

Else if not used flag 1, gets ":: ITEMIDLIST".

Else fails.

```
ITEMIDLIST*PidlFromStr $s
```

Converts path to **ITEMIDLIST**. Use [CoTaskMemFree](#) to free it.

s can be a file system path, ":: ITEMIDLIST", URL or "::{CLSID}". Cannot be a display name, like "Control Panel".

[Read more \(246\)](#).

```
#CreateShortcutEx $shortcut $SHORTCUTINFO&$si ;;Returns 1 if successful.
```

Creates shortcut. All **si** members except **target** are optional. You can also use simpler function [CreateShortcut](#).

```
#GetShortcutInfoEx $shortcut SHORTCUTINFO&si ;;Returns 1 if successful.
```

Gets shortcut's parameters (target path, icon, etc).

shortcut - shortcut file (.lnk) path.

si - variable that contains or receives shortcut properties.

```
type SHORTCUTINFO ~target ~param ~initdir ~descr ~iconfile iconindex showstate @hotkey
```

Some members:

target - file/folder path, or [ITEMIDLIST string \(246\)](#).

hotkey - hotkey.

- The [low-order \(250\)](#) byte is [virtual-key code \(270\)](#).
- The high-order byte is combination of [flags \(247\)](#) [HOTKEYF_ALT](#), [HOTKEYF_CONTROL](#), [HOTKEYF_SHIFT](#), [HOTKEYF_EXT](#).

Example

Create shortcut with all possible attributes

```
SHORTCUTINFO si.target="$system$\notepad.exe"
si.iconfile="shell32.dll"
si.iconindex=4
si.descr="test descr"
si.hotkey=HOTKEYF_CONTROL<<8|VK_F6
si.initdir="$my qm$"
si.param="-p"
si.showstate=SW_MAXIMIZE
CreateShortcutEx("$desktop$\test.lnk" si)
```

```
#GetFileIcon _file [index] [flags] ;;Returns icon handle if successful. Later call DestroyIcon. flags: 1 large, 2
don't use shell icon, 4 cursor, 8 create empty if fails
```

Extracts icon from an icon file, or gets icon for a file that itself does not contain icons. Returns icon handle. Later destroy it using [DestroyIcon](#). Or assign to a [_Hicon](#) variable.

_file - any file or folder.

- Can be full path, [filename \(246\)](#) or [ITEMIDLIST string \(246\)](#).
- Can be just file type extension, like ".txt". Gets icon of that file type.

index - 0, or icon index (0-based).

- If negative, **index** is interpreted as negative resource id.
- Alternatively, icon index or -id can be appended to **_file**. Example: "shell32.dll,3".

flags:

1	Get large icon (32x32). By default, gets small icon (16x16).
2	Don't use shell icon. Read more below.
4 (QM 2.3.0)	Get cursor. Returns cursor handle. Later destroy it with DestroyCursor .
8 (QM 2.4.3)	Create empty/transparent icon if failed or if _file string is "".

For ico files, always extracts icon from the file. For other files, extracts icon from the file if **index** is nonzero, or icon index is appended to **_file**, or flag 2 used. Otherwise, gets shell icon (icon that would be displayed in Windows Explorer).

Supports [macro resources \(261\)](#) (QM 2.4.1) and [exe \(51\)](#) resources. Adds (to exe) images of all sizes and colors that are in the icon file or resource, not only of the specified size.

QM 2.3.0. You can specify custom size in [high-order \(250\)](#) word of **flags**. Flag 1 then is ignored. If icon of that size is unavailable, gets icon of nearest available size.

See also: [GetWindowIcon](#).

Example

```
int hi=GetFileIcon("shell32.dll,3" 0 1)
int dc=GetDC(GetQmCodeEditor)
DrawIconEx dc 100 100 hi 0 0 0 0 DI_NORMAL
ReleaseDC 0 dc
DestroyIcon hi
```

```
#SaveBitmap hbitmap $ _file [flags] ;;Returns 1 if successful. flags: 1 _file is IStream.
```

Saves bitmap to a file.

hbitmap - bitmap handle.
_file - .bmp file.

```
#LoadPictureFile _file [flags] ;;Returns bitmap handle. flags: 1 return IPicture
```

Loads bitmap from a bmp, jpg or gif file. QM 2.3.4: also can be png.

Supports [macro resources \(261\)](#) (QM 2.4.1) and [exe \(51\)](#) resources.

```
#QmRegisterDropTarget hwndTarget hwndNotify flags ;;flags: 1 notify on enter, 2 notify on over, 4 notify on
leave, 16 need only files, 32 need .lnk
```

QM 2.2.0.

Registers a window as an OLE drag and drop target. Returns 1 on success, 0 on failure.

Can be used to receive paths of dropped files, other shell objects ([ITEMIDLIST \(246\)](#)) and Internet links. Also can be used to receive dropped data of other formats (text, HTML, etc).

Unlike WM_DROPFILES, this method works well on Windows [Vista \(277\)](#) and later. WM_DROPFILES does not work under UAC, because it does not allow to drag and drop files between processes of different integrity level (IL). QM normally has high IL, but Windows Explorer has medium IL. However in [exe \(51\)](#) running with high IL this method does not work too.

The information below may be hard to understand. Also, most of it is not useful in most cases. You can start from the example. Also, you can ask about it in the QM forum.

[QmRegisterDropTarget](#) parameters:

hwndTarget - handle of the target window. It can be a top level window (e.g. a custom dialog) or a child window (e.g. an edit control in a dialog).

hwndNotify - handle of the window that will receive notifications. For example, if the target is a control in a dialog, you'll probably want to receive notifications in the dialog procedure, and therefore use hDlg for **hwndNotify**. Can be 0 if it is the same window as **hwndTarget**.

flags:

1	notify on enter.
2	notify on over.
4	notify on leave.
16	need only files.
32	need .lnk (don't extract shortcut target).

On drag and drop events, the **hwndNotify** window receives **WM_QM_DRAGDROP** message. By default, the message is sent only on drop, unless flags 1, 2, and/or 4 are used. The message is sent with the following info:

wParam - 0 on drag enter, 1 on drag over, 2 on drag leave, 3 on drop.

IParam - address of variable of QMDRAGDROPINFO type.

```
type QMDRAGDROPINFO hwndTarget IDataObject 'dataObj' ARRAY (FORMATETC) formats keyState
POINT 'pt' effect str 'files'
```

hwndTarget - target window handle.

dataObj - IDataObject interface pointer. Available on drag enter and drop. Can be used to extract data of various formats. In QM, only GetData function is valid, and can retrieve only data described as TYMED_HGLOBAL. In exe, dataObj is fully valid.

formats - array of available formats. Available on drag enter and drop. A format is described by a FORMATETC type.

keyState - modifier keys and mouse buttons.

pt - mouse pointer position.

effect - combination of available drop effects (copy, move, link, etc). Available on drag enter, over and drop. If you did not use flag 16, you should remove effects you don't accept. To remove, use the & operator.

files - list of dropped files, other shell objects and Internet links. Valid only on drop. In most cases, it is the only member you need.

IDataObject, FORMATETC, keyState, effect are documented in [MSDN Library \(256\)](#). Look for IDropTarget and IDataObject interfaces. In most cases you will not need it.

Usually you'll need only **files**. To register drop target window, call [QmRegisterDropTarget](#) with **flags** 16. Then the message will be sent only on drop, and QM sets proper effect. In the window or dialog procedure, assign IParam to a QMDRAGDROPINFO* variable, get **files**, and return 0. See example.

If you also need other notifications, set other flags. Also, on **WM_QM_DRAGDROP** return 1. To return 1 from a dialog procedure, use `ret DT_Ret(hDlg 1)`. Always return 1 if you have modified **effect**.

Toolbars: By default, the child toolbar control is a drop target. To replace default drag and drop feature, use something like this in [hook procedure \(27\)](#) on **WM_INITDIALOG**: `QmRegisterDropTarget(wParam hwnd 16)`. Or you can register the toolbar window (`QmRegisterDropTarget(hwnd 0 16)`) as a drop target. It is covered by the child toolbar control, so you would have to make the control smaller. On **WM_INITDIALOG**, resize the control (**sz**), and on **WM_SIZE**

Example

Function DropTargetExampleDialog

```
\Dialog_Editor
function# hDlg message wParam lParam
if(hDlg) goto messages

str controls = "3"
str e3
if(!ShowDialog("DropTargetExampleDialog" &DropTargetExampleDialog &controls)) ret

BEGIN DIALOG
0 "" 0x90C80A44 0x100 0 0 223 41 "Dialog"
1 Button 0x54030001 0x4 4 22 46 14 "OK"
2 Button 0x54030000 0x4 54 22 46 14 "Cancel"
3 Edit 0x54030080 0x200 4 4 216 14 ""
END DIALOG
DIALOG EDITOR: "" 0x2020006 "" ""

ret
messages
sel message
case WM_INITDIALOG
QmRegisterDropTarget(id(3 hDlg) hDlg 16)
case WM_DESTROY
case WM_COMMAND goto messages2
case WM_QM_DRAGDROP
QMDRAGDROPINFO& di=+lParam
set edit box
str s.getl(di.files 0) ;;get first file
s.setwintext(id(3 hDlg))
to get all dropped files, you can use eg foreach. Example: foreach(s di.files) out s
```

```
ret
messages2
sel wParam
case IDOK
case IDCANCEL
ret 1
```

```
#QmUnregisterDropTarget hwndTarget
```

QM 2.2.0.

Unregisters drop target. Optional. Returns 1 on success, 0 on failure.

```
#GetExeResHandle
```

Returns module handle that can be used with API resource functions. Useful in [exe \(51\)](#).

- When the macro runs as separate exe file, gets exe module handle, which is the same as [_hinst \(144\)](#).
- When the macro runs in separate process and uses .qmm file, gets .qmm file handle.
- When the macro runs in QM and there is associated .exe or .qmm file, gets .exe or .qmm module handle.
- Else returns 0.

```
#ExeExtractFile resId $dest [flags] [resType] ;;Returns: 1 success, 0 failed. flags: must be 0
```

QM 2.2.0.

Extracts a file added to [exe \(51\)](#). The file can be added to exe using syntax ":resourceid filepath" (QM 2.3.0), or [#exe addfile \(179\)](#), or using a resource editor. Returns 1 on success, 0 on failure. This function works only in exe. It should not be used when the macro runs in QM.**resId** - resource id. Use the same **resId** as with [#exe addfile](#) or in ":resourceid filepath".**dest** - destination file. To extract to exe's folder, use "\$qm\$\filename.ext" or just "filename.ext". To extract to temporary folder, use for example "\$temp qm\$\myexename\filename.ext". If the folder does not exist, creates. If the file already exists, compares resource data with existing file data, and replaces existing file only if it is different.**flags:**

1	QM 2.3.4. Load dll and return handle.
---	---------------------------------------

resType (QM 2.3.4) - resource type. If omitted or 0, uses RT_RCDATA (10).**resType** and/or **resId** can be integer 1 to 0xFFFF or string (eg "WAVE").*Example*

```
#if EXE=1 ;;EXE is 1 if compiling exe, 2 if qmm, 0 if the macro runs in QM

#exe addfile "$desktop$\test.txt" 1
if(!ExeExtractFile(1 "$temp qm$\testexe\test.txt")) ret
run "$temp qm$\testexe\test.txt"

#else

run "$desktop$\test.txt"

#endif
```

See also code in function [ExeQmGridDll](#). It adds a dll to exe, and extracts/loads at run time.

```
#ExeGetResourceData resType resId str&data int*size ;;Returns: 1 success, 0 failed. resType: if 0, uses RT_RCDATA
```

QM 2.2.0.

Loads [exe \(51\)](#) resource data. Returns 1 on success, 0 on failure.

This function is used in exe. It also works while the macro runs in QM, if the exe is already created.

resType - resource type. If 0, uses RT_RCDATA.

resId - resource id.

data - variable that receives data.

- Can be 0. Then the function returns pointer to resource data in memory. It is read-only.

size (QM 2.3.4) - variable that receives resource size. Use when you need resource size when **data** is 0.

resType and/or **resId** can be integer 1 to 0xFFFF or string (eg "WAVE").

Example

```
#exe addfile "$desktop$\test.txt" 1
str s
if(!ExeGetResourceData(0 1 s)) ret
out s
```

#Crc32 !*data nbytes [flags] ;;Returns CRC of data. If data is string, nbytes can be string length or -1. flags: 1 data is file (nbytes not used).

Calculates CRC checksum (hash) of a string or binary data.

flags (QM 2.3.2):

1 **data** is file path. Calculates CRC of file data. **nbytes** not used and can be 0.

See also: [str.encrypt \(209\)](#) (MD5).

^Round ^number [cDec] ;;Returns number rounded to cDec digits after decimal point

QM 2.2.1.

Returns **number** rounded to **cDec** digits after decimal point. Can be used to convert a floating-point (double) value to nearest int value.

This and other math functions are added to the [math category \(159\)](#).

Examples

```
int i
i=1.1 ;;1
i=1.9 ;;1
i=Round(1.1 0) ;;1
i=Round(1.9 0) ;;2
i=Round(1.5 0) ;;2 (.5 rounds to the nearest even number)
i=Round(2.5 0) ;;2 (.5 rounds to the nearest even number)
double d=Round(1.5555 2) ;;1.56
```

^RandomNumber ;;Returns random number between 0.0 and 1.0, not including them

QM 2.2.1.

Random number generator. Returns a random value between 0 and 1, not including them. The value can be multiplied to get a random double number in any range.

#RandomInt bound1 bound2 ;;Returns random number between bound1 and bound2, including them

QM 2.2.1.

Returns a random integer value between **bound1** and **bound2**, including them.

```
%ConvertSignedUnsigned int'value [valueType] ;;valueType: 0 unsigned int, 1 signed byte (char), 2 signed word (short)
```

QM 2.3.2.

Converts from an integer type unavailable in QM to [long](#) (64-bit integer).

QM does not have these integer types:

- 32-bit unsigned. In C++ it is DWORD, UINT, unsigned int, WPARAM, etc. The matching QM type is [int](#), but it is signed.
- 8-bit signed. In C++ it is char. The matching QM type is [byte](#), but it is unsigned.
- 16-bit signed. In C++ it is short. The matching QM type is [word](#), but it is unsigned.

Values of these types can be returned by dll functions. Variables of the matching QM types can store them. However, when assigning to a bigger type using operator =, possible incorrect conversion. This function converts correctly.

valueType:

0	value is unsigned 32-bit integer. The return value must be stored into a long variable.
1	value is signed 8-bit integer. The return value must be stored into a int or long variable.
2	value is signed 16-bit integer. The return value must be stored into a int or long variable.

Examples

```
int i; long lo
i=0xffffffff
lo=i; out lo ;;-1
lo=ConvertSignedUnsigned(i); out lo ;;4294967295

word w; int j
w=0xffff
j=w; out j ;;65535
j=ConvertSignedUnsigned(w 2); out j ;;-1

byte c; int k
c=0xff
k=c; out k ;;255
k=ConvertSignedUnsigned(c 1); out k ;;-1
```

#GetQmCodeEditor

QM 2.3.0.

Returns QM code editor control handle. If there are two controls, returns one that is currently active, ie has focus or had focus more recently. Use this function instead of `id(2210 _hwndqm)`.

In QM 2.3.0 and later, QM code editor, output and some other controls are based on Scintilla control. In older versions - rich edit control. Therefore, if you have macros created in older QM versions that use rich edit control messages with these controls, they may stop working. Also, str.getwintext and str.setwintext now does not work because the new control does not support WM_GETTEXT and WM_SETTEXT messages. Now use [Scintilla messages](#). The required constants are in SCI reference file. Also, control id 2203 has been changed to 2210 (primary) and 2211 (right/bottom). See also: [InsertStatement](#) (below).

[Scintilla](#) is a free source code editing control created by Neil Hodgson. QM currently uses Scintilla version 3.3.9, upgraded from 2.23 in QM 2.4.1.

Not available in [exe \(51\)](#).

Example

```
str s
int h=GetQmCodeEditor
int lens=SendMessage(h SCI.SCI_GETTEXTLENGTH 0 0)
```

```
s.fix(SendMessage(h SCI.SCI_GETTEXT lens+1 s.all(lens)))
out s
```

```
InsertStatement $s [label] [icon] [flags] ;;flags: 1 simple, 2 set focus, 4 don't indent
```

QM 2.3.0.

Inserts one or more lines of QM code in the QM editor, at current position. Inserts as separate line(s), properly indents, etc.

s - one or more QM statements (commands).

label - label of menu item or toolbar button. Default: "Label". With other item types, it is appended as comments.

icon - icon file.

flags:

1	simply insert the string in current position, not necessary as separate line.
2	set focus.
4	don't indent.

When inserting multiline text in menu/toolbar/autotext, creates new [sub-function \(182\)](#) with m attribute.

QM uses this function to insert code created by the floating toolbar dialogs, recorded code, etc.

Not available in [exe \(51\)](#).

```
#HtmlHelp hwndCaller $pszFile uCommand dwData ;;HtmlHelp that supports UTF-8. For reference, search for
HtmlHelp in MSDN Library.
```

Controls HTML help. For reference, search for HtmlHelp in [MSDN Library \(256\)](#).

```
#RecGetWindowName hwnd $str&s [flags] [$str&sParent] [$str&sComments] [$str&sComments2]
```

Gets window or child window expression as it would be recorded. Can be [win](#), [child](#) or [id](#) function. Returns 1, or 0 if failed.

hwnd - window handle.

s - variable that receives the string.

flags - not used, must be 0.

sParent (QM 2.3.4) - if used, for child windows receives "win(...)" string (parent window), and **s** will be like "id(5 {window})"; for top-level windows will be empty.

sComments (QM 2.3.4) - if used, receives child window description.

sComments2 (QM 2.3.4) - if used, receives description of child accessible object from mouse.

QM 2.3.0. Works well with child windows too. Formats either [id](#) or [child](#) function, which is better in that case.

Not available in [exe \(51\)](#).

```
EditReplaceSel hDlg cid $s [flags] ;;flags: 1 replace all, 2 set focus, 4 move caret to end
```

QM 2.3.0.

Inserts or replaces text in an Edit control. Unlike `str.setwintext`, can replace only selection, or just insert if there is no selection. Also, Undo will work. The function is intended to use in QM dialogs, but also works with edit controls in other threads and processes, except flag 2.

hDlg, cid - can be one of:

- Edit control handle and 0.
- Parent window handle and Edit control id.

s - text.

```
QmCodeToHtml $sIn $str&sOut itemId bbcode flags
```

QM 2.3.2.

Formats HTML or BBCode or QM forum code from string containing QM code, like menu -> Edit -> Other Formats.

sIn - string containing QM code.

sOut - variable that receives the formatted string.

itemId: 0 macro/function, -1 menu/toolbar, -2 autotext list. Or [QM item id \(107\)](#).

bbcode: 0 HTML, 1 qm forum code, 2 standard bbcode.

flags:

1	with css.
2	with <pre>.
4	don't escape tabs/spaces.
8	 instead of <div>.

- All flags used only with HTML.

Not available in [exe \(51\)](#).

```
CompileAllItems $folder [$excludeList]
```

[Compiles \(47\)](#) multiple macros and functions. Purpose - quickly check for errors multiple macros and functions.

folder - name or id (<0x10000) of folder to compile. If 0, compiles all. If it is not folder, compiles all starting from item with this name/id.

excludeList (QM 2.3.4) - multiline list of names of functions and macros that the function must not compile.

Compiles only macros, functions and member functions. Not menus, toolbars, autotexts. Does not recompile if already compiled. Should be called soon after QM startup.

See also: [#compile \(174\)](#)

Not available in [exe \(51\)](#).

```
$OutWinMsg message wParam lParam [str&sOut] [hwnd]
```

QM 2.3.1.

Formats string from a Windows message. Can be used in window and dialog procedures to display received messages.

message - message.

wParam and **lParam** - message parameters. The function uses them to get command/notification codes of some special messages, such as WM_COMMAND and WM_NOTIFY.

sOut - variable that receives the string. If omitted or 0, the function displays the string in QM output.

flags (QM 2.4.1) - a window handle. The function will append it and window class and name to the string.

QM 2.3.3. Returns the formatted string. If **sOut** is +2, does not display it in QM output.

Not available in [exe \(51\)](#).

```
#Statement [caller] [statOffset] [str&statement] [int&itemId] [flags] ;;flags: 1 include indirect callers
```

QM 2.3.1. Added to [exe \(51\)](#) in QM 2.4.3.4.

Gets currently executing [statement \(44\)](#) info.

caller - 0 current function, 1 caller, 2 caller's caller, and so on.

statOffset - 0 current statement, 1 next, -1, previous, etc. If out of range, gets nearest (first or last) statement. Note that some statements, eg declarations, are not used at run time. To see used statements, run the macro in debug mode.

statement - variable that receives statement code (text). Can be 0 if don't need (faster). Not used in exe.

itemId - variable that receives QM item id. Can be 0 if don't need.

flags:

1	include indirect callers. <ul style="list-style-type: none"> Direct caller (151) is macro or function that explicitly called current function by name.
---	---

Returns:

>=0	Offset of the statement in code.
-1	There is no caller at the specified depth.
-2	The QM item is deleted or encrypted. Gets only itemId .

The results may be incorrect if code of the function changed after compiling.

See also: [deb \(99\)](#), [GetCallStack](#) (see below), [other debug functions \(47\)](#), [getopt nargs \(98\)](#)

```
GetCallStack [str&cs] [flags] ;;flags: 1 no code, 2 no formatting
```

QM 2.3.5.

Gets function call stack.

It is a multiline list of function names. The first is current function, then its caller, caller's caller and so on.

cs - variable that receives results. If omitted or 0, displays in QM output.

flags:

1	Get only function names. By default also gets current statement code (text). In exe never gets code.
2	Get plain text. By default the text is formatted for QM output.

The statement code may be incorrect if code of the functions changed after compiling.

Names of [sub-functions \(182\)](#) are full (like "<00001>SubName") with flag 2. Without this flag, names are like "SubName".

See also: [deb \(99\)](#), [Statement](#) (see above), [other debug functions \(47\)](#), [getopt \(98\)](#), [qitem \(107\)](#), [str.getmacro \(219\)](#)

```
q_printf $format ...
```

QM 2.3.2.

Shows formatted text in QM output like [out](#). Use this function where you cannot use [out](#). For example in C compiled code ([__Tcc](#)) instead of [printf](#).

```
#SetPrivilege $privilege ;;Returns 1 if successful.
```

Some Windows API functions require certain privileges. This function enables a privilege in current process. Note that not all privileges can be enabled, especially when QM is running not as administrator.

```
#RegisterComComponent $path flags ;;flags: 1 unregister, 2 check extension (must be dll or ocx), 4 run as admin, 8 show error/success message box
```

QM 2.2.0.

Registers or unregisters a COM component (dll, ocx). Returns 1 if successful, 0 if not.

Requires admin privileges. Use flag 4 to run as admin (possibly with a "run as" or consent dialog, especially in exe).

QM 2.2.1. Internally runs regsvr32.exe. Now can also register 64-bit dlls. Added flag 8.

To register .NET COM components, instead use regasm.exe; it is in .NET runtime folder; more info is on the internet.

See also: [using COM components without registration \(163\)](#).

```
#UnloadDll $dllName
```

QM 2.3.2.

Unloads a delay-loaded dll (declared with [dll \(153\)](#)-).

Returns 1 if successfully unloaded or still not loaded.

Returns 0 if the dll is not declared or is declared as not delay-loaded.

When you declare a dll with [dll](#)-, QM loads it on demand: when a delay-loaded function from it is called first time. Unloads when QM exits or when you open another or reopen current [QM file \(17\)](#). This function allows you to unload the dll at any time. It does not remove the dll declaration. The functions can be loaded on demand again.

This function is not thread-safe. If other threads use functions of this dll at the same time, QM may crash etc. Use it only when debugging a dll, for example a dll created at run time with [__Tcc](#) class. Don't need to unload dlls for other purposes.

Not available in [exe \(51\)](#).

```
RtOptions mask RTOPTIONS&x ;;mask: 0 get all, 1 flags, 2 spe_for_macros, 4 waitcpu_time, 8 waitcpu_threshold, 16 web_browser_class, 32 net_clr_version, 64 opt_
```

QM 2.3.4.

Sets or gets global run-time options. You will probably call this function from [init2](#) or other function that runs when QM or your exe starts.

mask - which options to change. One or more flags listed above.

x - variable that contains or receives run-time options. The function uses only members specified in **mask**. If **mask** is 0, the variable receives current options.

```
type __RTO_OPT !keysync !keymark !keychar !hungwindow !clip
type RTOPTIONS
    flags spe_for_macros waitcpu_time waitcpu_threshold
    str'web_browser_class str'net_clr_version
    __RTO_OPT'opt_macro_function __RTO_OPT'opt_menu_toolbar __RTO_OPT'opt_autotext
```

flags:

0x8000	Prevent activating message box (mes) windows by default.
--------	--

spe_for_macros - default [autodelay \(90\)](#) for macros and menu/toolbar/autotext items. Not applied to functions. Default 100.

waitcpu_time - additional autodelay, applied only after [\(97\)opt waitcpu 1](#). Default 1000.

waitcpu_threshold - default CPU threshold, %. Used by [opt waitcpu 1](#) and [wait ... P](#) (wait for CPU). Default 20.

web_browser_class - class name of your default web browser window. Used by [web \(94\)](#) and [htm \(86\)](#), when window not specified. Use only if your default web browser is not Internet Explorer but is IE-based. Don't use for Firefox, Chrome, Opera.

net_clr_version (QM 2.4.1) - .NET CLR version to load when/if will need it. Used by [CsScript](#) class and related functions (compile/execute C# or VB.NET code).

- Can be:
 - "v2.0.50727" - use .NET framework version 3.5. This is default if .NET 4.x is unavailable.
 - "v4.0.30319" - use .NET framework version 4.x (4.0, 4.5 etc, which is installed). This is default if .NET 4.x is available.
- Once a CLR is loaded in current process, it cannot be unloaded or changed. If need, restart QM or run macros in separate process.

opt_macro_function, opt_menu_toolbar, opt_autotext (QM 2.4.2) - initial run-time options for different [QM item \(19\)](#) types, as with [opt \(97\)](#).

- This function changes only options that are specified with non-zero values. To set an option to 0, assign 0x80.
- All default initial options are 0, except **opt_autotext.keymark**.

Example

```
RTOPTIONS x
x.spe_for_macros=25
```

```
x.opt_autotext.keysync=1
x.opt_macro_function.hungwindow=3
x.opt_menu_toolbar.hungwindow=3
x.opt_autotext.hungwindow=3
RtOptions 2|64 x
```

```
#QmSetWindowClassFlags $cls flags ;;flags: 0x80000000 get, 1 use dialog variables, 2 use dialog definition text,
4 disable text editing in Dialog Editor, 8 supports "WM_QM_GETDIALOGVARIABLEDATA"
```

QM 2.3.4.

Sets or gets flags for a window class. The flags are used only with QM functions, not by Windows.

cls - class name.

flags:

1	Dialog Editor adds variables for controls of this class.
2	<p>ShowDialog passes dialog definition text to controls of this class when creating them. Dialog Editor too.</p> <ul style="list-style-type: none"> By default, does it only for some known classes, eg Button, Static, Group, readonly Edit. For other controls, eg editable Edit, text can be changed only with dialog variables or setwintext. This flag is ignored for some known classes.
4	Disable text editing in Dialog Editor.
8	<p>QM 2.4.3. The control supports "WM_QM_DIALOGCONTROLDATA" message registered with RegisterWindowMessage. Use when you create a custom control that supports dialog variables, and don't want to use WM_SETTEXT and WM_GETTEXT messages to exchange control data with dialog variables.</p> <p>QM functions ShowDialog, DT_SetControl, DT_SetControls, DT_GetControl and DT_GetControls send this message to controls that support it, to set and get control data from/to dialog variables. If your control does not support it, sends WM_SETTEXT and WM_GETTEXT instead.</p> <p>"WM_QM_DIALOGCONTROLDATA" message parameters: lParam - address of dialog variable. wParam: 1 - need to set control data. The variable contains text stored in it before calling ShowDialog etc. 0 - need to get control data. Your window procedure stores control data in the variable and returns 1. The data must be a string in control-defined format.</p> <p><i>Example window procedure of your control</i></p> <pre>function# hwnd message wParam lParam sel message case WM_NCCREATE int- t_WM_QM_DIALOGCONTROLDATA=RegisterWindowMessage("WM_QM_DIALOGCONTROLDATA") ... case else if message=t_WM_QM_DIALOGCONTROLDATA str& dialogVar+=lParam sel wParam case 0 ret sub.MyGetControlData(dialogVar) case 1 ret sub.MySetControlData(dialogVar) ret DefWindowProcW(hwnd message wParam lParam)</pre>
0x80000000	Get. The function returns flags, does not set.

```
#IsValidCallback addr nBytesParam
```

QM 2.3.5.

Can be used in functions that want to validate a callback-function-address parameter.

Returns:

- 0 - invalid memory address, or the user-defined function has incorrect number of parameters.
- 1 - it is a user-defined function with correct number of parameters.

-1 - valid (readable) memory address, but it is not a user-defined function. It may be a dll function, other valid code, or not a valid code.

addr - function address.

nBytesParam - total size of parameters that the function must have.

- Function parameters are aligned at 4 bytes, and size of a single parameter usually is ≤ 4 , therefore **nBytesParam** usually must be the number of parameters multiplied by 4.
- Note that callback functions cannot have parameters of str and other composite types. Parameters of such types, as well as of other types of size > 4 , should be reference or pointer. For example, callback function can begin with:
`function str&r`. To call it: `call(addr &strVariable)`.
- Not used when the address is not of a user-defined function.

See also: [call \(132\)](#)

```
RedirectQmOutput *redirFunc
```

QM 2.4.1.

Sets callback function to intercept QM output. The callback function will be called when [out \(57\)](#) or QM sends text to QM output. Also when [out](#) or [ClearOutput](#) clears QM output.

redirFunc - address of callback function, or 0.

The callback function must begin with:

```
function# str&s reserved
```

s - text to be displayed in QM output.

- The callback function can change it.
- &s is 0 when the callback function is called to clear output.

reserved - currently not used.

The callback function can return:

- 0 - to send the text (possibly changed) to QM output. When clearing - to clear.
- 1 - to not send. When clearing - not clear.

Usage examples:

- Create an output window in exe.
- Write output text to a file etc.
- Search for certain strings in output text, and run code depending on it.
- Add number, date, etc to the output text.

The callback function is called in the same [thread \(49\)](#) that sends text. It can be any thread.

In the callback function [out](#) is not redirected.

If the output text begins with "<>", it is with [tags \(245\)](#). If you modify it, don't insert your text before the "<>". Also be careful to not break tags.

Multiple callback functions are not supported. This function replaces previous callback function. This function is used by [ExeOutputWindow](#) and [ExeConsoleRedirectQmOutput](#), therefore only one of these 3 functions can be used in an exe.

Example callback function

```
function# str&s reserved

if &s
    _OutputDebugString s
    s+=" <MODIFIED>"
else
    _OutputDebugString "<CLEAR>"

ret 1
```

```
#InitWindowsDll what ;;what: 0 winsock, 1 GDI+
```

QM 2.4.1.

Calls an API function to initialize a Windows dll that requires it. The API will be called once in process; does nothing if already called. On exit process will be called corresponding API function to uninitialized the dll.

what:

- 0 - call [WSAStartup](#) to initialize ws2_32.dll (Windows sockets). Uses version 0x202.
 - 1 - call [GdiplusStartup](#) to initialize gdiplus.dll (GDI+). Uses default GdiplusStartupInput (version 1).
-

Controls

[QM Grid control \(119\)](#)

[QM ComboBox and drop-down list controls \(118\)](#)

IStringMap interface

A string map object is an array of key-value pairs. A key is a unique string that is used to access the associated value. A value is a string that is associated with the key. Unlike a simple array, a string map is optimized to quickly find an item, even if there are 1000000 items. A string map, for example, can be used to store dictionary data in memory, where keys are words of language A, and values are words of language B. Also can be used to store a list of unique strings that are not necessary associated with values.

To create a string map object, use function [CreateStringMap](#) or [_create](#) (QM 2.3.4). To work with it, use [IStringMap](#) interface.

Example

```
IStringMap m._create
lpstr s=
    key1 value1
    key2 value2
    key3 value3
m.AddList(s)

lpstr v=m.Get("key2")
if(v) out v
else out "not found"
```

Global functions

```
dll "qm.exe" IStringMap'CreateStringMap flags ;;flags: 1 case insens., 2 exists - do nothing, 4 exists -
replace, 8 exists - add new, 16 exists - compare
```

Creates string map object and returns [IStringMap](#) COM interface pointer.

flags - same as with function [Flags](#), see below.

QM 2.3.4. You can instead use [_create](#).

IStringMap member functions

Member functions are called like: variable.Function(arguments). See the example code at the top of this topic. Note that the colored code lines below are not function calling examples. They are copied from interface declaration and used here to show function name, arguments etc.

```
[p]Flags(flags) ;;1 case insens., 2 exists - do nothing, 4 exists - replace, 8 exists - add new, 16 exists - compare
[g]#Flags()
```

Sets or gets flags to change default behavior of other functions.

Error if map is not empty.

flags:

1	Case insensitive. For example, <code>Get("a")</code> will find key "a" or "A".								
2, 4, 8, 16	What to do when Add , AddList or Rename tries to add a key that already exists. <table border="1"> <thead> <tr> <th>Default</th><th>Error.</th></tr> </thead> <tbody> <tr> <td>2</td><td>Don't add the new key.</td></tr> <tr> <td>4</td><td>Replace old value (like Set).</td></tr> <tr> <td>8</td><td>Add new item. Then the map can have duplicate keys. By default, functions Get, Get2, Set, Rename and Remove will get first added key; functions GetAll, GetAllOf and EnumNext will retrieve identical keys in FIFO order. If flag 4 also is used - last added key, LIFO order.</td></tr> </tbody> </table>	Default	Error.	2	Don't add the new key.	4	Replace old value (like Set).	8	Add new item. Then the map can have duplicate keys. By default, functions Get , Get2 , Set , Rename and Remove will get first added key; functions GetAll , GetAllOf and EnumNext will retrieve identical keys in FIFO order. If flag 4 also is used - last added key, LIFO order.
Default	Error.								
2	Don't add the new key.								
4	Replace old value (like Set).								
8	Add new item. Then the map can have duplicate keys. By default, functions Get , Get2 , Set , Rename and Remove will get first added key; functions GetAll , GetAllOf and EnumNext will retrieve identical keys in FIFO order. If flag 4 also is used - last added key, LIFO order.								

16	Compare new and old value. If equal, do nothing. If different, see other flags.
----	---

Evaluation order: 16, 8, 4, 2.

Added in QM 2.3.4.

Example

```
IStringMap m._create; m.Flags=1
```

```
Add($k $v)
```

Adds item (key-value pair) to the map. Error if the key already exists, unless flags 2-16 used with [Flags](#) or [CreateStringMap](#). **v** may be empty ("" or 0). If **v** is 0, is added "" instead.

```
AddList($s $sep)
```

Adds multiple items to the map. Error if a key from the list already exists in the map, unless flags 2-16 used with [Flags](#) or [CreateStringMap](#). In the list, each line should contain a key-value pair. To separate key and value, use characters specified in **sep**. Default **sep** is " [9]" (spaces and tabs). Note that **sep** is not a separator string; it is a set of characters that can be used. If a line is empty or begins with a separator character, does not add it. If a line contains only key, value is "". If you want to add simple list of strings (keys without values), you can use "[]" as **sep** to avoid breaking strings into keys and values.

QM 2.3.3. If **sep** is "csv", interprets **s** as [CSV \(116\)](#) string. For example, use CSV when items may be multiline. The 4-th character of **sep** sets separator. For example, "csv=" sets separator =. Default separator is comma. The CSV must contain 1 or 2 columns.

```
$Get($k)
```

Returns the value associated with the specified key. If the key does not exist, returns 0, and also sets **_hresult** to 1. Warning: the returned value is a lpstr that points to internal data, and therefore becomes invalid after you call a function that modifies the map. To avoid this, assign it to a str variable. Also, this function is not thread-safe, because another thread may call a function that modifies the map. This function is always safe if it is used just to check if the key exists ([if\(m.Get\("key"\)\)](#) ...). Note: if the key exists but the value is empty (added "" or 0), this function returns "", which evaluates to true with [if](#).

```
$Get2($k str&v)
```

Same as [Get](#), but is always safe. Stores the value into the str variable **v**. Although this function is slightly slower than [Get](#), use it if the map object can be modified by other threads.

```
Set($k $v)
```

Changes value. Error if the key does not exist.

```
Rename($k $newname)
```

Changes key name. Error if the key does not exist.

```
Remove($k)
```

Removes item (key and value). If the key does not exist, does not generate an error, but sets **_hresult** to 1.

```
RemoveAll()
```

Removes all items.

```
#Count()
```

Returns the number of items.

```
GetAll (ARRAY (str) &ak ARRAY (str) &av)
```

Retrieves all keys and/or values. Stores keys into str array **ak**, unless it is 0. Stores values into str array **av**, unless it is 0. Example:

```
ARRAY (str) ak av
m.GetAll (ak av)
int i
for (i 0 ak.len)
  out "%s %s" ak[i] av[i]
```

```
#GetAllOf ($k ARRAY (str) &av)
```

Retrieves values of all keys that match **k**. Can be useful if the map is created with flag 8, which allows to add duplicate keys.

Returns the number of matching keys. If there are no matching keys, returns 0 and sets **_hresult** to 1.

av - variable that receives values. Can be 0 if don't need.

```
GetList (str&s $sep)
```

Writes all items to a str variable. Each line will contain a key-value pair. Key and value will be separated by **sep**. Default **sep** is " ".

QM 2.3.3. If **sep** is "csv", creates [CSV \(116\)](#) string. For example, use CSV when items may be multiline. The 4-th character of **sep** sets separator. For example, "csv=" sets separator =. Default separator is comma. The CSV will contain 2 columns.

```
EnumBegin ()
```

Begins to enumerate items.

```
#EnumNext (str&k str&v)
```

Retrieves next item. Returns 1 if successful, or 0 if there are no more items. Stores key into str variable **sk**, unless it is 0. Stores value into str variable **sv**, unless it is 0. Example:

```
str sk sv
m.EnumBegin
rep
  if (!m.EnumNext (sk sv)) break
  out "%s %s" sk sv
```

```
EnumEnd ()
```

Should be called (although not necessary) if the enumeration loop exits before all items are enumerated.

```
IntAdd ($k v)
```

Adds item. Same as [Add](#), but **v** is integer number. Internally it is stored as string. To add a numeric value (including double and other types), also can be used [Add](#), but you have to convert it to string (assign it to a str variable and pass the variable to [Add](#)). This function just simplifies that.

```
#IntGet ($k int&v)
```

Gets value as integer number. If key exists, stores value into **v** and returns 1, else returns 0. To retrieve a numeric value also

can be used other functions ([Get](#), [GetAll](#), etc). These functions return the number as string, and you can use [val](#) to convert it to number. This function just simplifies that.

```
IntSet($k v)
```

Changes value. Same as [Set](#), but **v** is integer number.

Notes

To retrieve all items, can be used [GetAll](#), [GetList](#) or EnumX functions. [GetAll](#) is slower and requires much memory to store all data, especially if the map is large. However, EnumX functions should not be called simultaneously in multiple threads. Items always are retrieved sorted.

All [IStringMap](#) functions are thread-safe, except in cases mentioned above. It means that a map object can be used by multiple threads simultaneously.

If a function generates an error, its description always is "The parameter is incorrect". The most common error is when you try to add an item that already exists. To avoid it, you can use [Get](#) to check if it exists, or use [err](#) (faster), or use flags 2-16 with [Flags](#) or [CreateStringMap](#). Examples:

```
if(!m.Get("key2"))
    m.Add("key2" "new value")
else
    _out "already exists"

m.Add("key2" "new value")
err
_out "already exists"
```

ICsv interface

ICsv interface is used to work with **CSV** files and CSV-formatted strings. Parses CSV, creates/stores/manages 2-dim table in memory, composes CSV from the table.

CSV is a simple text format used to store tables. Rows are separated by new lines, cells by commas. Values containing new lines, commas or double quotes are enclosed in double quotes. Double quotes are doubled. Spaces around commas and new lines are ignored. Example:

```
value1,value2, value3
11,22,33
"value, with, commas","value with ""quotes""", "multiline
value"
'',
,rows with empty values,
'',
```

The CSV file format is supported by many programs, including Microsoft Excel, and therefore can be used to exchange data between them. Another popular file format [XML \(117\)](#) is more powerful but adds much overhead (slower parsing/composing, requires more space in file and in memory). To store tables, CSV format usually is better and easier to use.

You can find more information about CSV on the Internet.

CSV files also can be manipulated using **Database** class, but it is slower.

Added in QM 2.3.0.

Declare an **ICsv** variable and call `_create`. Then use **ICsv** interface functions.

Example 1

```
a CSV string for testing
str csv=
  A1, B1, C1
  A2, B2, C2

create ICsv variable and load data
ICsv x._create
x.FromString(csv)

get a cell
str cell
cell=x.Cell(1 1)
out F"cell 1 1 was: {cell}"

change cell
cell="Changed"
x.Cell(1 1)=cell

get CSV data to string
x.ToString(csv)

results
out "changed CSV:"
out csv
```

Example 2

```
create ICsv variable, change separator and load data from file
str s
ICsv v._create
v.Separator=";"
v.FromFile("$my qm$\test.csv")

enumerate rows and columns
```

```

int nr=v.RowCount
int nc=v.ColumnCount
int r c
for r 0 nr ;;for each row
  out "---- row %i ----" r
  for c 0 nc ;;for each column
    s=v.Cell(r c)
    out s

v.ToString(s); out s ;;show CSV

save to file
v.ToFile("$my qm$\test.csv")

```

Examples of adding rows

Instead of `_create` can be used global function `CreateCsv`. Before QM 2.3.4, `_create` could not be used.

```
ICsv'CreateCsv [flags] ;;flags: 1 separator is comma
```

Creates object and returns **ICsv** COM interface pointer.

flags (QM 2.3.2):

- | | |
|---|---|
| 1 | Use comma as separator. Same as <code>var.Separator=", "</code> . <ul style="list-style-type: none"> If this flag not used, default separator is as specified in Control Panel. Note that this behavior is different than with <code>_create</code>, where default separator is comma. |
|---|---|

Example: `ICsv x=CreateCsv(1)`

ICsv member functions

Member functions are called like: `variable.Function(arguments)`. See the example code at the top of this topic. Note that the colored code lines below are not function calling examples. They are copied from interface declaration and used here to show function name, arguments etc.

```
[p] Separator($sep)
[g] $Separator()
```

Sets or gets (QM 2.3.2) separator that is used when parsing and composing CSV. Example: `v.Separator="; "`.

If **sep** is "", uses separator specified in Control Panel -> Regional and Language Options -> Customize... -> List Separator.

Info: Excel, when opening and saving CSV files, also uses the separator that is set in Control Panel.

Initially the variable uses comma if it was created with `_create`. Uses "" if created with `CreateCsv` without flag 1.

```
FromString($s)
```

Parses CSV string and creates table in memory. The **ICsv** variable manages the table.

Error if the CSV string contains errors. Also, must match separator (see `Separator`, above).

```
ToString(str&so)
```

Composes CSV string from the memory table and store an a **str** variable.

```
FromFile($file)
```

Parses CSV file and creates table in memory.

Error if the CSV file contains errors or uses different separator.

Supports [macro resources \(261\)](#) (QM 2.4.1) and [exe \(51\)](#) resources.

```
ToFile($file [flags]) ;;flags: 1 append, 0x100 safe, 0x200 safe+backup
```

Saves to file in CSV format.

flags:

1	Append.
0x100	QM 2.4.0. Safe/atomic saving. The file will never be corrupted on power failure etc. Writes to a temporary file, flushes its buffers, and renames the temporary file to file , replacing if exists.
0x200	QM 2.4.0. Safe saving and backup. Same as 0x100, but also creates a backup file, named file-backup .

QM 2.3.5. Creates parent folder if does not exist.

```
FromArray(ARRAY(str) &a)
```

Creates the memory table from 2-dim array.

Added in QM 2.3.4.

Note: in 2-dim arrays, the first dimension is for columns, the second for rows.

```
ToArray(ARRAY(str) &a)
```

Creates 2-dim array from the memory table.

Added in QM 2.3.4.

Note: in 2-dim arrays, the first dimension is for columns, the second for rows.

```
FromQmGrid(hwnd [flags]) ;;flags: 1 no first column, 2 no empty rows, 4 selected
```

Gets cells from [QM_Grid \(119\)](#) control and creates table in memory.

hwnd - control handle.

flags:

1	don't get first column.
2	don't get empty rows.
4 (QM 2.3.2)	get only selected or checked (depending on style) rows.
8 (QM 2.3.4)	remove <...> in first column that was used to set row type.

Does not change [Separator](#) and [RowDataSize](#).

```
ToQmGrid(hwnd [flags]) ;;flags: 1 only first column, 2 no first column
```

Populates QM_Grid control.

hwnd - control handle.

flags:

1	only first column.
2	except first column.

If flag 2 used, does not clear the control. The control must already contain cells in the first column. Cells in other columns can be empty or not.

If flag 1 used, the CSV table should contain 1 column. The grid can contain 1 or more columns.

If flag 2 used, the CSV table should contain control's column count -1.

Else the CSV table should contain control's column count.

`Clear()`

Deletes all rows.

Sets `ColumnCount` to 0.

Does not change `Separator` and `RowDataSize`.

```
[g] #RowCount ()
[g] #ColumnCount ()
[p] ColumnCount (count)
```

Gets the number of rows or columns.

QM 2.3.2. `ColumnCount` also can be used to set column count.

```
[g] $Cell (row col)
[p] Cell (row col $value)
```

Gets or sets cell value. See the example at the beginning.

row - 0-based row index.
col - 0-based column index.
value - cell text.

The returned string of the "get" function is temporary. It becomes invalid after calling a function that modifies the table. To use it later, assign it to a `str` variable, like in the example.

QM 2.4.3: Can set cell even if cell specified by **row** and **col** does not exist. If **col** is > column count, adds more columns. If **row** is = row count or < 0, adds new row. Would be error in older QM.

```
[g] #CellInt (row col)
[p] CellInt (row col value)
[p] CellHex (row col value)
```

Gets or sets cell value. The same as `Cell` (see above), but converts string to/from number.

row - 0-based row index.
col - 0-based column index.
value - an integer value.

The "get" function gets cell text and converts to `int`. Cell text can be a number in decimal or hexadecimal format. Can contain operator |. For example, returns 7 if cell text is "1|2|0x4". If cell text is empty or does not begin with a number, sets `_hresult (144)`=1 and returns 0.

There are two "put" functions:

- [CellInt](#) puts **value** converted to decimal format string, like "10".
- [CellHex](#) puts **value** converted to hexadecimal format string, like "0xA".

Added in QM 2.4.3.

Examples

```
ICsv x._create
x.FromString("one,1[]two,2")
int i=5

x.CellInt(0 1)=i ;;the same as x.Cell(0 1)=F"{i}"
out x.Cell(0 1) ;; "5"

x.CellHex(0 1)=i ;;the same as x.Cell(0 1)=F"0x{i}"
out x.Cell(0 1) ;; "0x5"

i=x.CellInt(0 1) ;;the same as i=val(x.Cell(0 1))
if(_hresult) out "not a number"; else out i
```

```
RemoveRow(row)
```

Removes row.

row - 0-based row index.

```
#AddRowMS(row [nCells] [$cells] [firstCell])
#AddRowLA(row [nCells] [lpstr*cells] [firstCell])
#AddRowSA(row [nCells] [str*cells] [firstCell])
```

Adds or inserts new row.

See also (below): [AddRow1](#), [AddRow2](#), [AddRow3](#), [AddRowCSV](#).

row - 0-based row index where to insert the new row. Use an invalid index (e.g. -1) to add to the end.

nCells - number of cells to add.

cells - cell values.

firstCell (QM 2.3.2) - 0-based column index where to insert the cells.

Each of these functions differs only by the format of the values array (**cells**).

Function	cells type	Comments
AddRowMS	lpstr	cells must be in multistring format, ie multiple null-terminated strings following each other, like "string1[0]string2[0]string3[0]". See also GetRowMS , below.
AddRowLA	lpstr*	cells must be address of the first variable in an array of lpstr variables.
AddRowSA	str*	cells must be address of the first variable in an array of str variables.

If the number of columns ([ColumnCount](#)) is still not set (is 0), sets it to **nCells+firstCell**. You can also set it with [ColumnCount](#) before adding rows.

If **cells** or **nCells** is omitted or 0, adds 1 empty row. You can use [Cell](#) to set cell values.

Cells that are outside the range specified by **firstCell** and **nCells** will be "". If some strings in **cells** are null, the cells will be "". If [RowDataSize](#) is nonzero, the functions fill row data with 0.

QM 2.4.3: If **nCells+firstCell** is > [ColumnCount](#), adds more columns. Also if **nCells** is 0 and [ColumnCount](#) is 0. In both cases, in older QM would be error (need to set column count with [ColumnCount](#)).

Examples

```

out
ICsv x._create
x.ColumnCount=3

-----

add empty row and use Cell
int r=x.AddRowMS(-1)
x.Cell(r 0)="c1"; x.Cell(r 1)="c2"; x.Cell(r 2)="c3"

MS
lpstr multistring="ms1[0]ms2[0]ms3"
x.AddRowMS(-1 3 multistring)

LA with array
str st="a1 a2 a3"
ARRAY(lpstr) a
int nt=tok(st a 3 "" 1)
x.AddRowLA(-1 nt &a[0])

SA with array
ARRAY(str) as="as1[]as2[]as3"
x.AddRowSA(-1 as.len &as[0])

LA with local variables
lpstr s1("s1") s2("s1") s3("s1")
x.AddRowLA(-1 3 &s1)

SA with local variables
str ss1("ss1") ss2("ss1") ss3("ss1")
x.AddRowSA(-1 3 &ss1)

-----

str s
x.ToString(s)
out s

```

```

#AddRow1(row $s1)
#AddRow2(row $s1 [$s2])
#AddRow3(row $s1 [$s2] [$s3])

```

Adds or inserts row with 1, 2 or 3 cells. Similar to the above functions. Use when it is more convenient to pass 1, 2 or 3 values than to use an array.

row - 0-based row index where to insert the new row. Use an invalid index (e.g. -1) to add to the end.
s1, s2, s3 - cell text.

[AddRow2](#) added in QM 2.3.4, other functions in QM 2.4.3.

```

#AddRowCSV(row $s1)
#ReplaceRowCSV(row $s1 [$s2])

```

[AddRowCSV](#) adds or inserts one or more rows, using a CSV string to specify the cells. Returns first new row index.
[ReplaceRowCSV](#) replaces one or more rows, using a CSV string to specify the cells. Returns first replaced row index.

row - 0-based row index where to insert the new row. Use an invalid index (e.g. -1) to add to the end.
csv - CSV string containing one or more rows and any number of columns. Don't forget to "enclose" cells containing commas etc.

Added in QM 2.4.3.

Example

```
ICsv x._create
x.FromString("A,B[]C,D")
out x.AddRowCSV(-1 "E,F")
x.ToString(_s); out _s
```

```
#ReplaceRowMS(row [nCells] [$cells] [firstCell])
#ReplaceRowLA(row [nCells] [lpstr*cells] [firstCell])
#ReplaceRowSA(row [nCells] [str*cells] [firstCell])
```

Replaces row.

See also (above): [Cell](#), [ReplaceRowCSV](#).

Replaces **nCells** cells of row **row**, starting from **firstCell**. Does not change other cells. If **row** does not exist, adds new row to the end like the AddRowX functions.

Everything else is as with the AddRowMS/LA/SA functions.

Added in QM 2.3.2.

```
GetRowMS(row str*cells)
```

Stores all row cells into a **str** variable in multistring format.

Added in QM 2.3.2.

The variable then can be used to add/insert/replace a row of this or another **ICsv** variable. Use function [AddRowMS](#) or [ReplaceRowMS](#) (see above).

```
MoveRow(row to)
```

Moves row.

row - index of row to move.
to - new index of the row.

Added in QM 2.3.2.

```
InsertColumn(col)
RemoveColumn(col)
```

[InsertColumn](#) - inserts 1 empty column. If **col** is invalid (e.g. -1), adds to the end.

[RemoveColumn](#) - removes 1 column.

Alternatively you can use [ColumnCount](#) to add or remove columns.

Added in QM 2.3.2.

```
[p]RowDataSize(nBytes)
[g]#RowDataSize()
[g]!*RowData(row)
```

[RowDataSize](#) - sets or gets size of extra memory to allocate for each row.

[RowData](#) - returns address of extra memory of a row.

Allows to place any data in each row. It simplifies using **ICsv** as base of new table-based classes. The data exists only in memory.

Added in QM 2.3.2.

```
Sort(flags [col]) ;;flags: 0 simple, 1 insens, 2 ling, 3 ling/insens, 4 number/ling/insens, 128 date, 0x100 descending
```

Sorts rows using text of one of columns.

flags:

0	Simple, case sensitive. Uses StrCompare (114) to compare strings.
1	Simple, case insensitive. Uses StrCompare to compare strings.
2	Linguistic, case sensitive. Uses StrCmp to compare strings.
3	Linguistic, case insensitive. Uses StrCmpl to compare strings.
4	Number, linguistic, case insensitive. Uses StrCmpLogicalW to compare strings. It compares numbers in strings as number values, not as strings.
128	QM 2.3.3. Date.
0x100	Sort descending.

col - 0-based column index.

Added in QM 2.3.2.

```
#Find($s [flags] [col] [startRow]) ;;flags: 1 insens, 2 wildcard, 4 beginning, 8 end, 16 middle, 32 rx
```

Finds row. Returns 0-based row index, or -1 if not found.

s - cell text.

flags - how to compare strings.

col - 0-based column index.

startRow (QM 2.4.3) - 0-based row index from where to start searching.

Added in QM 2.3.4.

Notes

ICsv functions are not thread-safe. Don't use a single variable in multiple threads simultaneously. It can damage data. If need to use in multiple threads, use [lock \(112\)](#).

IXml and IXmlNode interfaces

IXml and **IXmlNode** interfaces are used to work with **XML**.

XML is a text file format used to store data or settings. It can be used to store tables as well as hierarchical data structures (trees) of any format. To save tables, **CSV (116)** or **SQLite** often is better and easier to use. You can find many information about XML on the Internet.

XML format is similar to HTML. Example:

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

You can find many examples on your computer, since many programs save their settings and data in XML format. Press Win+F and search for *.xml files.

Alternatively can be used MSXML. It has more features. It is a COM component, included in Windows. You can find an example in the QM forum. However different Windows versions have different MSXML versions. Also it is quite slow. **IXml** works on all Windows versions, is faster, uses less memory and does not load any dlls. **IXml** supports basic XML features. However it does not help you to work with advanced XML features, such as namespaces, DTD, schemas, XPath (partially supported), custom entities, etc. In most cases it successfully parses and composes documents that use these features, but does not help you to manage the advanced features.

Added in QM 2.3.0.

To create an XML object, use function **CreateXml** or **_create** (QM 2.3.4). To work with it, use **IXml** and **IXmlNode** interfaces. Examples:

This macro loads XML file and adds several new nodes.

```
out
IXml x._create

x.FromFile("$qm$\test.xml") ;;load XML file
err out "Error: %s" x.XmlParsingError; ret ;;error if the file is corrupted

IXmlNode my=x.RootElement.Add("myelement") ;;add one new element as child of the root element
my.Add("mysubelement" "text of my subelement") ;;add its child element
my.Add("mysubelement2" "some text").SetAttribute("a" "my attribute") ;;add another child element
and an attribute

str s
x.ToString(s) ;;compose xml string
out s
```

This macro creates new XML document, adds elements, attributess, finds and gets values.

```
out
IXml x._create

x.Add("?xml") ;;add xml declaration (optional)
IXmlNode re=x.Add("rootelem") ;;add root element (XML must have exactly 1 root element)
re.Add("child" "text").SetAttribute("a" "10") ;;add child element with text and 1 attribute

IXmlNode e=re.Add("elem2") ;;add another child element
e.Add("cc" "text of cc") ;;add child of child
e=e.Add("cc2") ;;add another child of child
e.SetAttribute("a" "AAA") ;;add attribute
e.SetAttribute("b" "BBB") ;;add another attribute

str v1=re.ChildValue("child") ;;get value of a child (same as re.Child("child").Value)
e=x.Path("rootelem/elem2/cc2/@b") ;;find a node by path
str v2=e.Value ;;get its value
out v1
out v2

out "-----"
str s
```

```
x.ToString(s) ;;compose xml string
out s

/
function IXml&xml [withAttr]

This function displays all XML nodes and their properties.
To test it, create new function, name it XmlOut and paste this code.

EXAMPLE
IXml x=CreateXml
x.FromFile("$my qm$test.xml")
XmlOut x 1

lpstr st="root[]el[]a[]text[]xml[]DOC[]PI[]CD[]comm"
ARRAY(str) at=st

ARRAY(IXmlNode) a; int i
xml.Root.GetAll(withAttr!=0 a)

for(i 0 a.len)
  XMLNODE xi; a[i].Properties(&xi)
  out "%-15s %-4s F=0x%X L=%i V='%s'", xi.name, at[xi.xtype], xi.flags, xi.level, xi.value
```

XML nodes

Example XML, containing all node types:

```
<?xml version="1.0" encoding="utf-8" ?>
<?xml-stylesheet type="text/xsl" href="example.xsl"?>
<!DOCTYPE example >
<!--this is an example xml-->
<example>
  <simple>
    <elem>text</elem>
    text before<a>left<b>center</b>right</a>after
    <elem>text again</elem>
  </simple>
  <entities>&lt;&gt;&amp;&quot;&apos;</entities>
  <unicode>&#2345; &#x5678; &#x10FFFF</unicode>
  <attr-and-text a="aaa" b="bbb" c="ccc">text</attr-and-text>
  <empty />
  text after empty
  <notext></notext>
  <empty-attr a="1" />
  <notext-attr b=""></notext-attr>
  <lang name="qm">
    <styles>
      <s32 f="Courier' &quot;New" fs="8">Default</s32>
      <s1 u="1">Tabs</s1>
      <?pi abc?>
    </styles>
    <misc>
      <multiline>
line1
line2
      </multiline>
      <mixed-content>
line1
      <?pi intext?>
line2
      </mixed-content>
    </misc>
  </lang>
  <![CDATA[can contain < & etc]]>
  <!--<rem>text</rem>-->
</example>
```

XML node types and properties used by IXml and IXmlNode:

Node type	Constant	Examples	Name	Value	Can have children	Can have attributes	Can be at the root	Can be not at the

117. IXml and IXmlNode interfaces

								root
virtual root	XT_Root 0	Not used in XML.			Yes			
element	XT_Element 1	<code><elem>text</elem></code> <code><example></example></code> <code><empty /></code> <code><empty-attr a="1" /></code> <code><s1 u="1">Tabs</s1></code>	name	Text between <code><elem></code> and <code></elem></code> . Empty if <code><elem /></code> or <code><elem></elem></code> or has child nodes.	Yes	Yes	Yes, 1	Yes
attribute	XT_Attribute 2	<code>a="1"</code> <code>encoding="utf-8"</code>	name	Attribute value. It is text within " or '.				
text	XT_Text 3	text line1 line2		The text, including all spaces and leading/trailing new lines. Note that this element type can be found only where the parent element also has other child nodes. If not, text is interpreted as value of the parent element, not as a separate node.				Yes
xml declaration	XT_XmlDeclaration 4	<code><?xml version="1.0" encoding="utf-8" ?></code>	?xml			Yes	Yes, 1, first	
DOCTYPE (document type)	XT_DocumentType 5	<code><!DOCTYPE example ></code>	!DOCTYPE	Text between <code><!DOCTYPE</code> and <code>></code> , not including leading and trailing spaces. QM does not parse it.			Yes, 1, before root element	
processing instruction	XT_ProcessingInstruction 6	<code><?xml-stylesheet type="text/xsl" href="example.xsl"?></code>	?name	Text between <code><?</code> instructionname and <code>?></code> , not including leading and trailing spaces. QM does not parse it.			Yes	Yes
CDATA (custom data)	XT_CDATA 7	<code><![CDATA[can contain < & etc]]></code>	![Text between <code><![CDATA[</code> and <code>]]></code> , including all spaces.				Yes
comment	XT_Comment 8	<code><!--this is an example xml--></code> <code><!--<rem>text</rem>--></code>	!-	Text between <code><!--</code> and <code>--></code> , including all spaces.			Yes	Yes

Note that XML element types 'end element' and 'white space' are not used. It is managed automatically.

Interfaces

There are two COM interfaces.

Interface **IXml** represents an XML document. It is responsible for loading, saving, parsing and composing XML. It manages XML nodes and has functions to access them.

Interface **IXmlNode** represents a node in the XML document. It has functions to get and set node values and other properties, find other nodes, add child nodes.

Global functions

```
IXml 'CreateXml [flags] ;;flags: 1 normalize newlines, 2 enable UserData, 4 ignore encoding, 8 auto save, 0x100
safe save, 0x200 safe save+backup
```

Creates XML object and returns **IXml** COM interface pointer.

flags - same as with function **Flags**, see below.

QM 2.3.4. You can instead use **_create**.

IXml member functions

Member functions are called like: variable.Function(arguments). See the example code at the top of this topic. Note that the colored code lines below are not function calling examples. They are copied from interface declaration and used here to show function name, arguments etc.

```
[p] Flags(flags) ;;1 normalize newlines, 2 enable UserData, 4 ignore encoding, 8 auto save, 0x100 safe save, 0x200
safe save+backup
[g] #Flags()
```

Sets or gets flags to change default behavior of other functions.

flags:

1	When parsing, replace various new line forms ([, [13]) to [10] (XML standard). When composing, make all new lines []. If this flag is not set, does not touch new lines.
2	Enable IXmlNode.UserData, which is disabled by default.
4	When parsing XML, ignore "encoding" attribute in xml declaration. Read more below, in Notes chapter.
8	Autosave. After successful FromFile , before destroying the IXml object automatically saves to the same file. However will not save if you after call FromString .
0x100	QM 2.4.0. Safe/atomic saving. This flag is added to ToFile flags.
0x200	QM 2.4.0. Safe saving and backup. This flag is added to ToFile flags.

Added in QM 2.3.4.

```
[g] IXmlNode 'Root()
```

Gets the virtual root node. It actually does not exist in XML, but can be used as parent of XML root nodes (xml declaration, DOCTYPE, root element, etc).

When you read "get node", "get element" or "get attribute", it means "get **IXmlNode** interface pointer that can be used to manipulate the node".

```
[g] IXmlNode 'RootElement()
```

Gets the root element. For example, in XML `<?xml version="1.0"?><elem><elem2>text</elem2></elem>`, the root element is elem.

```
IXmlNode 'Path($path [ARRAY (IXmlNode) &allMatching])
```

Gets any node by path. The **IXmlNode** interface also has these functions, [look there](#).

```
IXmlNode 'Add($name [$value])
```

Adds a node at the root. The **IXmlNode** interface also has this function, and it is documented there. Note that xml declaration and DOCTYPE nodes are always added where they should be, even if more nodes already exist at the root.

```
Delete (IXmlNode&node)
```

Delete a node. The argument must be a variable of **IXmlNode** type. If it is an element, also deletes its attributes, text and descendant nodes. Can be used to delete an attribute too.

```
Clear()
```

Delete all nodes.

```
[g] $XmlParsingError
```

If [FromString](#) or [FromFile](#) failed to parse XML, gets the place (substring) in XML that has error.

```
[g] #Count()
```

Gets the number of nodes in XML, including attributes.

```
IXmlNode'FromString($s)
```

Parses an XML string and creates a tree of node objects in memory. Returns root element.

```
ToString(str&so)
```

Composes XML string from the tree.

```
IXmlNode'FromFile($file [$defaultXML])
```

Parses an XML file and creates a tree of node objects in memory. Returns root element.

Error if the file does not exist, unless you provide **defaultXML**.

defaultXML - if the file does not exist, initializes the object (like [FromString](#)). The XML must contain at least the root element. Example: "<r/>".

After successful [FromFile](#), the object remembers **file**, and can use it to autosave (see [CreateXml](#) flag 8) and with [ToFile](#). The object loses this memory after [FromString](#) or unsuccessful [FromFile](#).

Supports [macro resources \(261\)](#) (QM 2.4.1) and [exe \(51\)](#) resources.

```
ToFile([$file] [flags]) ;;flags: 0x100 safe, 0x200 safe+backup
```

Saves to an XML file.

flags (QM 2.4.0):

0x100	QM 2.4.0. Safe/atomic saving. The file will never be corrupted on power failure etc. Writes to a temporary file, flushes its buffers, and renames the temporary file to file , replacing if exists.
0x200	QM 2.4.0. Safe saving and backup. Same as 0x100, but also creates a backup file, named file-backup .

If **file** is omitted or empty, uses the same file as used with [FromFile](#).

Flags 0x100 and 0x200 also can be specified earlier with [Flags](#) or [CreateXml](#). It is useful when you use the auto-save feature.

QM 2.3.5. Creates parent folder if does not exist.

IXmlNode member functions

```
[g] IUnknown'XmlDoc()
```

Gets parent [IXml](#).

```
[g] IXmlNode'Parent()
[g] IXmlNode'Prev()
[g] IXmlNode'Next()
[g] IXmlNode'FirstChild()
[g] IXmlNode'LastChild()
```

Gets parent, sibling or child node. The first three functions can be used with attribute nodes too.

```
[g] IXmlNode'Child($name [index])
```

117. IXml and IXmlNode interfaces

Gets a child node by name and/or index. Tip: if you need its value, you can instead use [ChildValue](#).

name - child node name.

- If "", matches any name.
- Can have a filter expression, like with [Path](#).
- QM 2.3.5. Supports [wildcard characters \(196\)](#) in element name.

index - 1-based match index. If name is "", it is index in all children, else - in children whose name is **name**. If omitted or 0 or 1, gets first matching child.

```
[g]IXmlNode.Attribute($name)
```

Gets an attribute node. If **name** is "", gets first attribute. If **name** is "", gets last attribute. Tip: if you need its value, you can instead use [AttributeValue](#).

```
IXmlNode.Path($path [ARRAY (IXmlNode) &allMatching] [flags])
```

Gets any node by path.

The [IXml](#) interface also has this function.

path - node path, like "rootElement/itsChild/theNode".

- It is not XPath, but supports something from it.
- [IXml.Path](#) starts searching from the root of the XML tree.
- [IXmlNode.Path](#) starts searching from (not including) the element of the [IXmlNode](#) variable, unless path begins with "/".

allMatching - array variable that receives [IXmlNode](#) objects of all matching nodes. Optional, can be 0.

flags (QM 2.3.5):

1	Search in all matching paths (like XPath). For example if path is "elem1/elem2", searches for "elem2" in all "elem1". Without this flag searches only in the first matching path (in the example - in the first found "elem1").
2	If last part of path is "", get only elements (like XPath). Without this flag gets nodes of any type except attributes.

Examples of supported paths:

"node"	Get child node "node", like Child does. The advantage is that here you can use allMatching .
"elem/node"	Get child node "node" of element "elem".
"elem/@a"	Get attribute "a" of element "elem".
"**"	Get first child node, like FirstChild does. If allMatching is used, it receives all child nodes.
"elem/**"	Get first child node of element "elem".
"elem/@**"	Get first attribute of element "elem".
"*/node"	Get child node "node" of the first or any (flag 1) child element.
"../node"	Get child node "node" of the parent element.
"/*/*node"	QM 2.3.5. Get child node "node" of the root element.
"//node"	QM 2.3.5. Get any descendant node "node". If "/*/*node", always searches from the root.

An element name can be followed by a filter expression. Examples:

"elem[='abc']"	Get element "elem" whose value is "abc".
"*[='abc']"	Get first node whose value is "abc".
"elem[@id='abc']"	Get element "elem" that has attribute "id" whose value is "abc".
"elem[@id*='ab*']"	Get element "elem" that has attribute "id" whose value begins with "ab".
"elem[node='abc']"	Get element "elem" that has a child "node" whose value is "abc".
"elem[node*='ab*']"	Get element "elem" that has a child "node" whose value begins with "ab".
"elem[.='abc']"	QM 2.3.5. Same as "elem[='abc']".
"elem[@id>0]"	QM 2.3.5. Get element "elem" if numeric value of its attribute "id" is > 0.
"elem[node]"	QM 2.3.5. Get element "elem" if it has child node "node".
"elem[@id]"	QM 2.3.5. Get element "elem" if it has attribute "id".

Filter expression operators:

=	If the right part is enclosed in ', compare as strings, case insensitive. Else compare as integer numbers (QM 2.3.5, in older QM use #=).
=	Compare as strings with wildcard characters (196) , case insensitive.
!	Logical not. Can be used before other operators. For example, != means not equal.
>, <, >=, <=	QM 2.3.5. Compares as integers.

<=, &

Filter expressions can be anywhere in path. Example: "elem1[@id='abc']/elem2".
Filter expressions can be used with all node types except attributes.

New in QM 2.3.5:

- Added **flags**.
- Supports [wildcard characters \(196\)](#) in element names, not just single "'".
- A "." part in path is valid. It explicitly specifies current node.
- If path begins with "/", always searches from the root.
- Supports "/" and "/" to find any descendant node without specifying path to it.
- If path ends with "@*", **allMatching** receives all attributes, not just the first.
- Path containing "@attribute/.." is valid and gets parent element of the attribute.
- In filter expression supports more operators, etc. See above.

Tip: To create path where some parts of it are variables, use [operator F \(138\)](#) or [str.format \(213\)](#).

See also: [GetAll](#)

```
[g]$Name()
```

Gets name of the node. Can be used with attributes too.

```
[g]$Value()
[p]Value($value)
```

Gets or set value of the node. Can be used with attributes too.

When used to set value, error if CDATA value contains]]> or comments value contains --. Values of elements and attributes can contain any characters because special characters are replaced with XML escape sequences.

```
ValueBinaryGet(str&value)
ValueBinarySet(str&value [flags]) ;;flags: 1 compress, 2 hex
```

ValueBinarySet - sets value of the node.

value - a str variable. It can contain binary data. In XML file the data will be converted to text, because binary data cannot be used.

flags:

1	compress.
2	use Hex encoding (fast encoding/decoding, 100% bigger encoded string). If not set, uses Base64 encoding (fast encoding, slow decoding, 33% bigger encoded string).

ValueBinaryGet - gets value of the node. It must be set by **ValueBinarySet**.

```
[g]#Type()
```

Gets type of the node. It is a numeric value listed in the Nodes chapter above. Can be used with attributes too.

```
[g]#UserData()
[p]UserData(userdata)
```

Attaches a numeric value to the node, or gets the attached value. You can use the value for any purpose. It is used only in memory, and is not saved or included in XML string. By default this function is disabled. To enable it, use flag 2 with **CreateXml**. Can be used with attributes too.

```
Properties(XMLNODE&xi)
```

Gets all node properties using single function call. The argument must be a variable of type **XMLNODE**. Can be used with attributes too.

```
type XMLNODE $name $value @level !xtype !flags userdata
```

name, value, xtype, userdata - described above.

level - level of the node in the XML hierarchy.

- Root nodes have level 0, direct child nodes of the root element have level 1, and so on.

flags:

1	the element has text.
2	the element has children. • Note that an element cannot have both text and children.
4	the element or xml declaration has attributes.
128	the attribute is in xml declaration.

```
[g]$ChildValue($name [index])
```

Gets value of a child node. Can be used instead of `n.Child("name").Value`.

Parameters are same as with [Child](#).

```
[g]$AttributeValue($name)
[g]#AttributeValueInt($name)
```

[AttributeValue](#) - gets value of an attribute. Can be used instead of `n.Attribute("name").Value`.

[AttributeValueInt](#) - the same, but converts the value to integer number.

```
IXmlNode'Add($name [$value])
```

Adds a child node and optionally sets its value. Adds to the end of the list of children. Does not replace existing items with the same name (use [SetChild](#) instead). Not used to add attributes (use [SetAttribute](#) instead).

```
IXmlNode'Insert(IXmlNode'iafter $name [$value])
```

The same as [Add](#), but adds after **iafter**, which must be a child node (variable of [IXmlNode](#) type) of the element. If **iafter** is 0, adds to the beginning.

```
IXmlNode'SetChild($name $value)
```

Adds or replaces a child node. If the child node exists, sets its value like [Value](#) does, else adds new node like [Add](#) does.

```
IXmlNode'SetAttribute($name $value)
IXmlNode'SetAttributeInt($name value)
```

[SetAttribute](#) - adds or replaces an attribute. Ensures that the element will not have duplicate attributes.

[SetAttributeInt](#) - the same, but **value** must be an integer number.

```
Move(IXmlNode'parent IXmlNode'iafter)
```

Moves an element to another place. Can be used only for elements.

parent - new parent element. If 0, moves within the same parent.

iafter - node after which to insert. If 0, inserts at the beginning.

```
GetAll(flags ARRAY(IXmlNode) &a)
```

Gets all descendants (direct children, their children, and so on).

flags:

1	include attributes.
2	get only direct children.

To get all nodes in whole XML, call this function for the virtual root node. See example at the beginning of this topic.

To get specific nodes, use [Path](#) instead. Example: `x.Path("./name" a)`.

```
[g]#ChildCount()
```

Gets number of direct child nodes.

To get number of specific child nodes, instead use [Path](#) with array. Then the number will be equal to the array length.

```
DeleteChild($name)
```

Deletes all child nodes that match **name**.

name - child node name.

- If "", deletes all child nodes.
- Can have a filter expression, like with [Path](#).
- Supports [wildcard characters \(196\)](#) in element name.

Not error if there are no matching nodes. Then sets variable [_hresult \(144\)](#) to 1.

Added in QM 2.4.0. In older QM versions would need to find child nodes, for example with [Child](#) or [Path](#), and call [IXml.Delete](#) for each.

```
DeleteAttribute($name)
```

Deletes an attribute.

Not error if the attribute does not exist. Then sets variable [_hresult \(144\)](#) to 1.

Added in QM 2.4.0. In older QM versions would need to call [Attribute](#) and [IXml.Delete](#).

Notes

XML encoding

[IXml](#) can load ANSI and Unicode XML files. Unicode can be in UTF-8 or UTF-16 format.

When loading UTF-16 XML file, QM converts the loaded XML to UTF-8.

In [Unicode \(267\)](#) mode, when loading ANSI XML file where 'encoding' attribute in xml declaration is ISO-8859-x or Windows-125x, QM converts the XML to UTF-8, unless flag 4 is used with [CreateXml](#) or the encoding is not supported by the OS.

In both cases, if you then call [ToFile](#) or [ToString](#), it saves/gets XML in UTF-8 format.

UTF-8 is used everywhere in QM when it is running in Unicode mode. It is also the default encoding used in XML (used if the XML is not UTF-16 and does not contain a non UTF-8 'encoding' attribute in xml declaration).

In ANSI mode QM ignores the 'encoding' attribute. It does not convert UTF-8 XML to ANSI.

If you are creating XML while QM is running in ANSI mode, and the XML may contain non [ASCII \(239\)](#) characters, and the XML file will be used not only in QM on your computer, you should add xml declaration with 'encoding' attribute. In Unicode mode it is not necessary because QM text encoding matches XML default text encoding (UTF-8).

XML validation and errors

When loading XML file or string, and when adding nodes and setting values, [IXml](#) validates it quite strictly. If the file is not well formed, or a name/value is invalid, it generates error. In some cases it silently makes corrections.

Functions that are used to get nodes don't generate error if the node does not exist. They set variable [_hresult \(144\)](#) to 1 and return 0.

Other notes

These functions are not thread-safe. Don't use a single variable in multiple threads simultaneously. It can damage data. If need to use in multiple threads, use [lock \(112\)](#).

[IXmlNode](#) does not use reference counting. Don't use variables of this type after the parent [IXml](#) variable is destroyed or cleared or the node is deleted. Declare [IXmlNode](#) variables after (not before) the [IXml](#) variable. Otherwise may be generated exception or damaged data.

ShowDropDownList; QM_ComboBox and QM_Edit controls

[ShowDropDownList](#)

[QM_ComboBox control](#)

[QM_Edit control](#)

[IQmDropdown](#)

[Using in exe](#)

Function ShowDropDownList

```
#ShowDropDownList ICsv'csv [int&iSelected] [ARRAY(byte) &aChecked] [funcFlags] [hwndHost]
[RECT&rDD] [cbFunc] [cbParam] [IQmDropdown&dd]
#ShowDropDownListSimple $csv [int&iSelected] [ARRAY(byte) &aChecked] [funcFlags] [hwndHost]
[RECT&rDD]
```

Shows a temporary top-level window containing a list of items that can be selectable (like in ComboBox controls), check box (more like in multi-select ListBox controls) or disabled (can be used as headers, separators etc). By default all items are selectable. Items can have icons, colors, etc.

[ShowDropDownListSimple](#) is a simplified version of [ShowDropDownList](#). It uses a CSV string as its first parameter, and several parameters are removed. Everything else is identical.

Returns one or more of these values (as [flags \(247\)](#)):

flag name	flag value	
QMDDRET_SELOK	1	Selected a selectable item (clicked or with Enter or Tab).
QMDDRET_CHECKCHANGED	2	Changed check box states.
QMDDRET_SELCHANGED	4	Changed the selected item. Can be only together with QMDDRET_SELOK .
QMDDRET_NOERRORS	0x100	DDL was shown (no errors).

Returns 0 if DDL could not be shown. For example, an invalid argument.

Parameters

csv - sets control properties, adds list items. With [ShowDropDownList](#) it is an [ICsv \(116\)](#) variable; it also receives results, if **iSelected/aChecked** not used. With [ShowDropDownListSimple](#) it is a [CSV \(116\)](#) string. Below is described the CSV format of the [ICsv](#) variable or CSV string.

First row - control properties. Columns:

- 0 - selected item index.
 - No selection if empty or -1.
 - If used control flag 2 (see below), this cell contains the 32-bit number for checked items.
 - This function uses and updates this cell if **iSelected** not used.
- 1 - imagelist. Can be:
 - Imagelist file or [resource \(261\)](#) name (if single line). To create imagelists you can use QM Imagelist Editor, look in floating toolbar.
 - List of icon files or resources (if multiple lines or ends with a newline character), enclosed in double quotes.
 - Imagelist handle (if a number). The function does not copy or destroy the imagelist. To load/create/destroy imagelists, you can use [__ImageList](#) variables.
- 2 - control [flags \(247\)](#):
 - 1 - with check boxes. To get/set check states, use **aChecked** or item flags (see below).
 - 2 - with check boxes. To get/set check states, use a 32-bit number where bits of checked items are 1. The number is stored in the place of the selected item index (**iSelected** or cell 0 0). Flag 1 and **aChecked** are not used.
 - 4 - don't close drop-down list when mouse is far.
 - 8 - auto colors. Text of disabled items will be gray. Background of "header" items (item flags 2|4|8) will be green. Not if item colors are explicitly specified.
 - 0x10 - multiple columns, horizontal scrollbar.
 - 0x20 - order items left to right, then top to bottom.
 - 0x40 - use **hwndHost/rDD** width as minimal DDL width (not exact width). Calculate real width from list item strings.
 - 0x80 - fast drop-down (no animation).

- 3 - unused.
- 4 - list text [color \(240\)](#) in 0xBBGGRR format.
- 5 - list background color in 0xBBGGRR format.
- 6 - column width, if used flag 0x10 or 0x20. Pixels if >0; dialog units if <0. Also sets minimal drop-down list width.
- 7 - maximal number of list items to show vertically in one scroll page.

Other CSV rows - list items. Columns:

- 0 - item text.
- 1 - item image index in imagelist. No image if empty or -1.
 - If imagelist not used, you can use this column for any purpose, for example as an item id to find it in CSV.
- 2 - item [flags \(247\)](#):
 - 1 checked. The function updates this flag, if need. Also adds/removes this flag: 0x80 changed check state.
 - 2 no check box. The item is selectable (if not disabled).
 - 4 disabled (cannot be selected).
 - 8 bold font. For example, for header items you can use item flags 0xE (no check box, disabled, bold font).
 - 16 selectable even if has check box. Click the check box to check/uncheck; click item text to select and close.
 - 32 group. Checking the item unchecks adjacent items that have this flag.
- 3 - item tooltip text.
- 4 - item text [color \(240\)](#), in 0xBBGGRR format.
- 5 - item background color, in 0xBBGGRR format.
- 6 - item indent.
- 7 - item data. Any text or number that you can use for any purpose. The function does not use it.

With [ShowDropDownListSimple](#), **csv** must be valid [CSV \(116\)](#) string. If a cell text contains special characters (comma (,), double quote (")), new line), it must be enclosed in double quotes. Double quotes in cell text must be replaced with two double quotes.

In numeric cells you can use decimal numbers, hexadecimal numbers (like "0xE"), and expressions with operator | (like "1|4|0x300").

CSV control flags and item flags have numeric [named constants \(139\)](#), defined in \System\Declarations\QmDll. The names can be used in CSV if it is [F-string \(138\)](#), for example `str`

```
csv=F",,{QMDD_CHECKBOXES1|QMDD_AUTOCOLORS} [ ] ...".
```

iSelected - variable that sets and receives selected item index.

- If omitted or 0, the function uses (gets/sets) **csv** cell 0 0.
- If used control flag 2, this variable (or **csv** cell 0 0) is used to store the 32-bit number for checked items.

aChecked - array variable that sets checked item states. Elements of checked items contain flag 1.

- The function updates **aChecked** elements if need. Elements will contain these flags: 1 checked, 0x80 changed check state.
- If omitted or 0, the function uses item flags in **csv**.
- If the array is empty or smaller, the function creates it if using check boxes.

funcFlags - function [flags \(247\)](#):

- [QMDDFF_CSVINPUT](#) (1) use **iSelected/aChecked** only for output, and **csv** for input.
- [QMDDFF_SORT](#) (8) - sort **csv** rows. Also updates cell 0 0 and **iSelected** to match the new selected item index. Not used with CSV flag 2 or if **aChecked** is used for input.

hwndHost - host control handle. The DDL will be below or above the host control, and will have its width and font.

rDD - a [RECT](#) variable. If used:

- The DDL will be in this rectangle on screen, and will have **rDD** width.
- If **rDD** width or/and height is 0, it does not limit DDL width or/and height.
- If **rDD** and **hwndHost** not used, the DDL will be by the mouse cursor.
- If DDL width specified by function parameters (**rDD**, **hwndHost** width etc) is less than height of two list items, the functions ignores it and calculates DDL width from list item strings.

Other parameters

cbFunc, **cbParam** - address of a callback function, and a value to pass to it. It will be called to notify about various events. Must begin with:

```
function# cbParam QMDROPDOWNCALLBACKDATA&x
```

cbParam - **cbParam** of [ShowDropDownList](#).

x - event info.

```
type QMDROPDOWNCALLBACKDATA IQmDropdown' dd code itemIndex itemState
```

dd - [IQmDropdown](#) variable. You can call functions **x.dd.Check**, **x.dd.IsChecked** etc.

code - event code:

- 1,2 - a check box state changed. **itemIndex** is item index. **itemState** is 1 if the item now is checked, 0 if not. When the user checks/unchecks, **code** is 1; when function [Check](#) does it, **code** is 2. Return 0.

dd - a variable that receives [IQmDropdown](#) interface pointer. Valid only while the DDL is visible. When [ShowDropDownList](#) returns, it sets the variable to 0. For example, if you call [ShowDropDownList](#) in a dialog procedure, and want to close the DDL when the procedure receives some message, you can call [dd.Close](#) then. Also you can call [dd.Check](#), [dd.IsChecked](#) etc.

Other features

Keyboard shortcuts (can use them only if **hwndHost** is used and is active and belongs to current thread):

- Down/Up/PgDn/PgUp - highlight items.
- Spacebar - toggle check state of the highlighted item.
- Enter or Tab - close the drop-down-list. The return value will contain [QMDDRET_SELOK](#) (1); it also can contain [QMDDRET_CHECKCHANGED](#) (2) or/and [QMDDRET_SELCHANGED](#) (4).
- Esc or Alt - close the drop-down-list. The return value will not contain [QMDDRET_SELOK](#), [QMDDRET_SELCHANGED](#); it can contain [QMDDRET_CHECKCHANGED](#).

Examples

Shows drop-down list and gets the selected item index.

```
str imagelist="$qm$\il_qm.bmp" ;;tip: to see images in QM code editor, check toolbar button "Images in code editor"
str csv=
F
, {imagelist}
one, 1
two, 2
three

int iSelected
int R=ShowDropDownListSimple(csv iSelected)
out F"R=0x{R}, selected={iSelected}"
```

Shows drop-down list with check boxes; uses a 32-bit variable to get checked items.

```
str icons="$qm$\new.ico[] $qm$\copy.ico[] $system$\shell32.dll, 4"
str csv=
F
, "{icons}", 2
one, 1
two, 2
three, , 1

int checked
int R=ShowDropDownListSimple(csv checked)
out F"R=0x{R} checkBoxes=0x{checked}"

int i
for i 0 3
if (checked>>i&1) out F"item {i} is checked"
```

Shows drop-down list with check boxes and gets checked items from CSV.

```
str csv=
, , 1
one, 1
two, 2
three, , 1

ICsv x._create; x.FromString(csv)

int R=ShowDropDownList(x)
out F"R=0x{R}"
x.ToString(_s); out _s
```

```
int i
for i 1 x.RowCount
  if(x.CellInt(i 2)&1) out F"item {i-1} is checked"
```

QM_ComboBox control

Controls of window class QM_ComboBox are ComboBox controls with a better drop-down list. Can have images, check boxes (multiple selections), etc. For the drop-down list part the control uses function [ShowDropDownList](#).

To add a QM_ComboBox control to a dialog, use the [Dialog Editor \(63\)](#). To set and get control data (properties and list items), use dialog variables created by the Dialog Editor.

Example dialog

\Dialog_Editor

```
str dd=
BEGIN DIALOG
0 "" 0x90C80AC8 0x0 0 0 224 136 "Dialog"
3 QM_ComboBox 0x54230243 0x0 8 8 96 213 ""
4 QM_ComboBox 0x54230242 0x0 8 28 96 213 ""
5 QM_ComboBox 0x54230243 0x0 8 48 96 213 ""
6 QM_ComboBox 0x54230243 0x0 8 68 96 213 ""
1 Button 0x54030001 0x4 116 116 48 14 "OK"
2 Button 0x54030000 0x4 168 116 48 14 "Cancel"
END DIALOG
DIALOG EDITOR: "" 0x2040300 "*" "" "" ""
```

```
str controls = "3 4 5 6"
```

```
str qmcb3 qmcb4 qmcb5 qmcb6
```

Simplest. Adds three items and selects Mercury (index 0).

```
qmcb3="0[]Mercury[]Venus[]Earth"
```

With icons. Adds three items, selects Mercury, and in the drop-down list displays the first three images from the imagelist file.

```
str imagelist="$qm$\il_qm.bmp" ;;tip: to see images in QM code editor, check toolbar button "Images in code editor"
```

```
qmcb4=F"0,{imagelist}[]Mercury,0[]Venus,1[]Earth,2"
```

With check boxes. Adds three items, checks Mercury (its item flags contain 1), and sets control text to "Mercury".

```
qmcb5="",1[]Mercury,,1[]Venus,,0[]Earth"
```

With check boxes, using a 32-bit number to check items. Adds three items, checks Mercury and Earth (first and third bits in 0x5 are 1), and sets control text to "Mercury, Earth".

```
qmcb6=
0x5,,3
Mercury
Venus
Earth
```

```
if(!ShowDialog(dd 0 &controls)) ret
```

```
out qmcb3
```

```
out qmcb4
```

```
out qmcb5
```

```
out qmcb6
```

How to set control properties and list items

Assign a [CSV \(116\)](#) string to the dialog control variable before calling [ShowDialog \(63\)](#).

First CSV row - control properties. Columns:

- 0 - selected item index.
 - No selection if empty or -1.
 - If like "-1 text", sets editable control text. Or can be just "text", if the text does not begin with a number. Only if the control is without check boxes.
 - If used control flag 2 (see below), this cell contains the 32-bit number for checked items.

- 1 - imagelist. Can be:
 - Imagelist file or [resource \(261\)](#) name (if single line). To create imagelists you can use QM Imagelist Editor, look in floating toolbar.
 - List of icon files or resources (if multiple lines or ends with a newline character), enclosed in double quotes.
 - Imagelist handle (if a number). The control does not copy or destroy the imagelist; it means that the imagelist should be alive until the control is destroyed or its items replaced or deleted. To load/create/destroy imagelists, you can use `__ImageList` variables.
- 2 - control [flags \(247\)](#):
 - 1 - with check boxes. Use item flags to get/set check states. Control text format: list of checked item labels like "Item1, Item2".
 - 2 - with check boxes. To get/set check states, use CSV cell 0 0. It is a 32-bit number where bits of checked items are 1. Control text format depends on flags 1 and 0x800, default is hex number.
 - 4 - don't close drop-down list when mouse is far.
 - 8 - auto colors. Text of disabled items will be gray. Background of "header" items (item flags 2|4|8) will be green. Not if item colors are explicitly specified.
 - 0x10 - multiple columns, horizontal scrollbar.
 - 0x20 - order items left to right, then top to bottom.
 - 0x40 - use control width as minimal drop-down list width (not exact width). Calculate real width from list item strings.
 - 0x80 - fast drop-down (no animation).
 - 0x100 - for a dialog variable get CSV. Read more below.
 - 0x200 - on click don't select all text in Edit control.
 - 0x400 - don't clear Edit control text when setting selected item = none or setting control data where checked items = none.
 - 0x800 - when flag 2 used, display checked items like "1|2|4|0x300" instead of "0x307".
 - 0x1000 - auto-add unknown items to the list from Edit text. Adds before showing drop-down list and before getting control data. Selects or checks the added items. Not used with flag 2.
- 3 - cue banner text. It is a brighter text displayed in the control when it is not focused and its text is empty.
- 4 - list text [color \(240\)](#) in 0xBBGGRR format.
- 5 - list background color in 0xBBGGRR format.
- 6 - column width, if used flag 0x10 or 0x20. Pixels if >0; dialog units if <0. Also sets minimal drop-down list width.
- 7 - maximal number of list items to show vertically in one scroll page.

Other CSV rows - list items. Columns:

- 0 - item text.
- 1 - item image index in imagelist. No image if empty or -1.
 - If imagelist not used, you can use this column for any purpose, for example as an item id to find it in CSV.
- 2 - item [flags \(247\)](#):
 - 1 checked.
 - 2 no check box. The item is selectable (if not disabled).
 - 4 disabled (cannot be selected).
 - 8 bold font. For example, for header items you can use flags 0xE (no check box, disabled, bold font).
 - 32 group. Checking the item unchecks adjacent items that have this flag.
- 3 - item tooltip text.
- 4 - item text [color \(240\)](#), in 0xBBGGRR format.
- 5 - item background color, in 0xBBGGRR format.
- 6 - item indent.
- 7 - item data. An integer number to use with messages `CB_GETITEMDATA` and `CB_SETITEMDATA`. If you don't use these messages, it can be any text too, the control does not use it.

It must be valid [CSV \(116\)](#) string. If a cell text contains special characters (comma (,), double quote (")), new line), it must be enclosed in double quotes. Double quotes in cell text must be replaced with two double quotes. To create valid CSV at run time, use an [ICsv \(116\)](#) variable.

In numeric cells you can use decimal numbers, hexadecimal numbers (like "0xE"), and expressions with operator | (like "1|4|0x300").

List items in the CSV are optional. For example, you may want just to set control properties. You can add items later, when handling `CBN_DROPDOWN` notification in dialog procedure. If on `CBN_DROPDOWN` you don't add items, the control will not show a drop-down list. On `CBN_DROPDOWN` you can instead call [ShowDropDownList](#), or show a message box, popup menu, whatever, or do nothing; but for this purpose probably it is better to use `QM_Edit` control.

If you want to set control data in dialog procedure, use function [DT_SetControl](#) or [CB_Add](#) or `CB_INSERTITEM` messages. Function [str.setwintext \(224\)](#) sets editable control text.

How to get control data

When [ShowDialog](#) returns on OK, the dialog control variable contains a string in a format that depends on control style and flags.

By default it is:

- If the control is editable: control text.
- Else if the control is without check boxes: selected item index (-1 if none selected). Use [val \(186\)](#) to convert to int.
- Else (if with check boxes):
 - If used control flag 2: the 32-bit number (read more above). Use [val](#) to convert to int.
 - Else: string consisting of '1' characters for checked items and '0' characters for unchecked. For example, if there are 3 items and only the first item is checked, the string is "100".

If used control flag 0x100, it is initial CSV string (you can use [ICsv \(116\)](#) to parse it), modified depending on user-made selection:

- If the control is without check boxes:
 - If the control is read-only: CSV cell 0 0 is the selected item index (-1 if none selected).
 - Else (if editable): CSV cell 0 0 is the selected item index followed by control text, like "0 First item" or "-1 any text".
- Else (if with check boxes):
 - If used control flag 2: CSV cell 0 0 is the 32-bit number (read more above).
 - Else: item flags (column 2 of list items) of checked items have flag 1; item flags of changed-check-state items have flag 0x80.

If you want to get control data in dialog procedure, use function [DT_GetControl](#). Function [str.getwintext \(224\)](#) gets control text.

Styles, messages and notifications

The control is based on standard ComboBox control (except the drop-down list part) and supports the most important its features.

Supported ComboBox styles: CBS_DROPDOWN (editable), CBS_DROPDOWNLIST (read-only), CBS_SORT, CBS_AUTOHSCROLL, CBS_LOWERCASE, CBS_UPPERCASE. Cannot be CBS_SIMPLE or owner-draw.

Supported ComboBox messages: CB_GETDROPPEDSTATE, CB_ADDSTRING, CB_INSERTSTRING, CB_DELETESTRING, CB_RESETCONTENT, CB_GETCOUNT, CB_GETCURSEL, CB_GETITEMDATA, CB_SETITEMDATA, CB_GETLBTEXT, CB_GETLBTEXTLEN, CB_SHOWDROPDOWN, CB_GETCOMBOBOXINFO, CB_GETCUEBANNER, CB_GETEDITSEL, CB_LIMITTEXT, CB_SETCUEBANNER, CB_SETEDITSEL.

The control supports QM CB_ functions ([CB_SelectedItem](#) etc). The functions send the above messages.

Partially supported ComboBox messages: CB_GETITEMHEIGHT, CB_SETITEMHEIGHT - only if wParam is -1. Messages that modify the list fail and return -1 when the list is dropped down. CB_GETDROPPEDWIDTH and CB_SETDROPPEDWIDTH not supported but automatic width can be specified in CSV (column 6, flag 0x40).

Other messages: LB_SETSEL sets item check state. LB_GETSEL gets item check state.

Supported ComboBox notifications: CBN_CLOSEUP, CBN_DROPDOWN, CBN_EDITCHANGE, CBN_EDITUPDATE, CBN_KILLFOCUS, CBN_SELCHANGE, CBN_SELENDON, CBN_SELENDOK, CBN_SETFOCUS.

The above messages, notifications and styles are documented in the MSDN Library.

Other notifications

The control also sends other notifications using WM_NOTIFY message, where lParam is address of a [NMQMCB](#) variable. Its first member hdr is of type [NMHDR](#) (documented in MSDN). Values of other members depend on notification code (hdr.code).

```
type NMQMCB NMHDR 'hdr IqmDropDown' dd itemIndex itemState
```

Notification codes:

- 1 - the user checked or unchecked a check box in the drop-down list. The notification is not sent when message LB_SETSEL or function [dd.Check](#) does it. Members of the [NMQMCB](#) variable:

itemIndex - item index.

itemState - 1 if the item now is checked, 0 if not.

dd - [IqmDropDown](#) variable. You can call functions [dd.Check](#), [dd.IsChecked](#) etc.

Other features

Incremental search. When the user changes editable control text while the drop-down list is shown, the control highlights the first item whose text begins with or contains the new text. Use Enter or Tab to select it. Only if without check boxes.

Adds large number of items much faster than ComboBox.

Shows "..." and infotip for clipped items.

Keyboard and mouse shortcuts:

- Down/Up/PgDn/PgUp - show drop-down list.
- Middle-click - clear editable control text.

Keyboard shortcuts in drop-down list:

- Down/Up/PgDn/PgUp - highlight items.
- Spacebar - toggle check state of the highlighted item.
- Enter or Tab - select the highlighted selectable item and close the drop-down-list. Apply new check box states.
- Esc or Alt - close the list without selecting. Apply new check box states if there are no selectable items.

QM_Edit control

Controls of window class QM_Edit are Edit controls that can have one or two buttons next to the edit field.

By default the control has an arrow button, similar as in ComboBox controls. When pressed, the control sends **CBN_DROPDOWN** command message to the parent dialog. The dialog procedure then can show a drop-down list (see example) or do whatever, and then set control text if need. The control itself does not show a drop-down list etc.

The control also can have another button (*user button*). When clicked, the control sends **BN_CLICKED** command message to the parent dialog. The dialog procedure then can do whatever, for example show a "Open file" dialog (see example) and set control text.

To add a QM_Edit control to a dialog, use the [Dialog Editor \(63\)](#). To set/get control text before/after [ShowDialog \(63\)](#), use dialog variables created by the Dialog Editor. Dialog variables contain control text. To set/get control text in dialog procedure, you can use [str.setwintext/str.getwintext \(224\)](#) or [DT_SetControl/DT_GetControl](#).

The control supports all Edit control styles, messages and notifications.

Messages specific to QM_Edit control:

- **QMEM_SETBUTTON** - adds/removes buttons and sets user button image (16x16 pixels).
 - **wParam** - [flags \(247\)](#):
 - Flags 0, 1 and 2 are used to set user button image:
 - 0 - **IParam** is icon file path or [resource \(261\)](#) name. Can be 0.
 - 1 - **IParam** is icon handle. The control makes its own copy of the icon.
 - 2 - **IParam** is imagelist handle. High-order word of **flags** is image index. Use [MakeInt\(flags, index\)](#). The control makes its own copy of the image. To create imagelists you can use QM Imagelist Editor, look in floating toolbar. To load or create from icons, use class [__ImageList](#).
 - 0x100 - don't remove arrow button. The control will have both buttons.
 - 0x200 - remove user button.
 - **IParam** - depends on **wParam**. If **IParam** is 0, the control will draw an "open folder" image.

WM_COMMAND notifications specific to QM_Edit control:

- **CBN_DROPDOWN** - the user pressed the arrow button or Down/Up/PgDn/PgUp key. The dialog procedure can show a drop-down list or whatever, as described above. See example.
- **BN_CLICKED** - the user clicked the user button or pressed Enter (if available in that window) or Alt+Enter.

Keyboard and mouse shortcuts:

- Down/Up/PgDn/PgUp - generate **CBN_DROPDOWN** notification, like when the user presses the arrow button. Need Alt if the control is multiline.
- Enter - generate **BN_CLICKED** notification, like when the user clicks the user button. Need Alt if the control is multiline or if Enter closes the parent window.
- Middle-click - clear text.

You can replace existing Edit controls with QM_Edit controls:

- Replace Edit with QM_Edit in dialog definition.
- Add code to handle the notifications (**CBN_DROPDOWN** or/and **BN_CLICKED**). See example.
- Optionally set button image etc (**QMEM_SETBUTTON**) under **WM_INITDIALOG**.
- Don't need more changes in your code, because QM_Edit controls work like Edit controls, they just have buttons added.

To create a multiline control, add styles **ES_MULTILINE**, **ES_WANTRETURN**, **ES_AUTOVSCROLL**, **WS_VSCROLL** and **WS_EX_LEFTSCROLLBAR**. Remove **ES_AUTOHSCROLL** if need to wrap lines.

Example

```
\Dialog_Editor

str dd=
BEGIN DIALOG
0 "" 0x90C80AC8 0x0 0 0 224 136 "Dialog"
3 QM_Edit 0x56030080 0x200 8 8 96 12 ""
4 QM_Edit 0x56030080 0x200 8 28 208 12 ""
1 Button 0x54030001 0x0 168 116 48 14 "OK"
END DIALOG
DIALOG EDITOR: "" 0x2040300 "*" "" "" ""

str controls = "3 4"
str qme3 qme4
qme3="zero"
if(!ShowDialog(dd &sub.DlgProc &controls _hwndqm)) ret
out qme3
out qme4

#sub DlgProc
function# hDlg message wParam lParam

sel message
_case WM_INITDIALOG
_SendDlgItemMessage hDlg 4 QMEM_SETBUTTON 0 0 ;;add user button "open folder" and remove arrow
_SendDlgItemMessage hDlg 4 EM_SETCUEBANNER 0 @ "File"
-
_case WM_COMMAND goto messages2
ret
messages2
sel wParam
_case CBN_DROPDOWN<<16|3 ;;pressed the arrow button
_show drop-down list, like ComboBox
_lpstr o=":5 $qm$\il_qm.bmp"
_str s=F"0,{o}[]zero,1[]one,2[]two,28"
_ICsv x._create; x.FromString(s)
_int i R=ShowDropDownList(x i 0 1 lParam)
_if(R&QMDDRET_SELOK=0) ret
_s=x.Cell(i+1 0); s.setwintext(lParam)
-
_case 4 ;;clicked the user button (BN_CLICKED)
_str sFile
_if(OpenSaveDialog(0 sFile)) sFile.setwintext(lParam)
ret 1
```

Interface IQmDropdown

Can be used in callbacks, notifications, etc, while the drop-down list is shown.

```
interface# IQmDropdown :IUnknown
_Check(i flags)
#IsChecked(i)
#Select(i [flags])
_Close([flags])
```

```

_Update(iFirst [iLast] [flags])
[g] #Hwnd()
[g] #HwndLV()
[g] #SelectedItem()
[g] ARRAY(byte) ItemStates()
[g] ICsv'Csv()
_{A461B246-3C33-4398-915B-0B834A60A670}

```

```
Check(i flags)
```

Changes checkbox state of item i.

Returns 1. Returns 0 if i is invalid or the item is not a check box.

flags: 0 uncheck, 1 check, 2 toggle, 0x100 don't call **cbFunc**.

```
#IsChecked(i)
```

Returns 1 if item i is checked, 0 if not.

```
#Select(i [flags])
```

Selects/highlights item i.

Returns new selected item index.

flags: 1 don't ensure visible, 2 skip disabled down, 4 skip disabled up.

If i is invalid (eg -1), removes selection and returns -1. Also if the item is disabled, if not used flag 2 or 4.

Does not close the window. Call [Close](#) for it.

```
Close([flags])
```

Closes the DDL window.

flags: 1 OK (close like when the user clicks a selectable item or presses Enter or Tab).

```
Update(iFirst [iLast] [flags])
```

Redraws one or more list items.

Call this function to apply changes to item properties in **csv** that you get with [Csv](#).

iFirst - 0-based index of first item to update.

iLast - 0-based index of last item to update. If omitted or <= **iFirst**, updates only **iFirst**.

flags: 1 item flags (third column) changed in **csv**. Don't need this flag if changed other columns (colors, text etc).

```
[g] #Hwnd()
```

Gets drop-down window handle. It is a top-level window.

```
[g] #HwndLV()
```

Gets listview control handle. It is a child window.

```
[g] #SelectedItem()
```

Gets the currently highlighted item index or -1.

```
[g] ARRAY(byte) ItemStates()
```

Gets array of item check states and other states/styles. Elements contain item flags from **csv** passed to [ShowDropDownList](#) (or from CSV string passed to QM_ComboBox control). These flags are updated: 1 checked, 0x80 check box state changed.

```
[g] ICsv'Csv()
```

Gets the **csv** variable passed to [ShowDropDownList](#) (or an **ICsv** variable created from CSV string passed to QM_ComboBox control). You can change item text, icon, tooltip, colors and indent: modify item cells in **csv** and call [Update](#). You can add more columns, but don't add/remove rows and don't modify the first row (control properties).

How to use in exe

All these functions, interfaces and controls can be used in [exe \(51\)](#) too. All they live in a QM dll qmgrid.dll, which it is not available on computers where QM is not installed.

You can add the dll to exe. Add this line somewhere near the beginning of exe code:

```
ExeQmGridDll
```

When creating exe, it adds qmgrid.dll to exe resources. At run time it extracts to the temporary folder and loads.

If you don't want to add the dll to exe: take qmgrid.dll from QM folder; add to a zip file or setup program together with exe; on other computers extract them to the same folder. If you use the controls in dialogs or with [CreateControl](#), exe loads the dll when need, else you can load it with [LoadLibrary](#).

You can add imagelist and icon files to exe resources. In "Make Exe" dialog check "Auto add files..." and in macro add unique resource id numbers to file paths, like ":5 \$my qm\$imagelist.bmp". Alternatively you can use [macro resources \(261\)](#). In the Resources dialog you can import files and create a resource string like "resource:<macro5>image:imagelist" or "image:imagelist". The file/resource name must be in a separate variable (see example), not directly in CSV string. If you don't want to add files to exe, make sure that they are available on computers where exe will run.

Example

```
str imagelist=":5 $qm$\il_qm.bmp" ;;5 is a resource id, can be 1-0xffff
str csv=
F
,{imagelist}
one,1
two,2
three
```

Grid control (QM_Grid)

Changes in QM 2.3.2

Before QM 2.3.2, this control was almost undocumented. Some samples were in the forum. In QM 2.3.2 it is documented, easier to use, and has many enhancements:

- Can be used in exe.
- Faster.
- Does not limit cell text length.
- Cut/Copy/Paste rows or row text.
- Several new messages, styles and controls.
- Easier to add columns and set row control types.
- Style and columns can be defined in Dialog Editor.
- Dialog variables can be used to set/get grid data.
- Added new class [DlgGrid](#) that simplifies working with the control in dialog procedures.
- Added new class [Sqlite](#) that has functions to display a sqlite database table in grid control.
- Better supports images and other row properties.
- **Possible incompatibility**: changed some features. If you used the control in your macros, please test them. The changes are documented below, in grid.h.

Where to use

The control can be used in dialogs and other windows to:

- Display text data as editable table (data grid). Like in Excel.
- Display multiple properties (property grid). Single grid control can replace many simple controls (edit, combo box, check box).
- Display text data as read-only table. The same as SysListView32 control in report view, but faster and easier to use.
- Display a list with check boxes.

How to use

How to add to a dialog

Use [Dialog Editor \(63\)](#). It also allows you to define style and columns.

You also can define style and columns at run time. Use class [DlgGrid](#). See function [sample_Grid](#).

How to set data

Dialog Editor generates code that includes variables for grid controls. Before calling [ShowDialog](#), assign a string in CSV format to a grid variable. To load and process CSV, use [ICsv \(116\)](#) interface.

You can also set data using class [DlgGrid](#). See function [sample_Grid](#).

How to get data

When [ShowDialog](#) returns on OK, the variable contains string in CSV format. To process and save it, use [ICsv \(116\)](#) interface.

You can also get data using class [DlgGrid](#). See function [sample_Grid](#).

How to work with the control in dialog procedure

Use class [DlgGrid](#). See function [sample_Grid](#).

Also you can use grid control and list view control (SysListView32) messages, notifications, types and constants.

Grid control messages etc are documented below, in grid.h. In QM they are declared in reference file `GRID`. Use like this: `GRID.LVN_QG_COMBOFILL`.

List view control messages etc are documented in [MSDN Library \(256\)](#). They are declared in reference files `WINAPI` and `WINAPIV`.

How to set control types for rows or cells

In dialog procedure use functions of class [DlgGrid](#).

If the grid has style "Can be set row control type" (QG_SETROWTYPE), you also can set row control types when adding text. Use syntax: <type>text in first column cells. Read more below, in Details.

Example

```
\Dialog_Editor

str controls = "3"
str qmg3
qmg3=
<0>edit,x
<1>combo,x
<2>check,Yes
<7>read-only,x
<8>edit multiline,x
<9>combo sorted,x
<16>edit+button,x
<17>combo+button,x

if(!ShowDialog("Dialog90" 0 &controls)) ret
out qmg3

BEGIN DIALOG
0 "" 0x90C80AC8 0x0 0 0 223 135 "Dialog"
3 QM_Grid 0x54030000 0x0 0 0 224 110 "7,0,0,2[A,,][B,,]"
1 Button 0x54030001 0x4 120 116 48 14 "OK"
2 Button 0x54030000 0x4 170 116 48 14 "Cancel"
END DIALOG
DIALOG EDITOR: "" 0x2030202 "*" "" ""
```

How to set icons, lparam and other list view control item properties

In dialog procedure use functions of class [DlgGrid](#).

To add icons, at first create imagelist with QM imagelist editor. It is in floating toolbar. It also can create code to load the imagelist. In dialog procedure, load the imagelist ([__ImageList.Load](#)), attach it to the control ([DlgGrid.SetImagelist](#)), and assign icon indices ([DlgGrid.RowPropSet](#) or [DlgGrid.RowAddSetX](#)).

If the grid has style "Can be set row control type" (QG_SETROWTYPE), you also can set these properties when adding text. Read more below, in Details. See function [sample_Grid_images](#).

How to use with databases

It is easy with sqlite databases. Use [Sqlite](#) class. It can display table in grid control. See function [sample_Grid_Sqlite](#).

To use with other database types, use [Database](#) and [DlgGrid](#) classes. Currently there are no functions that would transfer whole table from/to database to grid control. You will have to do it for each row.

How to use in exe (51)

QM grid control lives in a QM dll qmgrid.dll. It is not available on computers where QM is not installed.

You can add the dll to exe. Add this line somewhere near the beginning of exe code:

```
ExeQmGridDll
```

When creating exe, it adds qmgrid.dll to exe resources. At run time it extracts to the temporary folder and loads.

If you don't want to add the dll to exe: take qmgrid.dll from QM folder; add to a zip file or setup program together with exe; on other computers extract them to the same folder. If you use grid controls in a dialog or with [CreateControl](#), exe loads the dll when need. Else load the dll with [LoadLibrary](#).

Details

The class name is QM_Grid. Implemented in qmgrid.dll.

Grid row properties

If grid control has "Can be set row control type" style (GRID.QG_SETROWTYPE, 4):

1. When you set control type of the first cell in a row, it also sets control type for other cells in the row.
2. When adding a row, you can specify row control type and/or other row properties in first cell's text.

Syntax:

```
<ctype/lparam/image/overlayImage/stateImage/indent/ctypes>text
```

ctype - row control type. See below.

lparam - a user-defined int value. Or can be any string.

image - image index. Use -1 for no image.

overlayImage - overlay image index.

Valid values are 1 to 15. Index 0 will not add overlay image.

Overlay images are drawn transparently over normal images.

To define overlay images, use `__ImageList.SetOverlayImages` or `ImageList_SetOverlayImage` with the main imagelist.

stateImage - state image index.

Valid values are 1 to 15. Index 0 will not add state image.

The index is 1-based, therefore index 1 will display image 0 from the state imagelist.

State images are drawn in the space reserved to the left from normal images.

Must be set state imagelist or checkboxes style.

If checkboxes style, use **stateImage** 1 and 2 for unchecked and checked.

indent - indent. It is number of image widths. Valid values are -1 to 254.

ctypes - list of cell control types. Use . to inherit from **ctype** or column. Use -1 to inherit from column. Example: .0 -1..2

All parts are optional. For example, can be `<ctype>text`, or `<!/image>text`.

All numbers must be simple numbers, not constant names or expressions.

To display images and indent, imagelist must be set. Use `DlgGrid.SetImagelist`.

Row properties also can be set/retrieved using `DlgGrid` functions.

If style "User cannot edit first column" (GRID.QG_NOEDITFIRSTCOLUMN, 1) also is set, will not remove the `<...>` string.

Just will not display it.

Control types

0	edit.
1	combo.
2	check.
3	date.
7	read-only.
8	edit multiline.
(0 8)	
9	combo sorted.
(1 8)	
11	time.
(3 8)	
16	with button.
(flag)	
23	read-only edit + button.
(7 16)	

C++ header file

[grid.h](#)

Math functions from C run-time library

C run-time library (msvcrt.dll) contains many useful functions. You can find reference in [MSDN library \(256\)](#).

Several examples:

To calculate floating-point remainder, use function `fmod`.

Example: `double rem=fmod(x y)` .

To calculate x raised to the power of y, use function `pow`.

Example: `double pw=pow(x y)` .

To calculate x square root, use function `sqrt`.

Example: `double sq=sqrt(x)` .

Before using dll functions, they must be [declared \(153\)](#). For example, if function in MSDN library is given as:

```
double fmod(
double x,
double y
);
```

then QM declaration would be:

```
dll msvcrt ^fmod ^x ^y
```

QM declares several math functions from C run-time library. They are added to math [category \(159\)](#).

Example

```
double angleDegrees=30

def PI 3.1415926535897932384626433832795

double s=sin(angleDegrees*PI/180) ;;converts degrees to radians and calculates sin
double a=asin(s)*180/PI ;;calculates arcsin and converts radians to degrees
out s
out a
```

AR Services Manager Library 1.03

Obsolete. Does not work on Windows 10.

This library manages Windows services. It is a COM component.

If used in exe, copy ARServicesMgr.DLL from QM folder to your exe folder on other computers. Don't need to register COM component.

Example - enumerate services

```
Services.clsServices ses._create
Services.clsService se
foreach se ses
_out se.DisplayName
_
```

Example - state, start etc

```
Services.clsService se._create
BSTR dispName="Quick Macros"
se.DisplayName=dispName
if(!se.ServiceType) mes- "Service not found." "" "x"

str state
sel se.CurrentState
_case Services.Stopped state="stopped"
_case Services.Start_Pending_20 state="start pending"
_case Services.Stop_Pending_20 state="stop pending"
_case Services.Running state="running"
_case Services.continue_pending_20 state="continue pending"
_case Services.Pause_Pending_20 state="pause pending"
_case Services.Paused state="paused"
_case else state="unknown"

int i=ListDialog("Start[]Stop[]Pause[]Continue" F"Current state: {state}")
if(!i) ret
if(!IsUserAdmin) mes- "To change service state, QM must be running as admin." "" "x"
sel i
_case 1 se.StartService
_case 2 se.StopService
_case 3 se.PauseService
_case 4 se.ContinueService
```

The classes included in this library are:

clsServices

This class can be used to list and access services in a system. You can specify what type of services to want to have listed.

Properties:

- ComputerName. By default, this property is set to the name of the computer in which the library is running. You can specify the name of another computer to access the services running there.
- Count. Number of services listed.
- Item(Index). This property will take you to a clsService object that contains the properties and the methods to access the service specified. You can specify a service by putting the number that is has in the collection or by passing its name.
- ServicesEnumState. The possible values for this property are: Active Services, Both Active and Inactive Services or Inactive Services. Specify here what type of services you want to list. The library will update itself if this property is changed.
- ServicesEnumType. The possible values for this

Methods:

- Delete(Index). Pass the number that a service has in the collection or the name of a service to delete it. The method returns True (nonzero) if successful or False (0) if there was an error. After you delete a service, it is not immediately removed from the services, if it has references to it. If you try to call the function again after you got True (successful), the method will return False.
- Refresh. Call this method to refresh the contents of the object. Returns True if successful or False if there was an error.

property are: WIN32 Services, Kernel and File System Drivers or Both WIN32 and Drivers. Specify here what type of services you want to have listed. The library will update itself if this property is changed.

clsService

This class is returned by the clsServices class (one per service listed), but you can also use it independently to access a specific system, in the local machine or through a network. To access a specific service, just set the name or the display name of the service you want to access. The class will update itself automatically with the information of the service. Using this class you can also start, stop, pause or continue services.

(r/w) means read/write property; (r-o) means read-only property.

Properties:

- **AutoUpdate (r/w).** If this property is True (nonzero), when you tell the service to change its state (stop it, start it, pause it...) the library will start checking for the state of the service till its state changes completely or till the time specified in AutoUpdateTimeout expires. This is because usually a service doesn't change its state immediately, but needs some time for processing. Therefore, before when you changed the state of a service, the CurrentState property passed to something like 'Start Pending', and after that you had to refresh the state manually. This property is automatically True if you are using this class independently, and automatically False if you are accessing from a clsServices class.
- **AutoUpdateInterval (r/w).** How often the library will refresh the status of the service, given in milliseconds.
- **AutoUpdateTimeout (r/w).** If the service doesn't change its status, for how long the library should keep refreshing its contents, in milliseconds.
- **BinaryPathName (r/w).** Path to the binary file of the service.
- **CheckPoint.**
- **ComputerName (r/w).** By default, this property is empty if you are using this class independently, or has the same value as the class clsServices is this class is an item of the other. You can specify another name to access a service in another computer.
- **ControlsAccepted (r-o).** Possible values are: Accept Pause & Continue, Accept Shutdown or Accept Stop. Also, it could be any combination of these values. To know if a service accepts a certain control, use operator &. Example:
if(Svc.ControlsAccepted&Accept_Stop_20).
- **CurrentState (r/w).** Possible values are: Continue Pending, Pause Pending, Paused, Running, Start Pending, Stop Pending or Stopped. You can also set this property to Start, Stop or Pause to modify the state of the service.
- **Dependencies (r/w).** This property will return an object that contains the dependencies of the service, if any. You can also create this class independently and set it to this property to change the dependencies of the service.
- **DisplayName (r/w).** Display name for the service (those you can see in the services control manager). If you change this property, the class will look for another service that has that display name, and if it finds it, it will update itself with the information of that service.
- **LoadOrderGroup (r/w).**
- **Name (r/w).** Internal name of the service. You can change this property to the name of other service, and

Methods:

- **ChangeAccount(strUserID, strPassword, [strDomain]).** If you want to change the account under which a WIN32_OWN_Process service is run, you can't use the StartName and Password properties independently, since both must be changed at the same time. Use this method instead.
- **ChangeDisplayName(strDisplayName).** You can't use the DisplayName property because if you change it, the class will just look for another service with that display name, instead of changing that property for the current service. Use this method instead.
- **ContinueService.** Use this method to continue the service if it is paused. Returns True of successful or False if an error occurred.
- **PauseService.** Use this method to pause a service if it is running. Returns True of successful or False if an error occurred.
- **StartService.** Use this method to start a service if it is stopped. Returns True of successful or False if an error occurred.
- **StopService.** Use this method to stop a service if it is running. Returns True of successful or False if an error occurred.
- **Refresh.** Forces the library to update the information about the service. Returns True of successful or False if an error occurred.

Events:

- **StateChanged(INewState).** This event is fired when the service changes its status.

the class will update itself with the configuration of that service.

- Password (r/w). You can use this property to change the password of the account under which a service is run. If you do it, this property will also contain that password if the change is successful. If it is not, or you haven't changed the password, this property will be empty (what doesn't mean that the account doesn't have any password).
- ServiceSpecificExitCode (r-o).
- ServiceType (r/w). Possible values are: File System Driver, Kernel Driver, WIN32 Own Process, WIN32 Share Process, Interactive WIN32 Own Process or Interactive WIN32 Share Process.
- StartErrorSeverity (r/w). Possible values are: Ignore Error, Normal Error, Severe Error or Critical Error. Informs on how the server will act if an error occurs at startup.
- StartName (r/w). Depending on the type of service, this can be the name of the account under which the service is running.
- StartType (r/w). Possible values are: Auto Start, Boot Start, Demand Start, Disabled, System Start. Indicates how the service is started.
- TagID (r/w).
- WaitHint (r-o).
- Win32ExitCode (r-o).

clsSVCDependencies

This object is used to manage dependencies of the services.

Properties:

- Count. Number of dependencies being listed.
- Item(Index). Pass a number and this property returns the dependency that has that number in the collection.
- Source. This property returns a char 0 divided string with the dependencies. This is how they are originally given by the system.

Methods:

- Add(strCad). Use this method to add a dependency to the object.
- Remove(Index). Use this method to remove dependencies from the object.

Go to label

Syntax

```
goto label
...
(space)label
...
```

Parameters

label - label name.

label - label to go to. The line must begin with space or semicolon.

Remarks

Execution will continue after **label**.

Examples

```
goto g1
...
g1
...
```

```
...
g2
...
if(a>10) goto g2
```

If ... else

Syntax

```
if expression
(tab)statements
(tab)...
[else
(tab)statements
(tab)...]
```

Can be single line:

```
if(expression) statements
[else statements]
```

or

```
if(expression) statements [else statements]
```

Parameters

expression - any [expression \(244\)](#).

Remarks

If **expression** is true (not 0), executes **statements** after **if** and skips **statements** after **else** (if used). Else skips **statements** after **if** and executes **statements** after **else** (if used).

`else if` does not require double indentation of following lines. See example.

See also: [Operators \(133\)](#), [ifa \(78\)](#), [ifk \(59\)](#), [iif \(124\)](#), [sel \(125\)](#)

Examples

```
if i=10
_out "i is 10"

if i=10
_out "i is 10"
else
_out "i is not 10"

if i > 0
_out "i is greater than 0"
if(s.endi(".exe")) ret i; else ret 0
else if i < 0
_out "i is less than 0"
else
_out "i is 0"

if a and b
_out "a is not 0, and b is not 0"
if c or d or e
_out "c is not 0, or d is not 0, or e is not 0"
```

Inline if

Syntax

any `iif(x a b)`

Parameters

x - any [expression \(244\)](#) that is either true (not 0) or false (0).
a, b - expressions of any type. Can be numeric or string values, variables, functions, expressions with operators.

Remarks

Selects one of two expressions. If **x** is true (not 0), returns **a**, else returns **b**.

Does not evaluate the other expression. For example, if **a** and **b** are functions, calls only one of them.

Expressions **a** and **b** must have same or similar types. If one is pointer or string, other can be 0. The return type depends on types of **a** and **b**. If both are pointers of different types, the return type is type of **b**.

See also: [Operators \(133\)](#) [if \(123\)](#)

Example

```
if i<10 then j=i, else j=10  
j=iif(i<10 i 10)
```

Select

Syntax

```
sel expression [flags]
(tab)case a
(tab)statements
(tab)...
(tab)case b
(tab)statements
(tab)...
[ (tab)case else
(tab)statements
(tab)...]
```

Statements can be in the **case** line:

```
sel expression [flags]
(tab)case a statements
[ (tab)case else statements]
```

All can be in single line:

```
sel(expression [flags]) case a statements ... [case else statements]
```

Parameters

expression - variable or other [expression. \(244\)](#) Type - integer or string.

a, b - integer or string [constants \(137\)](#). Type must match type of **expression**.

- Can be several values enclosed in square brackets, like `case [1,2]`.

flags (247) - how to compare strings:

1	case insensitive.
2	use wildcard characters (196) in a, b , ...
4 (QM 2.4.1)	use regular expression (198) in a, b , ... that begin with \$ character. For example, string "\$^a\d+\$" is compared as regular expression "^a\d+\$". Strings that don't begin with \$ are compared as simple strings or strings with wildcard characters (if flag 2 also used).
8 (QM 2.4.1)	if expression is 0 (null) compare it as "". Without this flag, null strings don't match "".

Remarks

If value of **expression** matches one of case constants, execute **statements** after that **case** (until next **case**, if any). Otherwise, execute statements after **case else** (if any).

case keyword can be followed by colon.

See also: [SelInt](#), [SelStr](#)

Examples

If variable msg is 5, display 1; else if msg is WM_COMMAND, display 2 and play sound; else if msg is 512 or 513 or 514, display 3 and play sound; else display msg:

```
sel msg
case 5: out 1
case WM_COMMAND: out 2; bee
case [512,513,514]
out 3
bee
case else out msg
```

Execute different code depending on weekday

```
str weekday.time("%A")
sel weekday
case "Monday": out 1
-
case "Tuesday"
out 2
out "today is Tuesday"
-
case ["Sunday", "Saturday"]: out 3
case else out 4
```

Execute different code depending on window name

```
str s.getwintext(win("Quick Macros"))
sel s 3 ;;case insensitive, use wildcards
case ["*[Macro97]", "*[Macro98]"]: out 1 ;;window name ends with [Macro97] or [Macro98]
case else out 2
```

Repeat

Syntax

```
rep [n]  
  (tab) statements  
  (tab) . . .
```

Can be single line:

```
rep ( [n] ) statements
```

Parameters

n - number of times to repeat. If omitted, repeats forever.

Remarks

Repeatedly executes **statements**.

Use [break](#) to exit loop. Use [continue](#) to skip following **statements**.

See also : [for \(127\)](#), [foreach \(128\)](#).

Examples

```
rep 10  
  _bee  
  _1  
  
rep(10) bee; 1  
  
rep  
  i+1  
  if(i>10) break
```

for (repeat with counter variable)

Syntax

```
for counter initvalue finalvalue [step]
(tab) statements
(tab) ...
```

Can be single line:

```
for(counter initvalue finalvalue [step]) statements
```

Parameters

counter - counter variable. Type - int, long, lpstr or pointer.
initvalue - initial **counter** value. Type - type of **counter**.
finalvalue - final **counter** value. Type - type of **counter**.
step - value to add to **counter** after each loop. Type - int. Default: 1.

Remarks

Simple description:

Repeatedly executes **statements**, after each loop adding 1 (or **step**) to **counter** variable.

Complete description:

Assigns **initvalue** to **counter**, and, while **counter** < **finalvalue**, repeatedly executes **statements**. After each loop adds **step** to **counter**. If **step** is negative, executes **statements** while **counter** > **finalvalue**.

Use [break](#) to exit loop. Use [continue](#) to skip following **statements**.

Tips

If you don't need counter variable, use [rep \(126\)](#).

See also: [foreach \(128\)](#)

Examples

```
lpstr s = "abcd.ef"
int i
int j = len(s)
for i 0 j
    out i
    if s[i] = '.'
        break

for(i 0 j) out i; if(s[i] = '.') break
```

Repeat for each item in collection

Syntax

```
foreach item coll [function] [arguments]
(tab) statements
(tab) ...
```

Can be single line:

```
foreach(item coll [function] [arguments]) statements
```

Parameters

item - variable that in each loop receives next item from **coll** collection. Type depends on other parameters.

coll - usually a collection of some kind.

function - a user-defined function that populates the **item** variable.

arguments - additional arguments for **function**.

Remarks

foreach is similar to [rep \(126\)](#) and [for \(127\)](#). It simplifies enumeration of items in collections of various kinds. Executes **statements** for each item in the collection.

Currently there are two predefined kinds of collections that may be used with **foreach**. Used without **function**.

1. *For each line in string.*

If **coll** is string, in each loop is extracted next line and stored into **item** variable. The variable must also be string (str or lpstr).

2. *For each item in COM collection.*

If **coll** is a COM collection interface, in each loop is extracted next item and stored into **item** variable. Type of the variable must match type of collection elements, or can be VARIANT. The COM interface should be defined in a type library.

There is possibility to extend **foreach**. For example, enumerate windows, files, etc. [Read more](#).

For this purpose is used a user-defined function (**function**). **foreach** executes the function at the beginning of each loop. The function gets next item from **coll** collection and assign it to **item** variable. Actually **coll** can be anything (string, array, variable of a user-defined type, or not used at all).

The function must have two or more parameters. The first two parameters receive **item** and **coll**. They can have any type that is appropriate for function's purpose, but the first parameter should be reference. Other optional parameters will receive **arguments**.

The function must return: 1 - an item extracted; 0 - no more items (break **foreach** loop). It also can return -1 to tell that last item is extracted.

Although the function is executed repeatedly, it retains the same local variables (including parameters) through whole **foreach**. For example, you can use a local variable for internal indexing or one-time initialization.

Example function FE_Char:

```
/
function int&character $s

  Gets characters from string.

int i
if (s=0 or s[i]=0) ret
character=s[i]
i+1
ret 1
```

Using it:

```
int c
foreach c "ABC" FE_Char
_out c
```

Use **break** to exit loop. Use **continue** to skip following **statements**.

foreach can be in other block (if, for, ...).

Examples

For each line:

```
str s
str lines="line1[]line2[] []line3"
foreach s lines
  if(!s.len) continue
  out s
```

For each COM object (enumerate environment variables):

```
Wsh.WshShell shell._create
VARIANT v
foreach v shell.Environment
  out v
```

Handle run-time errors

Syntax1 - on error, continue

`err` [\[+\]](#)

Syntax2 - on error, execute code and continue

`err` [\[+\]](#)
 (tab) statements
 (tab) ...

Can be in single line:

`err` [\[+\]](#) statements

Syntax3 - the beginning of code guarded by err+

`err`-

Parameters

Options:

Default	Handle errors of the preceding statement.
+	Handle errors of all preceding statements (or all preceding statements below <code>err</code> -).
-	Marks the beginning of the block of statements guarded by <code>err</code> +

Remarks

Use `err` to continue macro execution when a run-time error occurs.

When an error occurs:

- If the next statement is `err` or `err`+, execution continues.
- Else if there is `err`+ somewhere below, execution continues after it.
- Else if used [opt noerrorshere \(97\)](#), the error is passed to the caller function.
- Else the [thread \(49\)](#) ends.

Syntax2: On error executes **statements**. If no error, skips **statements**.

`err` handles errors generated by QM functions, user-defined functions (see [end \(131\)](#)), COM functions and the OS (exceptions). It does not handle some non-continuable errors, such as "noncompiled function", "failed constructor". It is not used to handle [compile-time errors \(175\)](#).

When an error occurs, special variable `_error` is filled with information about the error.

```
type QMERROR ~description code iid source place ~line
```

description - string that you can see in QM output when error is generated and not handled.

code - a numeric value that can be used to identify the error. Can be 0.

iid - [id of QM item \(107\)](#) where the error is generated.

source - one of: 0 - no error, 1 - compile-time error, 2 - run-time error generated by QM, 3 - run-time error generated by [end \(131\)](#), 4 - exception.

place - offset of the statement in macro text. In [sub-functions \(182\)](#) it is relative to parent text.

line - the statement (text). Not available in exe (empty).

`_error` has [thread scope \(142\)](#), that is, is accessible in all functions of that thread, including functions registered to run when thread ends ([atend \(103\)](#)). In a function registered by [atend](#), `_error` contains error info if the thread ended due to an error; it is empty (**source** is 0) if no error.

`err` handles errors generated in current macro/function only. It does not handle errors generated in user-defined functions that are called from current macro/function. However called functions can pass their errors to the caller. There are two ways: 1. Add [opt noerrorshere 1](#) somewhere at the beginning (new in QM 2.3.5). 2. Add `err+ end _error` at the end (you can also add more code before `end`).

To see possible errors, check menu Run -> Compiler Options -> Show possible run-time errors. Then, when you compile a macro (click Run or Compile button), it shows functions that may throw errors when the macro runs. It adds indicators in the code editor. You can Alt+click an indicator to show the errors in QM output. This feature helps you to decide where to use `err`. Does not show some errors.

- Exceptions.
- `ERR_POINTER` ("invalid pointer").
- Type mismatch errors in automatic conversion from/to OLE types (145).
- Errors specified in 'Show RT Errors Options' dialog.

See also: Common errors (48), opt err, opt noerrorshere, opt nowarningshere (97), error constants (144)

Examples

Close "Notepad" window; ignore possible error (on error don't end macro):

```
clo "Notepad"; err
```

Activate "Notepad" window; on error display error description:

```
act "Notepad"
err
_out _error.description
```

Handle errors of all preceding statements and generate them in caller:

```
err+ end _error
```

Loop example:

```
out
run "notepad.exe"
1
int i
for i 0 3
_out "activate Notepad"
_act "Notepad" ;;throws error if Notepad not found
_err ;;handles the error
_out _error.description
_continue
_out "close Notepad"
clo "Notepad"
```

If macro ends due to an error, log the error to file 'My Documents\My QM\qm log.txt':

Insert this at the beginning of each macro that needs error logging:

```
atend LogErrors
```

And create function LogErrors:

```
if(_error.source) ;;macro ended due to an error
_str macroname.getmacro(getopt(itemid 3) 1)
_str functionname.getmacro(_error.iid 1)
LogFile F"Macro {macroname}: error in {functionname}.[]Error description:
{_error.description}[]Error line: {_error.line}[]" 1
```

Return (exit) from function

Syntax

```
ret [expression]
```

Parameters

expression - any [expression \(244\)](#). Default: 0.

Remarks

Use this statement to exit from a function and optionally return a value. If function returns not through **ret**, the return value is 0.

When used in a macro, **ret** also ends the macro. To end macro from a function, use **end**.

You should not use **ret** to return strings or arrays. Read the "Functions" topic.

See also: [Functions \(149\)](#)

Example

```
ret 10
```

Generate run-time error

Syntax

`end` [`errorstring`] [`flags`] [`errorcode`]

Parameters

errorstring - error description. Can be string or variable of type [QMERROR \(129\)](#).

- Can contain QM output [tags \(245\)](#).
- Use [F-string \(138\)](#) to insert variables.

flags (247):

0-3	Where to generate error: <ul style="list-style-type: none"> • 0 (default) - in caller. If there is no direct caller (151) - here. • 1 - here. • 2 - in caller that is not class member function called for this variable. • 3 - this flag is obsolete. <div style="background-color: #f0f0f0; padding: 5px; margin-top: 5px;"> In caller of first function that is not in private folder. This flag does not work in exe. </div> <p>See also: (97)opt noerrorshere, nowarningshere</p>
4	Don't open macro. You also can set this globally in Options.
8	QM 2.3.3. Generate warning, not error.
16	QM 2.3.5. Append last dll error string. See str.dllerror (207) .
32	QM 2.3.6. Temporary warning. It will be shown withing 1 hour after editing the macro where it would be generated. In exe the warning will never be shown. Flag 8 is optional.

errorcode - dll error code to use with flag 16. If omitted or 0, calls [GetLastError](#).

Remarks

Generates run-time error, which ends [thread \(49\)](#) (running macro) if not handled.

By default, if `end` is used in a user-defined function, the error is generated in caller (macro/function that called the function).

Caller can handle errors with [err \(129\)](#) statement. Then it may want to know error code. It is `_error.code`. There are several ways to set error code when generating error:

- Use an [error constant \(144\)](#) as **errorstring**.
- Prepend code (some number) to **errorstring**. Use error codes ≥ 2000 .
- Set `_error.code` and `_error.description`, and use `_error` as **errorstring**.

QM 2.3.3. If used flag 8, generates warning instead of error. It does not end macro. Just displays warning message in QM output. Flags 0-3 are applied. To disable such warnings, use [opt nowarnings 1 \(97\)](#).

In [function help \(245\)](#) you can see errors that the function may generate. QM displays errors from `end` statements used in that function, or errors explicitly specified in [Errors:](#) line.

`end` without arguments (just `end`) ends thread without generating error. But usually it's better to use [ret \(130\)](#). In some cases it's better to use [shutdown -7 \(104\)](#). Don't use Windows API functions such as [ExitThread](#), because then QM cannot manage threads properly.

`end` immediately ends execution of current function and its callers. However [class destructor functions \(157\)](#) and functions registered with [atend \(103\)](#) will be executed.

Avoid `end` (except to generate warning (flag 8)) in callback functions (dialog procedures, COM event functions, etc), unless really necessary. Also, avoid unhandled run-time errors there. If the callback function is called by a dll function, the dll may not free its allocated memory, critical sections, etc; in some cases the dll function will not work properly next time (need to restart QM). If the callback function is called with [call \(132\)](#), the caller cannot handle the error with `err`. The error is generated in the callback function, not in its caller.

Don't use `end` (except to generate warning (flag 8)) in [class constructor, destructor and operator= functions \(157\)](#) and functions registered with [atend \(103\)](#). Also, avoid unhandled run-time errors there. In destructor and `atend` function it just ends the function, not thread. In constructor and `operator=` it always generates error that cannot be handled; if constructor

131. end

fails, destructor calling behavior is undefined. If destructor or attend function can fail, it should handle errors (use `err` if need) and use `end` with flag 8 (warning) if need. If constructor or operator= can fail, move the code to a member function that must be explicitly called (like `Class var.Init` or `var.Clone(var2)`).

`end` in a [special thread \(49\)](#) (QM thread or trigger filter function) just ends the callback function, not the thread.

See also: [ret \(130\)](#), [#ret \(181\)](#)

Examples

```
end "file not found"
end "2000 my error description" ;;use an error code
end ERR_FILE ;;use an error constant

str filevariable="Z:\file"
end F"{ERR_FILE}: '{filevariable}'"

run "abc"
err
__error.description + " abc"
end __error

end ;;end thread without generating error
```

Call function by address

Syntax

```
int call(address [a1 ...])
```

Parameters

address - function address. It can be a user-defined function, dll function, sub-function or other code. For user-defined functions, you can also use function's name as string, or QM item id.

a1 ... - arguments.

Remarks

Calls a function by address, not by name. It allows you to choose a function at run time. For example, you can use it to call a callback function. Use [operator & \(134\)](#) to get function address.

Returns called function's return value, which must be of type int.

Argument types and number must match exactly. They are not converted. Composite types (str, interface pointers, etc) cannot be passed by value. Variables passed by reference must be with operator &, like `&var`.

Callback functions should handle errors (use [err \(129\)](#) if need). On unhandled error the thread ends, even if the caller uses [err \(129\)](#) to handle errors.

See also: [IsValidCallback \(114\)](#).

Example

```
Function MyFunction:
function# int'i str&s
...
```

Call MyFunction in usual way:

```
str s="string"
int r = MyFunction(1 s)
```

Call myfunction with call:

```
str s="string"
int fa = &MyFunction
int r = call(fa 1 &s)
or
str fn = "MyFunction"
r = call(fn 1 &s)
or
int iid = qmitem("MyFunction")
r = call(iid 1 &s)
```

Operators

An operator is a special symbol used to perform assignment, arithmetic, comparison or other operation. Here are listed binary operators, that is, operators that are used with two operands, like `a+b`. See also: [unary operators \(134\)](#) (used with single operand), [member access operator \(155\)](#) . and [array element access operator \(146\)](#) `[]`.

Syntax1

```
result = operand1 operator operand2
```

For example, `a = b + 1` means "add b and 1, and assign the result to a". Here b is **operand1**, + is **operator**, 1 is **operand2**.

Syntax2

```
operand1 operator operand2
```

Here **operand1** receives result. For example, `a + 1` means "add 1 to a" (the same as `a = a + 1`).

Parameters

operands - any [expressions \(244\)](#) (variables, constants, functions, expressions with operators).

operator - one of symbols listed below.

Assignment

To assign something to a variable, use `=`. Example: `a = 1`

When used with `if` or in other assignment, `=` is interpreted as comparison operator (see below).

Arithmetic and bitwise operators

Arithmetic	
+	add. Example: <code>a = b + 1</code>
-	subtract
*	multiply
/	divide
%	modulus

Bitwise	
&	AND
	OR
^	XOR
~	AND NOT
<<	left shift
>>	right shift (unsigned)

[Priority \(135\)](#) of all arithmetic and bitwise operators is equal. For example, in `a=b+c*d`, at first is calculated `b+c`, then multiplied by d.

Comparison operators

=	equal. Example: <code>if(a=5) out "a is 5"</code>
==	
!	not equal
!=	
<>	
>	more
>=	more or equal
<	less
<=	less or equal

Priority of comparison operators is lower than of arithmetic and bitwise operators. For example, `if(a < b+c)`, at first is calculated `b+c`, then compared with `a`.

Result of comparison operators is 1 (true) or 0 (false).

Logical operators

<code>and</code> &&	logical AND. Example: <code>if(a=b and c<d) out "a is equal to b, and c is less than d"</code>
<code>or</code> 	logical OR

Priority of logical operators is lower than of arithmetic, bitwise and comparison operators. For example, in `if(a>b and c!=d)`, at first are compared `a>b`, then `c!=d`.

Result of logical operators is 1 or 0. Operator `and` gives 1 if both operands are not 0. If `operand1` is 0, `operand2` is not evaluated. Operator `or` gives 1 if one of operands is not 0. If `operand1` is not 0, `operand2` is not evaluated.

Operators used with str variables

Comparison	
<code>=</code>	equal, case sensitive. Example: <code>if(s="abc") out "s is 'abc'"</code>
<code>!</code>	not equal, case sensitive
<code>~</code>	equal, case insensitive

Note that "" and null are not equal. Don't use `if(s="")` to check if `s` is empty. Instead use `if(empty(s))` or `if(!s.len)`. See also: [empty \(185\)](#), [StrCompare \(114\)](#).

Join (syntax2)	
<code>+</code>	append. Example: <code>str s; s="notepad"; s+".exe"</code>
<code>-</code>	append to the beginning

Operators `+` and `-` are not used with `syntax1` to join strings. Instead of `s1=s2+s3` use `s1.from(s2 s3)` or `s1=F"{s2}{s3}"`.

See also: [strings \(183\)](#)

Remarks

Only the assignment operator can be used with all variable types. Other operators can be used with QM intrinsic types (int, byte, word, long, double, str, lpstr). With other types, some operators cannot be used, or the result may be not as you need. Therefore usually it is better to convert to the intrinsic type that you need. Example: `BSTR b="xx"; str s=b; if(s="xx") ;;` don't use `if(b="xx")`.

Syntax1

Operands of type `str` are interpreted as `lpstr`, except with `str` comparison operators `=`, `!`, `~`.

`lpstr` and pointers are interpreted as numeric (unsigned int). For example `if(s1=s2)` compares pointers, but not the strings. Another example: in `s1=s2+1`, `s1` receives address of the second character of `s2` (if `s2` was "tree", `s1` will be "ree").

Syntax2

Like in C/C++, arithmetic and bitwise operators may be followed by `=`. Example: `a+=1`.

See also: [operator priority \(135\)](#) [expression type and precision \(136\)](#) [C run-time library \(120\)](#) [conversions, etc. \(265\)](#)

Examples

```
i = 5 ;;assign 5 to variable i
i + 2 ;;add 2 to variable i
a = b + 10 ;;calculate sum of b and 10, and assign it to variable a
```

133. Operators

```
a + b + 10 ;;calculate sum of b and 10, and add it to variable a
int j = i + 2 ;;declaration and assignment in the same statement
ave = i + j / 2 ;;add j to i, divide by 2, then assign result to variable ave
f = i - (10 * j) + Func(a b) * 10 ;;multiply 10 and j, extract result from i, call function Func and add its return
value, multiply by 10, then assign result to variable f
str s = "notepad" ;;declare variable s and assign string "notepad"
s + ".exe" ;;append ".exe"
if(i < 10 and s = "notepad.exe") i + j / 10 ;;if i is less than 10, and s is equal to "notepad.exe", calculate j
/ 10 and assign result to variable i
i = Func2(j s (i + 5) f) ;;call function Func2 and assign its return value to variable i; pass four arguments, one
of them is sum of i and 5
lef a-10 100 ;;left click; x is a-10, y is 100
```

Unary operators

Syntax

[+] [@] [!] [~] [-] [*] [*] [*] [&]operand

Example: `-10`. Here `-` is **operator**, `10` is **operand**.

Operators:

!	The logical-negation (logical-NOT) operator produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (0). The result has int type.
~	The bitwise-complement (bitwise-NOT) operator produces the bitwise complement of its operand. The result has the type of the operand.
-	The arithmetic-negation operator produces the negative of its operand. The result has the type of the operand.
*	The indirection operator accesses a value indirectly, through a pointer (147) . The operand must be a pointer value. The result is the value at the address to which operand points. The type of the result is the type that the operand addresses.
&	The address-of operator gives the address of its operand. The operand can be either a variable or a function name. The result is a pointer to (address of) the operand. If operand is non-initialized reference-variable, result is 0.
+	The type cast operator. Can be used in some assignment operations (=, ret , function arguments) to avoid "type mismatch" error. With interface pointers, it queries interface at run time. With lpstr and pointers, it simply passes raw value. It does not convert number to string; instead assign the number to a str variable and use it. It does not convert string to number; instead use function val .
@	QM 2.3.0. Converts string to Unicode (267) UTF-16. The operand must be a string (lpstr or str). The result is Unicode UTF-16 string. Use when calling dll and COM functions that accept UTF-16 strings. Parameter type usually is @* (word pointer; in C++ would be LPWSTR, OLECHAR* or similar), but also can be BSTR or int (for example with SendMessageW). Example (238) .

Remarks

Unary operators must be placed immediately before **operand**. With operators `!`, `~` and `-`, **operand** can be any [expression \(244\)](#). With operator `*`, **operand** can be any expression that has pointer type.

Unary operators are applied before [binary operators \(133\)](#), but after operators `.` (member access) and `[]` (array element access). If used more than one unary operator with the same operand, they must be in the order specified above, and are applied from right to left.

See also: [pointer and reference variables \(147\)](#), [set dll function address \(153\)](#)

Examples

```
a = -b
if(c > -10) a = Func(-10 ~ (d|e))
if(!win("Notepad")) out "Notepad window not found"
int* ip = &i
j = *ip
*ip = 10
POINT p; GetCursorPos(&p)
```

Operator priority

In QM, binary operators belong to 3 priority classes. At first are calculated arithmetic and bitwise operators, then comparison operators, then logical operators. Operators that belong to the same priority class are calculated from left to right. Parentheses can be used to change this order.

The two example statements below are equivalent (will be calculated in same order). In the second example, complex expression is broken into several single-operator expressions (*primary expressions*).

```
result = a+b*c < d+e and f > g*h or i = j
result = (((a+b)*c) < (d+e)) and (f > (g*h)) or (i = j)
```

[Unary operators \(134\)](#) are applied before [binary operators \(133\)](#), but after operators . (member access) and [] (array element access).

QM bug: expressions containing both `or` and `and` operators without enclosing are evaluated incorrectly. For example, `out (1 or 0 and 0)` is 1 (must be 0). Enclose parts, like `out ((1 or 0) and 0)`. For backward compatibility it is not fixed, but QM detects such expressions and shows warning.

Expression type and precision

Primary (single-operator) expression type and calculation precision depends on the operand that has higher precision class. There are 4 precision classes:

1. int (int, word, byte). In expressions, word and byte are implicitly converted to int.
2. unsigned int (pointers (including lpstr and interface pointers), unsigned int constants).
3. long (long).
4. double (double, OLE types).

The following are the exceptions:

- If **operator (133)** is logical or comparison, expression type is int (result 0 or 1).
- Else, if both operands are lpstr or pointer, and **operator** is -, expression type is int (difference between two pointers).
- Else, if **operand1** is lpstr or pointer, and **operand2** is int (int, byte or word) or unsigned int, expression type is type of **operand1**.
- Before calculating a bitwise operator, operands are converted to unsigned int or unsigned long.
- Before calculating the % operator (integer remainder), double operands are converted to int or long.

Type of more complex expression can be found by splitting it to primary expressions.

Numbers and strings

This topic is about literal constant numbers and strings that can be used in Quick Macros.

Numbers

Integer numbers can be in 3 formats:

1. Decimal. Examples: `47`, `-2000000000`.
2. Hexadecimal. Examples: `0x2F`, `0x88ca6c00`. Octal is not supported.
3. Character. Examples: `'a'`, `'.'`, `' '`. The value is its [character code \(239\)](#), which can be 1 to 255 in ANSI mode and 1 to 127 in [Unicode \(267\)](#) mode.

Numbers 0 to 2147483647 are of [type \(140\)](#) int. Numbers 2147483648 to 4294967295 are unsigned int. Bigger numbers are of type long. To change type can be used suffix letter I (int), U (unsigned int) or L (long). Example: `10L`.

Non-integer numbers are those that contain either decimal point (.) or exponent (E followed by a 1-3 digit number), or both. Examples: `5.12`, `0.975E-5`, `-2e10`. Must begin with a digit. The exponent specifies the magnitude of the number as a power of 10. For example, `1.5E2` is 150, and `1.5E-1` is 0.15. The type is double. Also called "floating-point".

Strings

Strings are enclosed in double quotes (quotation marks). Example: "notepad.exe". Type - lpstr.

Such strings cannot contain literal newlines and double quotes. Instead use escape sequences:

Escape sequence	Is replaced with
<code>[]</code>	Line break (carriage return+linefeed)
<code>' '</code> (two single quotes)	<code>"</code> (double quote)
<code>[digits]</code>	Character whose character code (239) is the number in <code>[]</code> . The number can be 1 to 255 in ANSI mode and 1 to 127 in Unicode (267) mode. For example, <code>[9]</code> is replaced with tab.

Example:

```
out "'name'[][65]"
Output:
"name"
A
```

If you need literal `[]`, to avoid replacing it to new line use `[91][]`. For `' '` use `[39]'`. For `[digits]` use `[91]digits`.

You can use the Text dialog to get properly escaped string. In the dialog you enter simple (nonescaped) text, and, when you click OK, it inserts escaped and enclosed string.

Escape sequences that are in a macro are replaced before the macro runs. Escape sequences are not replaced in text that your macro gets at run time (for example, from a file, clipboard, etc). If you need QM escape sequences at run time, use [str.escape \(210\)](#).

Some functions have own escape sequences. They are replaced at run time. For example, [str.format \(213\)](#) and other functions that support formatting, [str.timeformat \(236\)](#), regular expression functions.

Multiline strings

Multiline strings can be created using `[]`. Example: `"line1[]line2[]line3"`. Alternatively, you can place all text in other macro, and populate a str variable with text of that macro. Example: `str s.getmacro("big string")`.

A multiline string also can be created as a block of comments that is assigned to a lpstr or str variable. Example:

```
lpstr s=
```

```
multiline
string

out s
```

In such multiline strings, escape sequences are not replaced. For example, `[]` is not new line.

Make sure that there are no empty lines. An empty line terminates the multiline string, even if it is followed by more comments. To add new line characters to the end, add line containing just one space or semicolon.

QM 2.3.0. In the comments block, some or all lines can begin with tabs (before space). The tabs are removed. In previous versions, tabs are not allowed.

The variable must be in simplest form. For example, it can be `s` but cannot be `t.s` or `a[i]`. It must not be a member variable.

Unicode UTF-16 strings

Use operator `L` to create [Unicode \(267\)](#) UTF-16 constant strings. Example: `L"String"`. However all characters must be in range 1-255 (in ANSI mode) or 1-127 (in Unicode mode).

QM 2.3.0. You can instead use [operator @ \(134\)](#). It converts to UTF-16 at run time. Example: `@"String"`. It also can be used with variables.

Variables in strings

Add operator `F` before, and enclose variables in `{ }`. Such string is not constant.

```
strVariable=F"text{variable1}text{variable2}"
```

```
strVariable=
F
  text{variable1}text
  text{variable2}text
```

[More info \(138\)](#)

Or use string functions, for example [format \(213\)](#) or [from \(214\)](#).

Named constants

You can use [def \(139\)](#) to define named constants.

Strings with variables

QM 2.3.2.

Use operator `F` to insert variables and other expressions in strings. Enclose variables in `{ }`.

```
F"text{variable1}text{variable2}"
```

To define custom format, after variable insert [format field \(248\)](#) with two `%`.

```
F"text{variable1%%-20s}text"
```

Default format depends on variable type:

int, word, byte, pointers	%i. For 0x{variable} uses %X instead.
str, lpstr	%s
long	%I64i. For 0x{variable} uses I64X instead.
double	%.15G
Other types	Not supported.

Variables also can be used in [multiline strings \(137\)](#). Example:

```
str s=
F
    text{variable1}text
    text{variable2}text
```

Escape sequences are replaced only in text parts of quoted strings. As escape sequence for `{` can be used `{{`. The `{{` escape sequence also can be used in multiline strings.

QM 2.3.4. In variable fields can be used strings enclosed in ``` (grave accent). In multiline strings use `"` instead.

Strings with variables cannot be used in strings that must be constant, such as dll names and [case](#) strings.

Strings with variables, when used for macro names and file names, prevent automatic adding these macros and files to [exe \(51\)](#).

Examples

```
int i=50
str s="stringvar"
double d=3.1415926535897932384626433832795
str sv=F"variables: i={i}, s='{s}', d={d%.5G}, sum={i+d}, function={_s.from(`string`
i)}"
out sv ;;variables: i=50, s="stringvar", d=3.1416, sum=53.1415926535898, function=string50

also can use F"string" with key
key TT F"variables: 0x{i} {s%%20s} {d}" Y
```

See also: [str.format \(213\)](#), [string constants and escape sequences \(137\)](#), [str.escape \(210\)](#).

Define (242) named constant

Syntax

```
def name value
```

Parameters

name - [name \(257\)](#).

value - value. Can be a [constant value \(137\)](#) of any type (number, string), or any other [expression \(244\)](#). Can include nonconstant values (global variables, functions).

Remarks

Defines a named constant. A named constant is a meaningful name that takes the place of a number, string or other expression. It is similar to a variable, but cannot be modified. Named constants have global scope, i.e. can be used in any macro.

Named constants also can be defined in [reference files \(15\)](#) and [type libraries \(164\)](#). QM defines many Windows constants in WINAPI and other reference files. You can use them directly or with "WINAPI." prefix, like in this example:

```
out WINAPI.ERROR_BAD_PATHNAME
```

See also: [predefined/special constants and variables \(144\)](#), [declarations \(242\)](#), [scope \(257\)](#), [how to define constants at startup](#).

Examples

```
def MAX_PATH 260
def GMEM_SHARE 0x2000
def dblconst -1.25
def strconst "String constant"
def GPTR GMEM_FIXED|GMEM_ZEROINIT
def A10 A + 10
def IID_IShellFolder uuidof("{000214E6-0000-0000-C000-000000000046}")
```

Variables

Variables are used to temporarily store numbers, text and other data. The data can be changed. Variables can be passed to macro commands and functions as arguments.

Before using a variable, need to [declare \(141\)](#) it (except [predefined variables \(144\)](#)). Variable declaration includes type and name. The mostly used variable types are [int](#) (for numeric integer values), [str](#) (for strings) and [double](#) (for numeric floating-point values). Name can consist of characters A-Z, a-z, 0-9 and `_`, and cannot begin with a digit. In the following example, we declare variable `i` of type [int](#), variable `S3` of type [str](#), and variable `_a` of type [double](#). Then we assign some values:

```
int i; str S3; double _a
i=5
S3="text"
_a=15.477
```

QM has 7 intrinsic variable types: [byte](#), [word](#), [int](#), [long](#), [double](#), [lpstr](#), [str \(183\)](#). QM also has 7 [OLE-Automation variable types \(145\)](#) (including arrays) and defines several other types. More types can be defined with [type \(154\)](#), [class \(157\)](#) and [interface \(165\)](#) statements, or in [type libraries \(164\)](#). The table shows properties of intrinsic variable types, and how they are related to C++ and C# variable types:

Type	byte	word	int	long	double	lpstr	str
<i>Numeric</i>	Yes	Yes	Yes	Yes	Yes		
<i>String</i>						Yes	Yes
<i>Integer</i>	Yes	Yes	Yes	Yes			
<i>Signed</i>			Yes	Yes	Yes		
<i>Size (bytes)</i>	1	2	4	8	8	4	16 + string
Character (266)	!	@	#	%	^	\$	~
<i>Min</i>	0	0	-2147483648	-2 ⁶³			
<i>Max</i>	255	65535	2147483647	2 ⁶³ - 1			
C++	BYTE, char	WORD, short	int, long, DWORD, UINT, HWND, etc	__int64	double	LPSTR, char*	similar to CString
C#	byte	ushort	int	long	double		similar to string

The picture shows how variables of different types are stored in memory. Green squares represent bytes occupied by a variable of the specified type.



See also: [declaration \(141\)](#) [storage and scope \(142\)](#) [more types of storage \(143\)](#) [predefined variables \(144\)](#) [user-defined types \(154\)](#) [strings \(183\)](#) [OLE types and other predefined types \(145\)](#) [safe arrays \(146\)](#) [conversions, etc. \(265\)](#)

Usage: [expressions \(244\)](#) [operators \(133\)](#) [functions \(149\)](#) [strings \(183\)](#) [pointers \(147\)](#)

Declare (create) and initialize variables

Simplest syntax - create local variable with initial value = 0/empty

type var

Full syntax

type [scope] [ptr] [&] var [(expression)] [var2 [(expression)] ...] [= expression]

Parts

type - variable [type \(140\)](#) name.

var - variable [name \(257\)](#).

scope - variable [storage and scope \(142\)](#). Can be one of symbols listed below. Default: **var** is local.

+	var is global variable.
-	var is thread variable, shared by all functions of current thread (49) .
--	var is thread variable, private to current function.

ptr - up to three * characters used to declare [pointer \(147\)](#) variable.

& - declare [reference \(147\)](#) variable.

[\(244\)expression](#) - initial value. Default: 0 or empty.

Remarks

Creates one or several [variables \(140\)](#).

For ARRAY variables, also must be specified array type (type of elements): **ARRAY** (type) var

Member functions can be called inside declaration statement: type var.member(args)

Local variables can be declared inline, using syntax type 'var. See example with [for](#).

Examples

```
int a ;;declare local variable a of type int
byte b c d ;;declare 3 local variables b, c and d of type byte
int i = 5 ;;declare local variable i of type int; i=5
int e(10) f(20) ;;declare 2 local variables e and f of type int; e=10; f=20
int+ hwnd = win("Notepad") ;;declare global variable hwnd of type int; hwnd=win("Notepad")
double d1 ;;declare local variable d1 of type double
double d2 = d1 + (0.45 * i) ;;declare local variable d2 of type double; d2 = d1 + (0.45 * i)
str S1 = "Little Alien" ;;declare local variable S1 of type str; S1="Little Alien"
str S2.from("Little " "Alien") ;;declare local variable S2 of type str; S2.from("Little " "Alien")
str* sp = &S1 ;;declare local variable sp of type str* (pointer to variable of type str); sp=&S1 (sp is address of S1)
str& sr = &S2 ;;declare local variable sr of type str& (reference to variable of type str); sr is reference to (alias of) S2
str& sr = S2 ;;the same (& may be omitted)
str*& spr = &sp ;;declare local variable spr of type str*& spr is reference to sp
str** spp = &sp ;;declare local variable spp of type str** (pointer to pointer to variable of type str); spp=&sp (spp is address of sp)
str*** spps = &spp ;;declare local variable spps of type str*** (pointer to pointer to pointer to variable of type str); spps=&spp (spps is address of spp)
POINT point ;;declare local variable point of user-defined type POINT
function word'w int**i str&s ;;declare function parameters (local variables w, i and s)
for(int'j 0 len(S1)) out S1[j] ;;here j is declared inline. Same as int j; for(j 0 len(S1)) out S1[j]
inp str's ;;same as str s; inp s
ARRAY(str) a.create(10) ;;create local safe array of 10 elements of type str
Excel.Application b._create ;;create object of type Excel.Application (type Application is declared in Excel type library)
```

Variable storage and scope

Local (auto) variables

Local variables exist only in the function or macro where they are declared. You cannot simply access a variable declared in one function from another function. You can use the same variable name in different functions but it will not be the same variable.

Declaration example:

```
int i
```

Function [parameters \(152\)](#) also are local variables.

Local variables exist only as long as the function is executing. They are automatically created when entering the function and destroyed when leaving the function. Every executing function instance has its own set of local variables.

Local variables are located on the stack, in order they were declared. They are aligned on 4 bytes (variables of size 1 to 4 occupy 4 bytes, variables of size 5 to 8 occupy 8 bytes, etc). Variables of composite types (str, BSTR, ARRAY, VARIANT, interface pointers) can also have associated data in dynamic memory. Memory allocation is automatic.

Global (static) variables

Global variables are shared by all functions and macros of current process (QM or exe). Use global variables to share data between different macros or between different instances of the same macro.

To declare a global variable, use "+" like in this example:

```
int+ g_Variable
```

Global variables are created when [compiling \(47\)](#) the function or macro where they are declared, and destroyed when closing QM file, eg when QM exits.

The same global variable can be declared multiple times and in multiple places. If you assign a value in the declaration statement (eg `int+ g_v1=1`), the assignment is performed each time.

Don't use global variables where not necessary. Where possible, use local variables, thread variables, pass variables to functions as arguments.

To avoid confusion and conflicts when using multiple global variables, use descriptive names with some name convention, for example add "g_" prefix. To reduce the number of global variables, use [user-defined types \(154\)](#) to store several related variables in a single variable

If a global variable can be accessed by multiple [threads \(49\)](#) simultaneously, use [lock \(112\)](#) to prevent it, because it is dangerous, may give incorrect results, corrupt memory, make QM unstable.

If a macro runs in [separate process \(51\)](#), it has own global variables. Assume you use global variable g_x in 3 macros. Macro A runs in QM, macros B and C run in separate processes. You cannot use g_x to share data between these 3 macros, because each macro has its own instance of g_x. To share data between these macros you can use [registry \(106\)](#) or a file. Programmers can use Windows API functions, for example shared memory. QM has class `__SharedMemory` to make it simpler.

Thread variables

Thread scope is intermediate between local and global. Thread variables are shared between all functions of the same [thread \(49\)](#) (running macro). They must be declared in each function where used.

To declare a thread variable, use "-" like in this example:

```
int- t_i
```

Thread variables are created at run time, at the beginning of the function where first time declared in the thread. They are destroyed when the thread exits (macro ends).

If you assign a value in the declaration statement, the assignment is performed only the first time during thread execution (differently than local and global variables). Example:

```
int- t_i=5 ;;even if this line is encountered sevel times during thread execution, 5 is assigned only the first time
```

See also: [thread variables in special threads \(49\)](#)

Private thread variables

Variables declared with "--" are thread variables that are visible only in that function.

Declaration example:

```
int-- t_i
```

Member variables and functions

[Class \(157\)](#) members can be used in all member functions of that class. Instead of `variable.member` use just `member` or `this.member`.

Sub-function parent variables

[Sub-functions \(182\)](#) with v attribute can use local variables of parent.

Application variables (obsolete)

Local variables of the main function of a folder that has "[Application](#)" [property \(11\)](#) are also visible in other functions of that app folder.

Scope priority

If there are two or more visible variables or functions with the same name and different scope, QM uses the one that is the first in this list:

1. Predefined ([QM language keywords](#), [intrinsic functions \(52\)](#), [special variables/constants \(144\)](#), [intrinsic types \(140\)](#), [other QM-defined types \(145\)](#)). You cannot declare a variable (except class/type member) with the same name as one of predefined names.
2. Local and thread variables.
3. In [class \(157\)](#) member functions - members (variables and functions) of this class, then of base classes.
4. In sub-functions with v attribute - parent's local variables.
5. Application main function's variables (obsolete).
6. Global functions, variables, types, etc.

In class member functions:

- If a class member name matches a predefined name, to access the member use `this.member`.
- If a class member name matches a global name, to access the global use `not this.global`. Here `__not_this` is a [category \(159\)](#) that has been added in QM 2.4.3 for this purpose, but also can be used any category name.

See also: [more types of storage \(143\)](#), [predefined variables \(144\)](#)

Variable storage and scope (2)

Beside normal QM variables, you also can use other media to store data.

Environment variables

When QM starts, it receives a copy of system environment variables. You also can create and use your own environment variables. Your environment variables exist until QM exits (differently than normal global variables, that are destroyed when loading other file or reloading current file). Environment variables are strings.

To get/set/delete environment variables, use functions [GetEnvVar](#) and [SetEnvVar](#).

QM file commands, [dll](#), menus, toolbars, etc automatically [expand \(246\)](#) environment variables. See also: [str.expandpath \(231\)](#).

Environment variables usually are inherited from parent process, however not always.

Window properties

To associate some values with a window, you can use window properties. A window property is some numeric value (it can be int, pointer or lpstr), which has some name and can be set/retrieved/deleted using functions [SetProp](#), [GetProp](#) and [RemoveProp](#). Examples:

```
RECT* r
r._new
SetProp(hwnd "ra" r)
...
r+=GetProp(hwnd "ra")
...
RemoveProp(hwnd "ra")
r._delete
```

Call [RemoveProp](#) when destroying the window, for example on WM_NCDESTROY message. Read more about these functions in [MSDN library \(256\)](#).

Registry

You can use functions [\(106\)rget](#) and [rset](#) to save variables in the registry.

Memory in other process

You can use [__ProcessMemory](#) class to allocate, write and read memory in context of other process.

Dll variables

Some dlls export variables. You can declare such variable as dll function, but you cannot use it directly as variable. Instead, use it indirectly through reference or pointer variable. Example:

```
dll adll #_variable
int+& _var = &_variable
now _var can be used as dll variable _variable
```

Predefined/special variables and constants

Local variables

`_i` - variable of type int. It is normal local variable except that you don't have to declare it.

`_s` - variable of type str. The same as above.

`this` - reference to the variable for which current [member-function \(157\)](#) is called.

Thread variables

`_command` - command string of current [thread \(49\)](#). Type - lpstr. If thread started without command, it is 0. See also: [mac \(100\)](#), [QM command line](#).

`_hresult` - last called COM function's [error code \(168\)](#). Some other functions also use it.

`_error` - [error info \(129\)](#).

`_monitor` - monitor where various functions display dialogs, on-screen text, etc. You can change it to display them in certain monitor. Possible values:

- 0 (default) - depends on context. In most cases - primary monitor. If dialog owner window specified - owner's monitor. With [ShowDialog \(63\)](#), also depends on dialog style.
- 1-30 - monitor index.
- -1 - monitor from mouse.
- -2 - monitor from active window.
- -3 - primary monitor.
- The handle of a window whose monitor must be used.
- If invalid value (invalid index, handle, etc), is used the primary monitor.

`_monitor` is applied to these functions: [mes](#), [inp](#), [inpp](#), [OnScreenDisplay](#), [ListDialog](#), [ShowDialog \(63\)](#), and most functions that use them. Some other functions either have a **monitor** parameter instead, or a flag that tells to use `_monitor`.

Examples

```
display message box in monitor 2
_monitor=2
mes "text"

display on-screen text in monitor with the mouse pointer, and restore the variable
_i=_monitor; _monitor=-1
OnScreenDisplay "text"
_monitor=_i
```

Most of these variables normally are populated by QM, but you also can change some of them.

Also there are 10 int variables `tls0` ... `tls9`, for backward compatibility.

Global variables

`_winnt` - Windows major version. Value:

5 - XP/2003

6 - Vista/7/8

10 - Windows 10

`_winver` - Windows version. Value:

0x501 - XP (5.1)

0x502 - 2003 (5.2)

0x600 - Vista (6.0)

0x601 - Windows 7 (6.1)

0x602 - Windows 8 (6.2)

0x603 - Windows 8.1 (6.3)

0xA00 - Windows 10 (10.0)

`_iever` - Internet Explorer version. Value:

0x600 - IE 6

0x700 - IE 7

0x800 - IE 8

and so on.

These values are in hexadecimal format. To display a value in hexadecimal format, use code like this:

```
out "0x%X" _winver
```

`_win64` (QM 2.2.0) - 0 on 32-bit Windows, 1 on 64-bit Windows. Note: QM is 32-bit, but runs on 64-bit Windows too. See also: [IsWindow64Bit \(114\)](#).

`_unicode` (QM 2.3.0) - [Unicode \(267\)](#) mode. Nonzero if QM is running in Unicode mode (checked Options -> General -> Text: Unicode). It is equal to the default code page that is used with [str.ansi and str.unicode \(238\)](#), i.e. CP_ACP (0) in ANSI mode or CP_UTF8 (65001) in Unicode mode.

`_portable` (QM 2.3.5) - 1 if QM is running as [portable \(259\)](#) app, 0 if not.

`_hwndqm` - QM main window handle. In [exe \(51\)](#) it is handle of a hidden window that is created in the primary thread of the process (not the thread that executes the macro).

`_hinst` - program module handle (HMODULE or HINSTANCE).

- If the macro runs in QM, it is qm.exe. In [exe \(51\)](#) it is the exe file.
- Can be used with [CreateWindowEx](#) and some other API functions.
- With API resource functions instead use [GetExeResHandle \(114\)](#).

`_logfile` - default log file used by [LogFile](#). Default is "qm log.txt" in My Documents\My QM. In exe, default is "exename log.txt" in program's folder. You can change it.

`_logfilesize` - maximal size of log file used by [LogFile](#). Default is 1 MB. You can change it.

`_qmdir` - QM path without filename. Ends with \. Same as expanded "\$qm\$\". In exe it is exe path. To get path with filename, use [ExeFullPath](#).

`_qmver_str` - QM version string like "2.1.5". See also QMVER constant, below.

QM 2.3.2: All the global predefined variables except `_logfile` and `_logfilesize` are read-only.

Constants

QMVER - Quick Macros version. For example, 0x2010506 is 2.1.5.6.

EXE - 1 if compiling [exe \(51\)](#), 2 if qmm (macro that runs in separate process but not as separate exe file), 0 if the macro runs in QM.

WINNT, WINVER, IEVER - obsolete, instead use variables `_winnt`, `_winver` and `_iever` (see above). In macros running in QM these constants are the same as the variables. However in macros compiled to [exe \(51\)](#), the variables contain values for the computer where the program is running, whereas the constants have values for the computer where the program was compiled. Therefore it's better to use the variables, not constants. Also, in exe it's better to use them with `if`, not `#if`.

QM 2.3.3. QM defines constants for QM run-time errors. Names begin with `ERR_` or `ERRC_`.

- The `ERR_` constants are defined as strings containing error code, like `def ERR_FILECOPY "502 cannot copy file"`. You can use them with [end \(131\)](#), like `end ERR_FILECOPY`.
- The `ERRC_` constants are defined as numeric error codes, like `def ERR_FILECOPY 502`. You can use them with [err \(129\)](#), like `if (error.code=ERR_FILECOPY)`.

The error codes are in range 500-1999.

OLE Automation variable types and other types defined by QM

See also: [intrinsic types \(140\)](#)

QM supports 7 OLE-Automation types: [FLOAT](#), [CURRENCY](#), [DECIMAL](#), [VARIANT \(170\)](#), [BSTR \(171\)](#), [DATE \(93\)](#) and [ARRAY \(146\)](#). The first 6 types are mostly used with COM functions. Actually they are user-defined types but in most cases are interpreted like intrinsic types. For example, QM automatically [converts \(265\)](#) between intrinsic types and OLE-Automation types when necessary. However, only few QM intrinsic functions support these types. Other functions convert them to intrinsic types. In expressions with operators, OLE-Automation types are converted to double. For better precision, use [member functions \(170\)](#).

The table shows properties of OLE-Automation types and how they are related to C++ and C# variable types.

Type	FLOAT	CURRENCY	DECIMAL	VARIANT	BSTR	DATE	ARRAY
<i>Numeric</i>	Yes	Yes	Yes	*		Yes	
<i>String</i>				*	Yes		
<i>Can be x.x</i>	Yes	Yes	Yes	*		Yes	
<i>Signed</i>	Yes	Yes	Yes	*		Yes	
<i>Size (bytes)</i>	4	8	16 (2 unused)	16**	4**	8	4**
<i>Precision</i>	< double	~ double	> double	*			
C++	float	CY	DECIMAL	VARIANT	BSTR	DATE, double	SAFEARRAY*
C#	float		decimal	object	string		arrays

* A variable of VARIANT type can contain value of various types.

** A variable of BSTR type is pointer (4 bytes) to length-prefixed Unicode UTF-16 (double-byte) string, so whole size is 4 + 4 + (num. characters * 2) + 2 bytes. A variable of type VARIANT also can contain BSTR or pointer to data of other type. A variable of type ARRAY is pointer (4 bytes) to array descriptor, which has pointer to array data. Size of array descriptor depends on array type and number of dimensions.

QM also defines several other types and interfaces: [QMITEM \(107\)](#), [FILTER \(40\)](#), [QMERROR \(129\)](#), [POINT \(55\)](#), [RECT \(87\)](#), [SYSTEMTIME](#), [FILETIME \(93\)](#), [FINDRX](#), [CHARRANGE \(202\)](#), [REPLACERX](#), [REPLACERXCB \(203\)](#), [CALLOUT \(201\)](#), [FINDWORDN \(190\)](#), [MES \(62\)](#), [UDTRIGGER \(39\)](#), [KEYEVENT \(56\)](#), [GUID \(110\)](#), [IUnknown](#), [IDispatch \(165\)](#), [IAccessible \(84\)](#) and several other.

Variable type ARRAY

An array is a variable consisting of any number of variables (elements) of same type that are accessed by index. You can use safe arrays (read in this topic), [pointer-based arrays \(147\)](#), [embedded arrays \(154\)](#), [string map \(115\)](#) and [CSV \(116\)](#).

A variable of type ARRAY is an OLE-Automation-compatible dynamic (resizable) multi-dimensional (1 to 10 dimensions) safe array.

When declaring a variable of type ARRAY, you also must specify the type of elements. Syntax:

```
ARRAY(type) var
```

type - any [type \(140\)](#) except ARRAY.
var - variable name.

To access an element, use syntax:

```
arr[i1 [i2 ...]]
```

arr - variable of type ARRAY.
i1 - element index in leftmost dimension (dimension 1).
i2 - element index in dimension 2, etc.

Error if the index is invalid (the element does not exist).

There are several functions and other ways to add elements. Read about it later in this topic.

Examples:

```
ARRAY(str) a ;;declare array of strings
a.create(10) ;;add 10 empty elements
a[0]="first element" ;;set first element
str s
s=a[0] ;;assign value of first element to a variable
out s

ARRAY(str) am.create(3 10) ;;2 dimensions, 3 elements in first dimension, 10 in second
int i j
for i 0 am.len(2) ;;enumerate dimension 2
  _for j 0 am.len(1) ;;enumerate dimension 1
    _am[j i].from(j " " i)
  _out am[j i]
```

To add one element to a 1-dimension array and assign a value, can be used empty []. Examples:

```
int i
ARRAY(int) a
for(i 0 10) a[]=i ;;adds and sets 1 element
for(i 0 a.len) out a[i]

ARRAY(str) b
for(i 0 10) b[].from("element " i) ;;adds and sets 1 element
for(i 0 b.len) out b[i]

ARRAY(str) c
for(i 0 10)
  _str& s=c[] ;;adds 1 element and gets reference to it
  _s.from("element " i)
  _s+" in array c"
for(i 0 c.len) out c[i]
```

```

ARRAY (POINT) d
for (i 0 10)
    POINT& p=d[] ;;adds 1 element and gets reference to it
    p.x=i; p.y=i*2
for (i 0 d.len)
    &p=d[i]
    out F"{p.x} {p.y}"

```

To populate a string array with multiple values, you can assign a multiline string. You also can assign array to a str variable. Examples:

```

ARRAY (str) a="zero[]one[]two"
int i
for (i 0 a.len) out "%i %s" i a[i]

str s=a
out s

```

ARRAY variables are freed automatically. If you need to explicitly remove all elements, assign 0 or call [redim](#) without arguments:

```

a=0
or
a.redim

```

More details

ARRAY can be used in user-defined types, but cannot be used in embedded arrays. Example:

```
type ABC int 'x' ARRAY (BSTR) 'a'
```

ARRAY can be passed to functions. When passing to a user-defined function, whole array is copied, unless it is passed by reference (address of ARRAY variable). Arrays are not copied when passing to other functions (not user-defined). COM functions usually accept SAFEARRAY*, which is the same as ARRAY, therefore ARRAY variable can be simply passed. DLL functions that accept arrays usually expect direct pointer to array data (not ARRAY or SAFEARRAY*). Then you can pass address of first element (e.g., `&arr[0]`).

A variable of type ARRAY actually is pointer (4 bytes) to array descriptor. The array descriptor (variable of type SAFEARRAY), along with other properties, holds the pointer to the actual data. All memory allocation is [managed \(265\)](#) automatically.

ARRAY is compatible with Visual Basic dynamic array. However, elements of multidimensional array are stored differently than in C or Pascal arrays. For example, `ARRAY (int) a.create (x y)` creates array that by storage order is equivalent to C array declared as `int a[y][x]`, that is, order of dimensions is reverse.

ARRAY type supports operator = (assign). When you assign one array to another, whole array is copied, including all associated data.

You can also assign string (str, lpstr or BSTR) to array of str or BSTR type. Result will be single-dimensional array of strings where each string is one line from source string. If you assign single-dimensional array of strings to str or BSTR variable, result will be string where every line is element of source array. If array is multidimensional, result will be string where all elements are joined.

Array also can be assigned to variable of type VARIANT. When VARIANT variable is destroyed, or type is changed, containing array is freed. If you use operator =, array is copied. If you use function attach on VARIANT variable, it stores array into that variable, and sets ARRAY variable to 0.

To declare a global variable or pointer to ARRAY, etc, use appropriate [characters \(141\)](#) after the closing parentheses. Examples:

```

ARRAY (str) as ;;array of str variables
ARRAY (int) + ai ;;global array of int variables
ARRAY (VARIANT) av ;;array of VARIANT variables. VARIANT can contain other data types, therefore we can say
that array element types can be different.
ARRAY (RECT) ar ;;array of RECT variables

```

```
ARRAY (RECT*) arp ;;array of pointers to RECT variables
ARRAY (int)* pai = &ai ;;pointer to array of int variables
```

Functions

Here **var** is variable of type ARRAY. If a return type is not specified, the function returns **var** itself.

```
var.create([n1] [n2 ...])
```

Creates and initializes array.

n1, n2, ... - number of elements in each dimension.

- Array can have 1 to 10 dimensions.
- Lower bound of each dimension is 0.

This function also can be used to resize array without preserving previous content.

If called without arguments, creates array descriptor but does not allocate memory for elements.

```
var.createlb(n1 lb1 [n2 lb2 ...])
```

Same as **create**, but allows you to specify lower bounds.

lb1, lb2, ... - lower bound (index of first element) of each dimension. Can be negative.

```
int var.redim([n] [lb])
```

Resizes rightmost dimension. If the array is not created, creates 1-dimension array.

n - new number of elements. Negative **n** will add **-n** elements.

lb - new lower bound. If **lb** is omitted, does not change it.

If array is growing, returns index of the first added element (rightmost dimension). If shrinking, the return value is undefined.

The function preserves elements that exist in both old and new arrays.

If called without arguments, frees the array and sets **var** to 0. Other way to free array - assign 0 (a=0).

```
int var.insert(index)
```

Inserts new empty element. If the array is not created, creates 1-dimension array.

index - element index. If it is index of last element + 1, adds to the end.

Returns **index**.

Supports multi-dimension arrays. Inserts in the rightmost dimension. For example, if the array has 2 dimensions, where the first used for columns and the second for rows, the function inserts new row.

Added in QM 2.3.2.

```
var.remove(index)
```

Deletes an element.

index - element index.

QM 2.3.2. Supports multi-dimension arrays. Removes from the rightmost dimension. For example, if the array has 2 dimensions, where the first used for columns and the second for rows, the function removes whole row.

Note: When removing elements in **for** loop, start from the end. Example:

```
ARRAY(str) a="a[]b[]c[]d[]e"
int i; for(i a.len-1 -1 -1) if(i&1) a.remove(i)
out a
```

```
int var.len([dim])
int var.lbound([dim])
int var.ubound([dim])
```

len returns the number of elements.

lbound returns index of the first element.

ubound returns index of the last element.

dim - 1-based dimension index. Default: rightmost dimension.

```
int var.ndim
```

Returns the number of dimensions.

```
var.sort([flags] [sortfunction] [param])
```

Sorts array elements.

flags (247):

1	Sort descending. This flag also is applied when using sort function.
2	Case insensitive.
4	QM 2.3.2. Linguistic. Uses StrCmp to compare strings.
8	QM 2.3.2. Number, linguistic, case insensitive. Uses StrCmpLogicalW to compare strings. It compares numbers in strings as number values, not as strings.
0x100 (flag)	QM 2.3.3. Date. Converts strings to DATE and compares. If cannot convert both strings, compares like without this flag.

Flags 2, 4 and 8 used with **str** and **lpstr** arrays, when **sortfunction** not used. If without flags 4 and 8, uses [StrCompare \(114\)](#) to compare strings.

sortfunction - name of a user-defined callback function. Required only if type of array elements is not **str**, **lpstr**, **int**, **byte**, **word**, **double**, **long**.

param - some value to pass to the sort function. Default: 0.

The callback function is used for custom sorting. It is called multiple times while sorting, and must compare two elements in the array.

A template is available in menu -> File -> New -> Templates.

The function must begin with:

```
function# param TYPE&a TYPE&b
```

param - param of **sort**.

- Can be declared as pointer or reference of any type. Then you can pass [address \(134\)](#) of a variable of that type to **sort** as **param**.

TYPE - type of array elements.

a, b - two array elements.

The callback function must compare **a** and **b**, and return:

- -1 if **a** must be placed before **b**.
- 1 if **a** must be placed after **b**.
- 0 if there is no difference.

Example:

```
sorts multiline string
str s="zero[]one[]TWO[]THREE"
ARRAY(str) a=s
a.sort(2)
s=a
out s
```

QM 2.3.2. Supports multi-dimension arrays. Sorts rightmost dimension.

When sorting multi-dimension array, compares first element for each indice of rightmost dimension. For example, if the array has 2 dimensions, where the first used for columns and the second for rows, the function compares the first element in each row. When using callback function, the function receives reference to that element. To compare other element, adjust references as in the example.

```
function# param str&a str&b

initially a and b are elements in column 0
&a=&a+(2*sizeof(a))
&b=&b+(2*sizeof(b))
now a and b are elements in column 2

...
```

`var.shuffle`

Shuffles (randomizes) array elements.

QM 2.3.2. Supports multi-dimension arrays. Shuffles rightmost dimension.

Notes

ARRAY functions are not thread-safe. Don't use a single variable in multiple threads simultaneously. It can damage data. If need to use in multiple threads, use [lock \(112\)](#).

Pointer, reference, array

Warning: incorrectly used pointers can cause instability and data corruption.

Tip: if you are looking for arrays, [safe arrays \(146\)](#) are often the best choice.

Pointer variables

A pointer is a variable that holds address of other variable. Use operator & to get address of a variable or function. Use operator * to access variable to which pointer points. Examples:

```

Declare variable v:
int v = 5
Declare pointer p:
int* p = &v
Now p points to (holds address of) v
int a = *p
Now variable a = 5. This is the same as a = v
*p = 10
Now variable v = 10. This is the same as v = 10

Pass address of variable r to dll function:
RECT r
GetWindowRect(win() &r)

Pass address of function TimerProc to dll function:
timeSetEvent(1000, 10, &TimerProc, 0, 0)

```

To declare a pointer variable, use type of variable to which it will point, and append *. For example, if you will use pointer p to hold address of a str variable, declaration must be `str* p`.

Maximal level of indirection is 3. For example, `int*** p` is OK, but `int**** p` is error.

A pointer variable is similar to an int variable. Like of int, the size is 4 bytes. Unlike int, it is interpreted as unsigned.

Operator * also can be used with enclosed expressions and functions that return pointer. Examples:

```

*(arr + 2) = 10
int i = *FunctionThatReturnsIntPtr

```

Arrays

A pointer can point to an array of variables. To access an array element, use operator []. Array indices begin with 0. If pointer points to array, it means that it points to the first variable in the array (the following two expressions are true: `arr == &arr[0]` and `*arr == arr[0]`). If you add size of variable to the pointer (for example `arr + sizeof(str)`), the pointer will point to the next variable in the array (`arr == &arr[1]` and `*arr == arr[1]`).

You can create array in several ways:

1. Dynamic array allocated with intrinsic [memory-allocation functions \(148\)](#). Use `_new` or `_resize` to allocate or resize array. Use `_delete` to free it. These functions properly construct-destruct elements of any type. Example:

```

str* arr._new(10)
arr[0] = "abc"
str s = arr[9]
...
arr._delete

```

2. Dynamic array allocated with other functions (`calloc`, `LocalAlloc`, etc). Example:

```

word* arr = calloc(10, sizeof(word))
...
free arr

```

3. Dynamic array in str variable. QM automatically frees such array when str variable goes out of scope. Example:

```
str s.all(10*sizeof(word) 0 0)
word* arr = +s
...
```

4. Declare n local variables. Example:

```
str s0 s1 s2 s3
str* arr = &s0
...
```

5. Define variable type with embedded array. Example:

```
type LPSTRARRAY50 $s[50]
LPSTRARRAY50 a
lpstr* arr = &a[0]
...
```

6. Use [safe array \(146\)](#). Then you don't have to worry about array initialization and freeing.

In case 2 and 3, constructors and destructors for array elements are not called. For arrays of composite types (e.g. str), you must explicitly initialize array memory (calloc does it), and clear each element before freeing array memory. Example:

```
str* arr = calloc(10 sizeof(str))
...
for(int i 0 10) arr[i].all
free arr
```

Reference variables

A reference variable holds address of other variable, but behaves syntactically like that variable. Usually it is initialized (receives address of other variable) when declaring. After that, it is used like normal variable. When various operations are performed with the reference variable, actually is modified the variable to which the reference variable points. Examples:

Declare variable v:

```
int v = 5
```

Declare reference r:

```
int& r = &v
```

Now r is reference to (holds address of) v

```
int a = r
```

Now variable a = 5. This is the same as a = v

```
r = 10
```

Now variable v = 10. This is the same as v = 10

References can be [re]initialized later:

```
int& r
int v2=100
&r = &v2 ;;now r is reference to v2
out r ;;100
```

When initializing reference variable, it must receive address of other variable. To get address, use operator &. However, when variable is of same type, operator & is optional. It is also true with function arguments. When this optimization is undesirable, use operator +. Example:

```
int v = 5
int& r = &v ;;OK, r is initialized with address of v
int& r = v ;;OK, r is initialized with address of v
int& r = +v ;;r is initialized with value of v
```

Usage

Reference variable usage is easier (does not require *), but pointer variable is more flexible (can be used arrays, etc). Pointers and references are mostly used as function parameters. Caller passes address of a variable. Then function can modify that variable. This is often used with str, ARRAY and other variables to avoid copying of all data.

If a parameter is declared as byte pointer (`byte*` or `!*`), can be passed pointer of any type, or string. This is similar to `void*` in C++.

If expression that you assign to a pointer variable (or parameter, etc) has different type, QM may generate error. Use [`operator + \(134\)`](#) to compile without error.

When using pointers, they always should point to an existing variable or other valid location in memory. Some functions also accept 0. Otherwise, macro will end with error message "Exception" or "Invalid pointer", or QM will exit or become unstable, or data will be corrupted. You should know life time of variables, because, when variable goes out of scope (is destroyed), pointer to it becomes invalid.

Memory allocation functions

Normally, memory for variables is allocated automatically. Alternatively, you can allocate variables in dynamic memory (heap) using these functions.

These functions can be used to allocate and free memory for variables of any types, including composite types (str, VARIANT, etc) and types that have constructors/destructors. They are similar to C++ operators [new](#) and [delete](#). To simply allocate memory also can be used [dll functions \(114\)](#) or [str variables \(205\)](#). See also [safe arrays \(146\)](#).

Syntax

```
p._new([count])
```

```
p._resize([count])
```

```
p._delete
```

```
int p._len
```

Parameters

p - pointer variable of any type.

count - number of elements. For [_new](#), default is 1. For [_resize](#), default is plus 1, and can be negative to add **-count** elements.

Remarks

[_new](#) allocates memory for single variable or array of type of **p**. All bytes are set to 0. If the type has constructor, it is called for each element. The function sets **p** to point to the first element. Returns reference to **p**. When the memory is not needed anymore, it must be freed using function [_delete](#). The function does not free previously allocated memory.

[_resize](#) resizes memory to hold more or less elements. Whole or part of previous content is preserved. If **count** is greater than previous number of elements, the added memory is initialized in same way as with [_new](#). If **count** is less, the deleted elements are freed in same way as with [_delete](#). Variable **p** must point to memory allocated with [_new](#) or [_resize](#), or be 0. The function sets **p** to point to the first element of resized and possibly moved memory block. Returns reference to **p**.

[_delete](#) frees memory and sets **p** to 0. If the type has destructor, it is called for each element. Elements of str and other composite types also are properly cleaned. Variable **p** must point to memory allocated with [_new](#) or [_resize](#), or be 0. The function returns reference to **p**.

[_len](#) returns number of elements in array. Variable **p** must point to memory allocated with [_new](#) or [_resize](#), or be 0.

Memory allocated using these functions is not automatically freed when the pointer variable is destroyed. You must use [_delete](#) to free it.

If a pointer variable uses these functions, it cannot use other memory-allocation functions, and vice versa.

Examples

```
str* p._new(10)
out p._len ;;10
p._resize(20)
out p._len ;;20
p._delete

MyClass* c._new
...
c._delete
```

Functions

A function is a named piece of code executed as a unit. It can receive several values (arguments) and return some value. For example, function [win](#) receives window name, finds the window and returns window handle. Beside predefined functions (QM intrinsic functions, dll and COM functions), you can create your own (user-defined) functions. Usually, you create a function when you want to have a piece of code that can be executed more than once, in any macro. Instead of placing the same code in each macro, you place it in a function, and then call the function by name from any macro.

See also: [Function as QM item type \(21\)](#), [function tips \(150\)](#), [declaration \(parameters etc\) \(152\)](#), [various ways of calling \(151\)](#).

Function call syntax

Global functions (intrinsic, user-defined and dll functions) are called using syntax

```
func ([a b ...])
```

Here [func](#) is function name; a, b and c are arguments.

If function's return value is not used, parentheses are optional. Function's return value can be assigned to a variable. Or, function can be an argument of another function or part of an expression with operators. Examples:

```
Func a b
variable = Func(a b)
Func2(a Func(b c))
d = e + Func(b c) / 10
```

Member functions of [str \(183\)](#), [OLE types \(145\)](#), [user-defined classes \(157\)](#) and [COM interfaces \(168\)](#) are called using syntax:

```
var.Func ([a b ...])
```

Here var is variable for which is called function Func. Type of var is type to which belongs function Func. For example, to use str functions, you declare str variable:

```
str s
s.format("%i %i" a b)
```

User-defined and dll functions also can be called [by address \(132\)](#).

QM intrinsic functions

[Reference \(52\)](#).

In editor QM intrinsic functions have [this color](#).

User-defined functions

A [user-defined function \(21\)](#) is a macro that can be called from other macros. In editor user-defined functions have [this color](#).

See also: [Function tips \(150\)](#), [sub-functions \(182\)](#).

To define function's type and parameters, use [function \(152\)](#). To return a value, use [ret \(130\)](#). You can get function address with [& operator \(134\)](#) and use it as callback function. You can also start a user-defined function like a macro (not from code). Functions can be recursive (call itself, directly or through other function). Every running function instance has its own local variables.

Below is shown how function is called and executed. Red lines - execution flow direction. Green lines - passing and returning values. The second parameter is declared as reference (&), therefore the function receives address of variable e and can modify its value.

Macro

Function "fileext"

```

Macro:
lpstr s1 = "c:\md\t.txt"
str s2
int d
d = fileext(s1 s2)
out "%i %s" d s2

Function "fileext":
function lpstr f str&e
int i=findchr(f '.\')
if(i>=0) e.get(f i)
else e = ""
ret i

```

User-defined member functions

[Class member functions \(157\)](#) are similar to other user-defined functions, but can be called only with a variable of that type. There are no other ways to execute member functions.

Dll functions

You can use Windows API and other dll functions. To [declare \(242\)](#) a dll function: [dll \(153\)](#). To find dll function help: type/click its name in editor, then press [F1 \(245\)](#) and search in [MSDN library \(256\)](#) or Internet.

COM functions

You also can use [COM functions \(162\)](#).

Tips and recommendations for user-defined functions

[Why and how to create functions](#)

[How to receive and return values](#)

[Passing and returning strings, arrays, etc](#)

[Local variables](#)

[The help section](#)

[Miscellaneous tips](#)

[Using passwords](#)

See also: [user-defined functions \(21\)](#), [about functions \(149\)](#), [declaration \(parameters etc\) \(152\)](#), [programming in QM \(45\)](#)

Why and how to create functions

A user-defined function is a macro that can be called from other macros. You create a function using menu File -> New -> New Function. You can place functions anywhere except in System folder. A function can be called from any macro/function/menu/toolbar. Also you can create [sub-functions \(182\)](#) in code of other macros/functions.

Sooner or later, you'll find that several your macros contain the same or similar code that performs the same operation. But you probably don't want to create or copy the same code again and again. You can create a function and place the code there. Then each macro could use the code by simply calling the function by name.

For example, you have two macros:

Macro1:

```
lpstr s1="c:\md\t.txt"
str s2
int i

get filename extension
i=findcr(s1 '.')
if(i>=0) s2.get(s1 i)
else s2=""

out "%i %s" i s2
```

Macro2:

```
lpstr a="d:\mm\fav.mp3"
str b
int c

get filename extension
c=findcr(a '.')
if(c>=0) b.get(a c)
else b=""

out "%i %s" c b
```

Both macros use the same code to get filename extension. Instead of using the same code in each macro, you can create a function and place the code there. Create new function using menu File -> New -> New Function, and give it some meaningful name, e.g. fileext.

Function fileext:

```
/
function lpstr'f str&e

int i=findcr(f '.')
if(i>=0) e.get(f i)
else e=""
ret i
```

Now you can use the function in macros:

Macro1:

```
lpstr s1="c:\md\t.txt"
str s2
int i

i=fileext(s1 s2)

out "%i %s" i s2
```

Macro2:

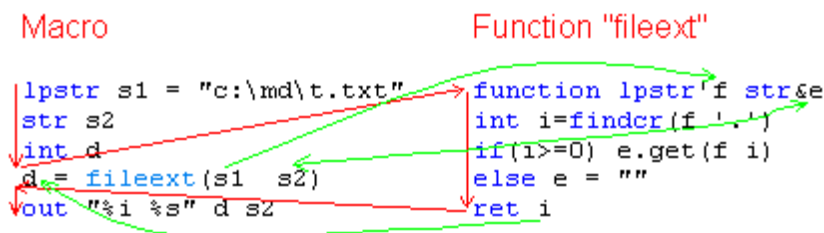
```
lpstr a="d:\mm\fav.mp3"
str b
int c

c=fileext(a b)

out "%i %s" c b
```

Now macros are not only shorter, but also easier to read (the function name says what the function does). Also, if you ever want to enhance the code that gets filename extension, you don't have to do it in all macros.

Below is shown how the function is called and executed. Red lines - execution flow direction. Green lines - passing and returning values. The second parameter is declared as reference (&), therefore the function receives address of variable e and can modify its value.



How to receive and return values

Here is an example of code at the beginning of a function:

```
/
function# a str's [str&so]
```

The `/` at the beginning means that this function can be called from code, but cannot be launched to run like macro. It can be followed by the name of a macro. Then, when you press the Run button, the macro would run instead of this function. You can set it in the Properties dialog.

The [function \(152\)](#) statement defines function's parameters and return type. It does not include function's name.

`#` tells that this function returns an integer value. You can use `'int'` instead. See [type declaration characters \(140\)](#).

This function has 3 parameters: `a`, `s` and `so`. The `[]` shows that `so` is optional.

Parameters are local variables. They are created and initialized each time when the function is called.

`a` declares parameter `a` of type `int` (integer). The declaration also can be `int'a` or `#a` (`#` is type declaration character for `int`), but for parameters of type `int` it is not necessary. When the function is called, `a` receives copy of the value (argument) passed by the caller.

`str's` declares parameter `s` of type `str` (string). Instead you can use `~s`. When the function is called, `s` receives copy of the value (argument) passed by the caller.

`str&so` declares parameter **so** that is reference to a variable of type `str`. When the function is called, **so** actually receives address of the variable passed by the caller. If the function modifies **so**, it actually modifies the caller's variable. Imagine that **so** is alias of the caller's variable. This method of passing arguments is called "by reference". It provides better performance, because string copying is eliminated. Also you can use this method to return one or more values. It is a good alternative to the `ret` statement, which copies and returns single value.

The `function` statement is optional. Use it when need parameters or when the function returns something (uses `ret`).

The `ret (130)` statement is used to exit function and continue to execute caller's code (caller waits while function runs). It also can return some value. It is optional. If function exits not through `ret`, or `ret` does not return a value, the function returns 0. See example below.

Passing and returning strings, arrays, etc

There are several ways to pass strings to functions. Each way has its advantages and disadvantages. Below are examples. For simplicity, the examples have only single parameter, although any number, type and order of parameters could be used.

1. Function begins with

```
function lpstr's ;;or function $s
```

Then you can pass any string. If you accidentally try to pass a number, you get error. I use this way in most my functions because it is fast and type-safe. If I need to use `str` functions with `s`, I assign `s` to a `str` variable inside the function. But if you don't know what is `lpstr` and how to use it, use way 2 or 3 instead. Although the function can modify the string, you should never do it (use way 3 instead), because the caller may pass a constant.

2. Function begins with

```
function str's ;;or function ~s
```

Then you can pass any string or number. Numbers are automatically converted. Advantages: easiest, does not require knowledge about `lpstr`; `s` can be manipulated using `str` functions without at first storing to a `str` variable. Disadvantages: whole string is copied, which is slower and in case of large strings requires much memory; not error if you accidentally pass a number.

3. Function begins with

```
function str&s ;;or function ~&s
```

Then you must pass a `str` variable. The function can modify the variable. Usually used to return string values (instead of using `ret`). An example is given above (function `fileext`). Advantages: fast and type-safe. Disadvantages: you cannot pass string constants (e.g. "some text") or variables of other than `str` type.

Also can be used `function str*s`, which is similar to 3 but requires knowledge about working with pointers.

To return strings, the function may begin with `function'str` or `function'lpstr`, and use `ret`. Although such functions are easier to use, but they are dangerous. You must understand variable scope, etc. Therefore, the preferred way to return string values is the way 3 described above.

Safe arrays (and not only `str` arrays) and user-defined types also usually are passed and returned like strings using the way 3, because it is fastest (does not copy whole data) and safest. Example function:

```
function ARRAY(str) &a
```

Passing an interface pointer using 3 is slightly faster, although passing it using 2 usually also is quite fast and does not copy whole object.

Local variables

Variables that you declare in a function are local to the function (unless you declare them as global or thread). They are destroyed when the function exits. Even if the function is called multiple times, its variables do not retain values between calls. Local variables are visible (can be used) only in that function. Caller's variables are not visible in the function. Parameters (used in `function` statement) also are local variables.

Read more: [variables \(140\)](#), [declaring variables \(141\)](#), [variable scope \(142\)](#)

The help section

[Read here \(245\)](#)

See example below.

Miscellaneous tips

Below is an example function. It does nothing useful, but shows something that can be useful in functions.

```
/
function# a str's [str&so]

What it does ...
Returns: ...

a - ...
s - ...
so - ...

REMARKS
...

EXAMPLE
str s
FunctionName 1 "abc" s
out s

opt noerrors here 1 ;;pass errors generated in this function to the caller. Other way to do it: err+ end _error (at the
end of this function).

spe -1 ;;set speed to be equal to caller's speed (initially function's speed is 0). Of course, this is not useful in functions
that don't have macro commands that are affected by spe.

if(a=0) end "invalid argument" ;;validate a: if a is 0, generate error. Error is generated in caller. If caller does
not use "err" statement to handle it, macro ends.

if(getopt(nargs) < 3) ;;if third argument is omitted
_ ... (more code)
_ret ;;exit function and return 0

if(&so) ;;if so is valid (caller may pass 0, or omit third argument)
_so.from(s a) ;;modify the variable that was passed by reference

... (more code)

wait -2 ;;autodelay (wait number of milliseconds equal to caller' speed). You may consider to add it at the end of some
functions.

ret 1 ;;exit function and return 1

here ret is not necessary (the function will return 0)
```

Using passwords

You should not pass passwords to nonsecure user-defined functions. You should use encrypted passwords. A user-defined function that accepts a password is secure only if it matches all these requirements:

1. Is encrypted.
2. Supports encrypted password. That is, contains code like this:

```
truepassword.decrypt(16 encryptedpassword "encryptionkey")
```

Here **truepassword** is a str variable, **encryptedpassword** is the password argument, **encryptionkey** is an encryption key that should be unique to this function. Read more about [str.decrypt and str.encrypt \(209\)](#).

To encrypt passwords, can be used [Options -> Security \(14\)](#) dialog or str.encrypt. When encrypting a password, QM extracts the encryption key from the function, and encrypts the password using the encryption key. An encrypted password has this format: [*XXXXXXXXXXXXXXXXXXXX*]. You can simply pass it to the function, like [Function\(a b](#)
[" \[*0123456789ABCDEF*\] "\)](#), or embed it in a string, like [Function\(a b](#)
["user=Me;password= \[*0123456789ABCDEF*\] ;"\)](#).

QM functions that accept password ([net](#), [AutoPassword](#), etc) support encrypted password.

3. Does not pass nonencrypted password to nonsecure user-defined functions.

4. Does not paste or type the password. Does not use anything that allows the password to be entered somewhere in visible form.

To enter a password into a password field of a certain program, use function [AutoPassword](#). It is easiest and most secure way to enter a password from a macro. It cannot enter the password in a non-password-field.

To make your macro that uses passwords really secure, you should also encrypt it and use [inpp \(61\)](#) to ask for password to run it. You can use the "Password input box" dialog for this.

Calling user-defined functions

[Functions \(149\)](#) can be called in several ways.

1. Directly.

Directly called functions run synchronously. The caller macro/function waits until the called function returns.

Examples:

```
functions
FileRename a b
x=GetAttr(c)

member functions
Tray t
t.AddIcon(a b)
```

A special case of directly called functions are functions used with [foreach \(128\)](#).

See also: [sub-functions \(182\)](#)

2. As callback function.

You give address of function A to function B. Then function B can call function A using the address. With some QM intrinsic functions instead of address can be used name or some other property.

Function A can be a user-defined function or a dll function, but cannot be a member, COM or QM intrinsic function. Function B can be any.

Function B can call function A immediately (eg [EnumWindows](#)), or set it to call later (eg [SetTimer](#)). COM [event \(169\)](#) functions also are callback functions. Also there are functions that run in special [threads \(49\)](#) (trigger filter functions and functions that run in QM main thread).

To call a user-defined function when you have its address or name, use [call \(132\)](#).

Examples:

```
EnumWindows &enum_proc 0

SetTimer hDlg 1 1000 &MyTimerProc

x=ShowDialog("Dialog55" &Dialog55)

int fa=&my_callback_func
call fa 10 20

str fn="my_callback_func"
call fn 10 20

atend my_atend_func ;;with some QM intrinsic functions must be used function name without &
```

3. Special class member functions (constructor, destructor and operator=) are called implicitly. They run synchronously.

4. [Thread \(49\)](#) entry functions. A thread entry function is a macro or function that has been launched using the Run button, a trigger, [mac \(100\)](#), or some other way if it caused to create new thread. Functions launched with [mac](#) run asynchronously. The macro/function that contains [mac](#) does not wait, unless it is explicitly programmed to wait.

Define function parameters and return type

Syntax

```
function [c] [functype] [parameters]
```

Parameters

[\(266\)](#)**functype** - return type. Default - int.

parameters - list of parameters.

- Parameter syntax: [\[type \(266\)\]name](#)
- Optional parameter syntax: [\[\[type \(266\)\]name \]](#)
- Default **type** is int.

c - `__cdecl` calling convention. Use for C callback functions. Default: `__stdcall`.

Remarks

Defines user-defined function's return type and parameters.

It must be the first statement in the function or [sub-function \(182\)](#). Only comments can be before.

This statement is optional if don't need parameters. If not used, function's return type is int.

Parameters are local variables. When calling the function, they receive values that are passed to the function (arguments).

Parameters enclosed in `[]` are optional. When calling the function, values of unused optional parameters will be 0.

If a parameter is declared as pointer or reference, you must pass address of a variable of that type. Then function can modify that variable. This also can be used to pass str variables, arrays, (to avoid copying of whole string or array) and variables of user-defined types. To get address of a variable, prepend operator `&`. If an parameter is declared as reference, operator `&` is optional. Some functions may also accept 0 for pointer/reference parameters (usually, for optional parameters).

Parameters of type `byte*` can receive pointers of any type, also str, lpstr and interface pointers of any type. For parameters of other types, must be passed values of same or compatible type (string to string, numeric to numeric, etc). If type casting is required, use [operator + \(134\)](#).

Functions can be called by other functions or macros, or as callback functions. If a function is started using some other way (Run button, trigger, etc), usually arguments are not passed, and values of all parameters are 0. Arguments can be passed with [mac \(100\)](#) and [command line](#).

See also: [about functions \(149\)](#), [Function as QM item type \(21\)](#), [function tips \(150\)](#), [various ways of calling \(151\)](#), [variables \(140\)](#), [pointers \(147\)](#).

Examples

Examples are given in pairs. The first example in a pair is function's text. The second one is how it can be called (code in another function or macro). Assume that function's name in all examples is "Func".

Example 1:

Function accepts 2 int parameters and returns value of type int.

```
function# a b
ret a+b/2
```

Call it. Pass values 10 and 20. Assign function's return value to variable i.

```
i = Func(10 20)
```

Example 2:

Function accepts 2 parameters (lpstr and str reference) and returns value of type int.

```
function# lpstr'sFile str&sr
if(!&sr) end "invalid argument"
sr.from("$Desktop\$\" sFile)
sr.searchpath()
ret s.len
```

Call it. Pass string constant and address of str variable. If function returns nonzero value then show s, else show error-string.

```
str s
if(Func("my file.txt" &s)) out s
else out "file not found"
```

Example 3:

Function accepts 2 parameters (byte pointer) and returns value of type int. Function is called using __cdecl calling convention.

```
function[c]# !*a !*b
if(a[0]>b[0]) ret 1
if(a[0]<b[0]) ret -1
```

If function returns not through ret, return value is 0.

Here we used type declaration character to define types of parameters. Same as function[c]# byte*a byte*b .

Here we don't call it directly, but call dll function qsort, which calls it as callback function.

```
str s="New York"
qsort s s.len 1 &Func
out s
```

Now s is "NYekorw" (sorted characters).

Declare (242) dll function

Syntax1 - declare single function

```
dll [-] dllfile [functype]Function [parameters]
```

Syntax2 - declare several functions from same dll

```
dll [-] dllfile
(tab) [functype]Function1 [parameters]
[ (tab) [functype]Function2 [parameters]
...]
```

Use this syntax to declare different name (alias) than it is in dll:

```
dll dllfile [TrueName] [functype]AnyName [parameters]
```

Use this syntax to find function by its ordinal number:

```
dll dllfile [ordinal] [functype]AnyName [parameters]
```

Parameters

dllfile - dll [filename \(246\)](#) or full path.

- Can be enclosed in quotes or not.
- Can be used named constants, but only if begin with `_DLL_` (QM 2.2.0).
- Can be "" if you'll explicitly set address with operator & (read below).

(266)functype - return type. If omitted, the return value is undefined (void).

Function - [name \(257\)](#).

parameters - list of parameters.

- Parameter syntax: `[type (266)]name`
- Optional parameter syntax: `[[type (266)]name]`
- Default **type** is int.

Options:

-	Delay-loading. Also use if you'll explicitly set address with operator & (read below).
---	--

Remarks

Declares a dll function.

By default, loads the dll file and finds the function when compiling (in [exe \(51\)](#) - when starting the program). If the dll or function not found, generates error (in exe - ends process with an error message box). If `dll-` used (delay-loading), the dll and function will be loaded at run time, when need (called, address queried, etc); will be run-time error (see [err \(129\)](#)) if the dll or function not found.

Some functions in dll have two versions: with 'A' and with 'W' suffix. The A version uses ANSI text. The W version uses Unicode UTF-16 text. You can omit 'A' suffix. If QM does not find the specified function, it searches for function with 'A' suffix. To support [Unicode \(267\)](#), use functions with 'W' suffix. [Example](#).

```
dll user32 #SetWindowTextW hWnd @*lpString
dll user32 #GetWindowTextW hWnd @*lpString nMaxCount

SetWindowTextW _hwndqm @"unicode text"

str s; BSTR b
GetWindowTextW(_hwndqm b.alloc(300) 300)
s.ansi(b)
out s
```

Supported are `__stdcall` and `__cdecl` calling conventions. Don't need to specify it.

QM 2.2.1. Can be used optional parameters. They are declared by enclosing into `[]`. The default value is 0. User-defined types passed by value cannot be optional.

QM 2.2.1. To declare variable number of parameters, add `...` at the end. When calling the function, argument types are not converted, therefore must match expected.

Allowed is function declaration without parameters. When calling such function, can be passed any number of arguments; argument types must match expected.

QM 2.4.1. You can declare parameters as [reference instead of pointer \(147\)](#). For example, `POINT&p` is the same as `POINT*p`. You can pass variables for reference and pointer parameters with or without operator `&`. For example, code `POINT p; GetCursorPos(p)` is the same as `POINT p; GetCursorPos(&p)`.

Dll functions also can be declared in [reference files \(15\)](#) and [type libraries \(164\)](#), which allows you to use them without declaring explicitly. Many declarations are in WINAPI and WINAPIV reference files. Usage example:

```
int hdc=WINAPI.CreateCompatibleDC(0)
```

Some Windows functions are declared by default, in the System\Declarations folder.

Usually you don't use full path in **dllfile**. Then the dll file should be in QM folder or in System32 folder. If used in exe, it should be in exe folder (however when creating exe it also must be in QM folder) or in System32 folder.

See also: [Functions \(149\)](#) [reference files \(160\)](#) [type libraries \(164\)](#) [declarations \(242\)](#) [scope \(257\)](#)

Examples

```
dll user32 #SendMessage hWnd wParam lParam
dll user32 #FindWindow $lpClassName $lpWindowName
dll user32 #GetCursorPos POINT*lpPoint
dll msvcrt
_ ^pow ^x ^y
_ [tolower] #ToLower c
_ [795] #ToUpper c
dll "qm.exe" ^Round ^number [cDec] ;;cDec is optional
dll msvcrt #sprintf $buffer $format ... ;;2 or more parameters
```

Explicitly set function address with operator & (QM 2.4.1)

Syntax

`&Function=address`

Parameters

Function - function name previously declared with **dll-**. You can declare any names.

address - any function address. The function can be a dll function, a user-defined function, a function compiled at run time with `__Tcc` class (C compiler), etc.

Remarks

Sets a function address to be used to call a function previously declared with **dll-**.

When you declare a dll function with **dll-**, by default QM automatically loads the dll and finds function at run time when the function is actually used first time. But sometimes you may want to just declare the function, and at run time explicitly get function address (eg with [GetProcAddress](#)) and bind it to the declaration. Then use operator `&`. Also it allows you to declare any functions with **dll-**, not just dll functions. The purpose is to be able to call the function as easily and safely as any declared dll function. You could instead use [call \(132\)](#), but then calling is not so easy and safe, because function parameters are not defined.

In function declaration (**dll-**), **dllfile** can be `""`. Operator `&` does not use it anyway.

Examples

```
declare a user-defined function as a dll function
dll- "" D1Test x ;;declare
```

```
&DllTest=&Function274 ;;set address
DllTest 100 ;;call
```

declare a dll function

```
dll- "" #MessageBox2 hWnd $lpText $lpCaption uType
&MessageBox2=GetProcAddress(GetModuleHandle("user32") "MessageBoxA")
MessageBox2 0 "text" "caption" 0
```

declare C functions compiled at run time

```
dll- ""
_test_add a b
_test_sub a b
__Tcc+ g_test_tcc
if !g_test_tcc.f
int* p=g_test_tcc.Compile("" "add[]sub")
&test_add=p[0]
&test_sub=p[1]
out test_add(4 5)
out test_sub(4 5)
#ret
int add(int a, int b){return a+b;}
int sub(int a, int b){return a-b;}
```

Define (242) variable type

Syntax (simplified)

```
type typename members
```

Members can be in the same line or/and in multiple lines:

```
type typename
(tab)member1
(tab)member2
(tab)...
```

Instead of the `type` keyword, you can use [class \(157\)](#) or [category \(159\)](#).

Parameters

typename - [name \(257\)](#) of the new type.

members - list of members.

- Member syntax: `[membertype (266)]membername`
- For members of type `int`, **membertype** can be omitted.

Remarks

QM has 7 intrinsic [variable \(140\)](#) types - `int`, `byte`, `word`, `long`, `double`, `lpstr` and `str`. It also defines [7 OLE types and several other types \(145\)](#). You can combine variables of several different types to create user-defined types (also known as structures or records). User-defined types are useful when you want to create a single variable that contains several related pieces of information. You create a user-defined type with the `type` statement. Once you have new type defined, you can create variables of that type (**typename**) like variables of intrinsic types. The type definition exists until QM exits or other file is opened.

A user-defined type can contain member variables of any type, including other user-defined types.

A user-defined type can contain embedded single-dimension arrays. The number of elements must be enclosed in square brackets and immediately follow member name: `[membertype]membername[numelem]`. Embedded strings (not `lpstr` or `str`) must be declared as byte array. Such array cannot be used directly as `lpstr` or pointer. Instead, use `&` operator to get address of array, and assign it to a `lpstr` or pointer variable. Example: `lpstr s = &var.array`. Array name, when used without `[]`, means first element.

It is possible to specify nonstandard member alignment, member offsets (e.g. to define unions), and define anonymous types within types. Read more [here \(156\)](#).

It is possible to use a base type ([inheritance \(157\)](#)).

It is possible to add global identifiers. Usually it is used to define a [category \(159\)](#). Syntax:

```
type typename [members] [: globals]
```

User-defined types also can be defined in [reference files \(15\)](#) and [type libraries \(164\)](#), which allows you to use them without defining explicitly. Many declarations are in WINAPI and WINAPIV reference files. Usage example:

```
WINAPI.PARAFORMAT p
```

Some Windows types are defined by default. Some of them are [defined by QM \(145\)](#), others in the System\Declarations folder.

Some Windows types actually are not user-defined types, but rather aliases of other types or of pointers to other types. For example, instead of various handle types (`HWND`, `HANDLE`, `HMODULE`, `HICON`, etc), in QM is used `int`. Instead of various pointer types (usually `LPTYPENAME`), use `TYPENAME*`. Instead of string types (`LPSTR`, etc), use `lpstr`.

QM 2.3.0. Allowed comments. See example. Also can contain lines containing only comments, but the comments must always begin with `;;`.

See also: [usage \(155\)](#) [unions \(156\)](#) [classes \(157\)](#) [categories \(159\)](#) [declarations \(242\)](#) [scope \(257\)](#)

Examples

```

type RECT left top right bottom
type PAINTSTRUCT hdc fErase RECT'rcPaint fRestore fIncUpd !r[32]
type MY @w !b ~s i[4] ~*sp RECT*rp[10]
type MY2 word'w byte'b str's int'i[4] str*sp RECT*rp[10]
type MY3 ;;the same as above, but in multiple lines
_word'w byte'b str's ;;the same as @w !b ~s
_int'i[4] ;;embedded array
_str*sp ;;pointer
_RECT*rp[10] ;;embedded array (10 pointers)
type ARR2 ARRAY(str)a ARRAY(int)b

```

User-defined type variable usage

To access a member of a variable of a user-defined type (UDT variable), use this syntax:

```
var.member
```

Here **var** is name of the variable of user-defined type, **member** is member's name. This syntax also is used when **var** is pointer.

To access the member of a single-member UDT variable, can by used only **var**. Exception is [OLE Automation types \(145\)](#).

User-defined types can contain other user-defined types. Assume that variable *a* has member *b*, and *b* has member *c*. To access *c* use `a.b.c`. If *b* is [base \(157\)](#) of *a*, *c* can be accessed using syntax `a.c`.

From a member function, members of variable for which the function is called can be accessed directly or using syntax `this.member`. Here **this** is special variable that can be used only in member functions.

UDT variables often are quite large. To avoid copying of whole variable when passing it to a function, functions usually accept pointer or reference. To pass a variable to a function that expects pointer, prepend the address-of operator `&`.

Examples

Declare variable of type RECT (RECT is defined as type RECT left top right bottom):

```
RECT r
```

Use members:

```
r.left = 100
```

```
r.right = r.left + 180
```

Copy r to rr:

```
RECT rr=r
```

Pass address of r to function:

```
GetWindowRect(win("Quick") &r)
```

Define type MY:

```
type MY word'w RECT'r byte'arr[32]
```

Declare variable m of type MY and use members:

```
MY m
```

```
m.r.left = 100
```

```
int i = m.arr[4]
```

```
lpstr s=&m.arr; out s
```

Pointers, references:

```
MY* mp = &m
```

```
i = mp.r.top
```

```
MY& mr = &m
```

```
i = mr.r.top
```

Type member alignment; unions

Structures

A *structure* is a type that has member variables. Most of these types are [user-defined types \(154\)](#).

Only these types don't have member variables:

byte - 1 byte.

word - 2 bytes.

int, lpstr, pointer, interface pointer - 4 bytes.

long, double - 8 bytes.

Default alignment

Structure members are placed in memory in the order they are defined. In the following example, offset of **a** is 0, offset of **b** is 4 (because size of **a** is 4), and size of structure is 8.

```
type T1 int'a int'b
T1 t
out "%i %i %i" &t.a-&t &t.b-&t sizeof(t)
```

Assuming that all members are not structures, these rules are applied:

- A member is aligned so that its offset is divisible by its size.
- The size of the structure is divisible by the size of its largest member.

For the above reason, in some cases there may be some padding (1-7 unused bytes) between members and at the end. In the following example, offset of **a** is 0, offset of **b** is 4 (3-byte padding after **a**), and size of structure is 12 (2-byte padding at the end).

```
type T2 byte'a int'b word'c
T2 t
out "%i %i %i %i" &t.a-&t &t.b-&t &t.c-&t sizeof(t)
```

Nonstandard alignment

Some types use nonstandard member alignment. In C/C++ languages, to set it is used `#pragma pack`. It can set alignment 1, 2, 4 or 8 (default). Structure members are aligned so that their offsets are divisible by their sizes or by the alignment value, whichever is smaller.

QM 2.3.2. To set alignment, add `[pack1]`, `[pack2]`, `[pack4]` or `[pack8]` (default) at the end of type definition. Example:

```
type T3 byte'a int'b word'c [pack1]
T3 t
out "%i %i %i %i" &t.a-&t &t.b-&t &t.c-&t sizeof(t)
```

Explicit alignment; unions

You can explicitly specify offsets of some members. It allows you to define unions (structures with overlapped members of different types) and structures with nonstandard member alignment. The member declaration must be preceded by the offset (integer number) in square brackets. Example:

```
type LWORD int'i [0]word'lo [2]word'hi
LWORD lh.i=0x00050007
out "LOWORD is %i, HIWORD is %i" lh.lo lh.hi
Output: LOWORD is 7, HIWORD is 5
```

If offset begins with +, it is interpreted as offset from offset of previous member. Empty brackets means same offset as of previous member. Example:

```
type LWORD int'i []word'lo [+2]word'hi
```

If offsets of all members are explicitly set, the size of the type also is explicitly set. In the following example it is 6 (normally it would be 8, because **i** would be at 4-byte offset):

```
type TYPE [0]word'w [2]int'i
```

QM 2.2.0. Implicit and explicit constructor/destructor/copy functions are not called for union members (except of the VARIANT type). A [warning \(176\)](#) is shown. QM cannot know which union member is valid at the time. Clearing wrong member may be catastrophic. These members should be cleared explicitly. For example, if an union member is an interface pointer, and you know it is valid at the time, call its hidden function Release to clear it. For BSTR, call SysFreeString. For ARRAY - SafeArrayDestroy. Also, there are special functions for some types, for example ReleaseStgMedium clears variables of STGMEDIUM type. All this is documented in [MSDN Library \(256\)](#).

Anonymous types within types

You probably already know that members of user-defined types can be other user-defined types. Example:

```
type T1 int'a int'b
type T2 int'x T1'y
T2 t
t.y.a=5
```

It is also possible to define types within types (QM 2.2.0). It can be used to simplify conversion between C++ and QM declarations, because in C++ it is used to define complex unions. These nested types don't have a type name and a member name, but can have specified offset. Their members are accessed directly. To define such nested types, enclose their members in {}. Examples:

```
type T2 int'x {int'a int'b}
T2 t
t.a=5



type LWORD int'i []{word'lo word'hi}
LWORD lh.i=0x00050007
out "LWORD is %i, HIWORD is %i" lh.lo lh.hi
```

Classes

Tip: To create new class, you can use menu File -> New -> New Class.

See also: [QM classes tutorial \(158\)](#)

Most programming languages have classes. A class is a user-defined type that also has member functions. To understand classes, you have to be familiar with variables, user-defined types and user-defined functions.

In QM, class definition and other features are the same as of user-defined [types \(154\)](#). The only difference is that you use keyword `class` instead of `type`, and it only changes icon in [popup lists \(46\)](#) from  to .

Member functions

To add a member function to a class, create new [QM item \(19\)](#) of type 'Member function' (menu File -> New -> New Member Function). Its name must be like "ClassName.FunctionName". See also [sub-functions \(182\)](#).

To call a member function, use syntax `var.FunctionName()`, where `var` is a variable of that type. Example:

```
class Class a b c ;;define class Class
Class var ;;create variable var of Class type
var.Func(arguments) ;;call Class member function Func (QM item name is Class.Func)
var.a=1; out var.a ;;access member variables
```

In a member function you can use keyword `this` to refer to the variable for which it is called. To access other member functions and variables of that class, in member functions you can use member name with or without "this." prefix. For example, to access member variable `c`, you can use `this.c` or `c`. To call other member function `Func2`, you can use `Func2(arguments)` or `this.Func2(arguments)`.

You cannot use a class (as well as a simple type) if macro containing [class definition \(154\)](#) is not compiled. To make a class always available, place its definition (or [#compile \(174\)](#)) into the `init2` function or some other function that runs at startup. You can create/delete/rename member functions at any time.

You can add member functions to classes, other user-defined types and even to QM intrinsic types (`str`, `int` etc). Cannot add to arrays and interfaces.

See also: [scope priority \(142\)](#) (when a member name matches a non-member name).

Special member functions

A class (only user-defined) can have three special functions - constructor, destructor and operator `=`. These functions cannot be called explicitly. They are recognized by QM item name.

	QM item name	Description
Constructor	Class	Is called implicitly when a variable of that class is created. Must not have parameters. Must not return a value.
Destructor	Class.	Is called implicitly before destroying a variable of that class (for example when a function containing a local variable of that class returns). Must not have parameters. Must not return a value.
Operator =	Class=	Is called implicitly when copying a variable of that class (<code>var1=var2</code> , <code>Function(var)</code> , etc). Must begin with <code>function ClassName&var</code> , where <code>var</code> is another variable of that class. The function should properly copy var members (<code>this.member=var.member...</code>), because QM does not copy members if the class has an operator= function. Must not return a value.

Here Class is class name. For example, if you want to add destructor to class `Abc`, create member function "Abc."

Constructor and destructor of global variables are executed in QM main thread. Constructor is called when compiling the macro where the global variable is declared first time. Destructor is called before unloading QM file, for example when QM exits.

Elements of ARRAY cannot have constructor.

In these functions don't use `end` (except to generate warning) and avoid unhandled run-time errors. [More info \(131\)](#).

Inheritance

New classes can be derived from existing classes (extend them). This feature is called *inheritance*, because the *derived class* inherits member functions and variables of its *base class* (an existing class). Member functions of derived class can access non-private members (variables and functions) of base class. Members of derived class override members of base class with same name. A derived class is defined by adding colon (:) before the first member variable which is of base class type.

Example:

```
class Base a b c
class Derived :Base'x c d e
Derived der
der.a=5 ;;same as der.x.a=5
der.c=5 ;;c of Derived
der.x.c=6 ;;c of Base
```

Members of base class can be accessed directly (e.g. `der.a`) or through the first member name (e.g. `der.x.a`). In member functions too: `this.a` or `this.x.a` or `a` or `x.a`.

QM 2.2.0. In derived class declaration can be used base type without member name. To access base members use the base type name. [Bug](#). Example:

If in declaration used only base type, member functions of the derived class cannot declare variables of base type, because the base type name in code is interpreted as member name, not as type name. For backward compatibility, the bug cannot be fixed.

```
class Base a b c
class Derived :Base c d e
Derived der
der.Base.c=6
```

Derived class, base classes and members can have constructors and destructors. When creating a variable of derived class, QM at first fills all memory with 0, then calls constructors for base classes, then for members (in the order as they are defined), then for derived class. Destructors are called in reverse order.

Public, protected, private and hidden members

Public members (functions and variables) can be accessed from anywhere. Private members can be accessed only from member functions of that class. Protected members can be accessed only from member functions of that class and classes derived from that class.

By default, class members are public. To make a member function protected or private, use the Properties dialog or Ctrl+right-click it in the list of QM items. To make a member variable protected or private, place one or two hyphens before it in class definition. Example:

```
class Base a b
class Derived :-Base'base --c d
Derived der
der.d=5 ;;OK: public member
der.c=5 ;;error: private member
der.b=5 ;;error: protected member
```

If member's name begins with `__` (two `_`), or function is in private folder, it is not shown in [the popup list of members \(46\)](#), unless you are editing a member function or use Ctrl+Shift+.

All [sub-functions \(182\)](#) can access private/protected members of class of parent function, even if they aren't member functions.

QM 2.4.3. QM items named like `Class_X` can access private and protected members of that class.

- Example 1: Function `Class1_Func` can access protected/private members of class `Class1`.
- Example 2: All member functions of class `Class1_X` can access protected/private members of class `Class1`, because

their QM item names begin with "Class1_".

- Example 3: Sub-function `Class1_X` (`#sub Class1_X`) can access protected/private members of class `Class1`.
- Such QM items cannot access protected/private members of base class(es) of that class.

Variables in dynamic memory

Local variables of class type are allocated on the stack (like all local variables). To allocate a variable in dynamic memory, declare pointer and call function `_new` (148). It allocates memory and calls constructor. The `_new` function also can be used to allocate an array. When the variable is no longer needed, call function `_delete`. It calls destructor and frees memory. Example:

```
MyClass* c._new  
...  
c._delete
```

Classes tutorial

Most programming languages have classes. A [class \(157\)](#) is a user-defined type that also has functions. To understand user-defined classes, you have to be familiar with variables, user-defined types and user-defined functions.

Creating and using a class

Note: To create new class you can use menu File -> New -> New Class. This tutorial does not use it.

Let's create a simple rectangle class. Class definition includes class name and member variables.

```
class CRect
double'm_width
double'm_height
```

Put it in some macro and compile or run the macro. It lets QM know about the new class. To make it always available, put it in a function that runs at startup. For example, in init2 (create it if does not exist).

Now create 3 member functions. To add member functions use menu File -> New -> New Member Function. The item name consists of class name, . and function name.

Member function **CRect.Init**

```
function double'width double'height

Initializes the object.

m_width=width
m_height=height
```

Member function **CRect.Area**

```
function'double

Calculates rectangle area.

ret m_width*m_height
```

Member function **CRect.Hypotenuse**

```
function'double

Calculates rectangle hypotenuse.
It is distance between two opposite corners.

ret _hypot(m_width m_height)
```

Now the class is created and you can use it anywhere. Declare variables of CRect type and call functions using syntax variable.Function(arguments). Example:

```
CRect r.Init(10 20)
out r.Area
out r.Hypotenuse

CRect r2.Init(11 19)
if(r2.Area>r.Area) out "r2 is bigger"
else out "r is bigger"
```

Class inheritance

You can create new classes that inherit member variables and member functions of existing classes. Let's create a class that inherits from CRect.

```
class CColorRect :CRect
    m_color
```

Member function CColorRect.Init

```
function double'width double'height color

this.CRect.Init(width height)
m_color=color
```

Member function CColorRect.GetColor

```
function#

ret m_color
```

Example:

```
CColorRect r3.Init(10 20 0xffff)
out "0x%X" r3.GetColor ;;call function of CColorRect
out r3.Area ;;call function inherited from CRect
```

Class member access control

By default, member variables and functions are public. They can be accessed like

```
CRect r
r.m_width=15
```

To protect member variables from accessing from outside of the class, in class definition add one or two hyphens before these members:

```
class CRect
    -double'm_width ;;m_width is protected. It can be used only in functions of CRect and inherited classes (eg
CColorRect).
    --double'm_height ;;m_height is private. It can be used only in functions of CRect class.
```

Now `r.m_width=15` would generate error.

You also can protect some member functions from calling from outside the class. You can do it in function's Properties dialog.

To hide member variables and functions without protecting, let the variable/function name begin with `__`. Examples: `__m_hidden`, `CRect.__Hidden`. Or place the functions in a folder that has 'private functions' checked in Folder Properties dialog.

Accessing the variable from a member function

Member functions always are called with a variable of that type. Example:

```
CRect r1 r2
...
r1.Func
r2.Func
```

If `Func` wants to access the variable (`r1`, `r2` or other), it can use *this*. It is a reference to the variable for which the function called. Example:

Member function CRect.Func

```
out this.m_width ;;same as out m_width
out this.Area ;;same as out Area
out &this ;;address of the variable
```

Special functions

A class can optionally have constructor, destructor and operator=. To add them, use item names like in the examples.

Constructor

Member function **CRect** (the name is the same as the class name)

```
out "This function is called when the variable is created."

m_width=1
m_height=1
```

Destructor

Member function **CRect.** (the name is the same as the class name, with . at the end)

```
out "This function is called before destroying the variable."
```

Operator =

Member function **CRect=** (the name is the same as the class name, with = at the end)

```
function CRect&source

out "This function is called when you assign one variable to another variable, both of
CRect type."

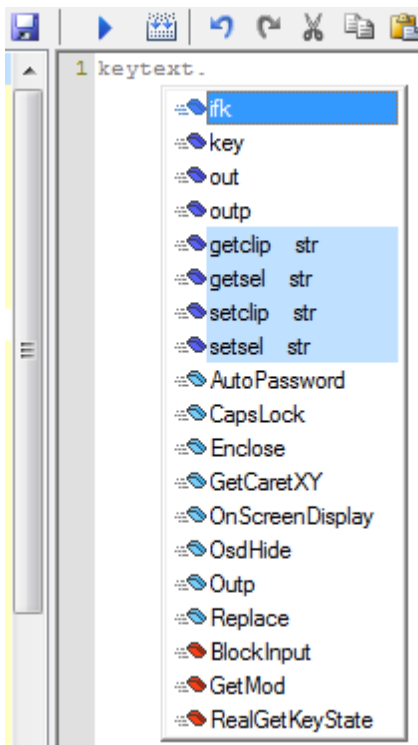
this.m_width=source.m_width
this.m_height=source.m_height
```

Test:

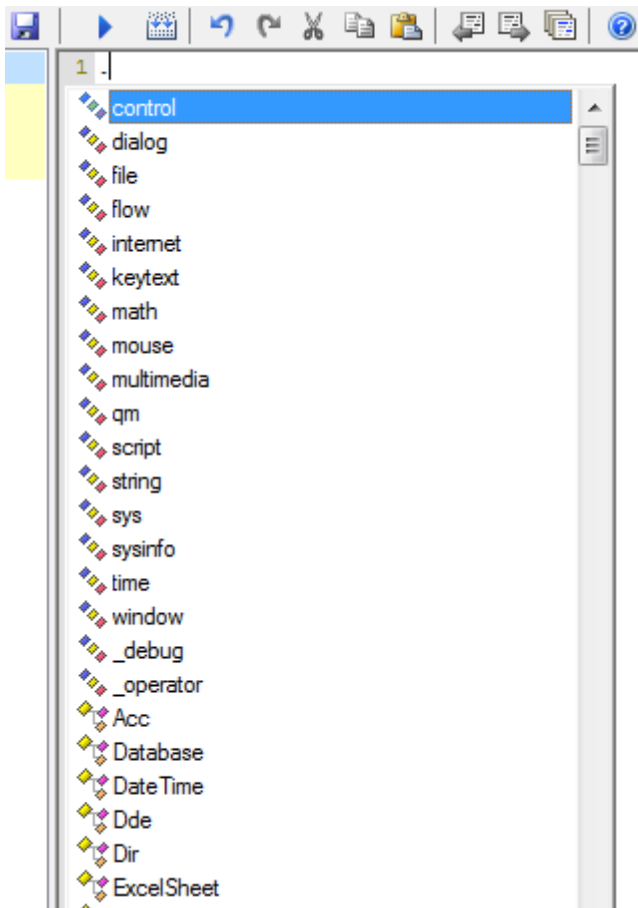
```
CRect r1 r2
r1.Init(5 5)
out r1.Area
out r2.Area
r2=r1
out r1.Area
out r2.Area
```

Categories

Related functions are placed into categories. You can use syntax `category.function` to access them, and see a list of functions when you type dot (.) after category name.



To view [list of categories \(46\)](#), type dot somewhere in macro text. Categories are at the top of the list, with  icon.



A category is a [user-defined type \(154\)](#) declared using special syntax. In declaration, instead of (or in addition to) normal member variables, you can add any global identifiers (names of functions, classes, etc) as members. Such identifiers aren't true members, but you can use `category.identifier` syntax to access them, and see them in function list. In declaration, names of global identifiers follow optional data members and colon. The colon must be separated by spaces. Instead of keyword `type` use keyword `category`. It just changes icon and placement in the [popup list of global identifiers \(46\)](#). Example:

```
define category files:
category files :
  _run del mkdir
  _SetCurDir GetCurDir

use functions from files category:
files.run "abc.exe" ;;same as run "abc.exe"
out files.GetCurDir ;;same as out GetCurDir
```

Syntax `category.identifier` is equivalent to simply `identifier`. The identifiers aren't strictly attached. Any identifier may follow category name and dot, and it will not be error. Category declaration can even include non-existing identifiers.

Category declaration can include other categories. One category can be derived from another category.

Category declaration can include dlls. Will be added all declared functions from them. Dll names must be enclosed in quotes. Example:

```
category mydll : "dll1" "dll2"
```

Category declaration can include QM folders. Will be added all functions and classes from them, except from private subfolders. Folder name or \path must be preceded by > and enclosed in quotes. Example:

```
category internet :
  ">\System\Functions\Internet" ">\User\Functions\Internet"
  _IeWait Ftp Http
```

Default categories are declared in \System\Declarations\Categories. They include folders from \System\Functions and \User\Functions folders. If you want to add your functions to the default categories, do the following: Open System\Declarations\Categories function, see what folders are defined for user functions, create such folders, and move your functions there. For example, to add a function to the default `internet` category, create folder User, folder Functions in it, folder Internet in it, and move the function there.

QM 2.3.2. Category declaration can include class members. Class name and list of its members must be preceded by ? and enclosed in quotes. If only class name given, adds all its functions. Also can add other strings, for example operators. Examples:

```
category test :
  "?str getclip getsel setclip setsel"
  "?MyClass"
  Func1 Func2
  "? (operator) + - * / and or"
```

See also: [declarations \(242\)](#) [scope \(257\)](#)

Declare (242) reference file or macro

Syntax1 - declare a reference file

```
ref refname file [flags]
```

Syntax2 - use previously declared file

```
ref refname
```

Parameters

refname - some [name \(257\)](#) that will be used to identify the reference file.

file - file or QM item (macro etc) that contains declarations.

flags (247):

1	Load on demand. Will be loaded only when actually needed.
2	Global scope. Identifiers from this file can be used anywhere without specifying refname .
4	Low priority. QM searches for declarations in low-priority refs after it does it in other refs.
8	Always show members in the main popup list (46) . Also, members are always declared automatically when they are first time used or viewed in editor. This flag is used together with flag 2. Flag 1 would not have effect.

Remarks

Syntax1

Declare file or macro where QM will look for declarations. Can be anywhere, not necessary in the same macro where used.

Syntax2

Makes identifiers from a somewhere declared reference file or macro available in current macro without specifying **refname**. Also, if the file is still not loaded, loads.

Reference files/macros can contain declarations of types ([type \(154\)](#), [class](#), [category](#)), constants ([def \(139\)](#)), dll functions ([dll \(153\)](#)), COM interfaces ([interface \(165\)](#)) and type libraries ([typelib \(164\)](#)). The declarations are in QM format. Each declaration starts from the beginning of a line.

The purpose is the same as of type libraries ([typelib \(164\)](#)) or [#compile \(174\)](#): declare various identifiers (constants, dll functions, types, etc) so that you don't have to do it when you want to use them. Unlike type libraries, reference files can be used only in QM. Unlike [#compile](#), [ref](#) does not compile whole file/macro. For example, if you compile a macro that contains declarations of dll functions, dll files are immediately loaded into memory. With [ref](#), dll is loaded only when (if) it is needed.

QM 2.2.0. A reference file/macro (lets call it *file1*) also can contain references to other reference files/macros (*file2*). In *file2* QM will look for declarations of identifiers that are used in declarations in *file1* but not declared in *file1*. To add such a reference, add `[ref refname]` line in *file1*. Also declare *file2* somewhere in macros, using [syntax1](#), before or after of *file1* declaration.

A declaration in a reference file/macro can be followed by comments in the next line. Comments will be displayed in QM status bar. The line must begin with single space and two semicolons. Declarations should not have comments in the same line.

Accessing identifiers declared in a reference file

If a reference file is declared with global scope (flag 2), identifiers from it can be used anywhere, without any special syntax. Otherwise use `refname.identifier`, unless the reference file is declared in the same macro/function. Also, when using `refname.identifier` syntax, identifier is automatically declared and colored when typing or opening macro.

See also: [sub-function attribute r \(182\)](#), [application folders \(11\)](#).

Existing reference files

Quick Macros comes with several reference files containing Windows API and other declarations:

WINAPI - mostly used Windows XP API declarations. It is declared with flag 2 (global scope) and therefore you can use Windows API simply, without prepending `WINAPI.`. That is why you can use Windows API without declaring them. That is

why when you type or click an identifier (type, function, etc), its declaration is displayed in QM status bar.

The file has been updated in QM 2.3.0. To create it was used Vista SDK v6.0 headers/libraries and preprocessor definitions for Windows 2003 and Internet Explorer 6 (WINVER 0x0502, _WIN32_IE 0x0603). The old file (QM 2.2.0-2.2.1) was created from older header files and therefore there are some changes. If you upgraded QM from an older version, you can still find the old file in QM folder. The old file is named winapiqm.txt. The new file is winapi.txt.

WINAPIV - declarations specific to Windows Vista. To avoid conflicts, it is declared not as global, and therefore functions and other identifiers can be accessed only like `WINAPIV.Function`. The file has been added in QM 2.3.0. To create it was used Vista SDK v6.0 headers/libraries and preprocessor definitions for Windows Vista and Internet Explorer 7 (WINVER 0x0600, _WIN32_IE 0x0700).

WINAPI7 - declarations specific to Windows 7. To avoid conflicts, it is declared not as global, and therefore functions and other identifiers can be accessed only like `WINAPIV.Function`. The file has been added in QM 2.3.1. To create it was used Windows 7 SDK v7.0 headers/libraries and preprocessor definitions for Windows 7 and Internet Explorer 8 (WINVER 0x0601, _WIN32_IE 0x0800).

WINAPI, WINAPIV and WINAPI7 contain declarations from [these header files](#).

```
#define NOCRYPT //don't add cryptography, 218 KB
#include "crt.h" //most header files for msvcrt.dll
#include <windows.h>
#include <winternl.h>
#include <winioctl.h> //QM 2.3.0
#include <shlobj.h> //includes <commctrl.h>
#include <shlwapi.h>
#include <richedit.h>
#include <olectl.h>
#include <oleacc.h>
#include <htmlhelp.h>
#include <pdh.h>
#include <tlhelp32.h>
#include <winsock2.h>
#include <wtsapi32.h> //QM 2.2.0.3
#include <mstask.h>
#include <msterr.h>
#include <dbt.h>
#include <psapi.h>
#include <uxtheme.h>
#include <wininet.h> //QM 2.3.0
```

You can [download](#) file WINAPI2 that contains more declarations.

Using with distributed macros

If you create macros for distributing, in them you should not use identifiers from reference files, unless you are sure that others have the same reference files. For example, WINAPI has been updated in QM 2.2.0 and in QM 2.3.0. If you use identifiers from it in your macros, other people that use other QM versions may get "unknown identifier" errors. You should rather extract these declarations from the reference file and place them in the macros. All this is not actual if the macros are compiled to [exe \(51\)](#).

Extracting and logging declarations

To extract a single declaration from a reference file or macro, type or click the identifier (e.g. function name) in the editor and do one of the following: 1. Ctrl+click the identifier and copy declaration from output. 2. Copy declaration from status bar, or press Alt+F8 to transfer it from status bar to output. 3. Press F2 and copy declaration from file. Also, to extract multiple declarations (except from macros), you can use menu Run -> Compile Options-> Show declarations from ref files. Then used declarations that are found in reference files are displayed in the output. A declaration is displayed when the identifier is first time used in a macro that you run or compile, or first time displayed in code when you use syntax `refname.identifier`.

Errors in declarations

Sometimes may happen that some declaration in reference file fails, for example, due to a missing dll or type library. In such case, file is opened, and statement highlighted. If you want that on error would be executed some user-defined function, place `#err function` in reference file, immediately after declaration. If function returns nonzero value, error is not shown, and compilation stops. If returns negative value, "unknown identifier" error also is not shown.

See also: [declarations \(242\)](#) [scope \(257\)](#) [#err \(175\)](#)

Examples

```
ref MyRef "MyRefFile.txt" 1
ref MyRef2 "MyRefMacro"
```

About COM

The Component Object Model (COM) is a standard for creating binary software components that can interact. It enables one applications (clients) to use capabilities of other applications and other software components (servers).

COM is the foundation technology for Microsoft's OLE (compound documents), ActiveX® (Internet-enabled components), as well as others. Automation (OLE automation) is COM-based technology that enables applications to provide objects in a consistent way to other applications, development tools, and macro languages. Part of Windows operating system features, that programmers can use, is based on COM, and cannot be accessed with Windows API ([dll \(153\)](#)) functions. Some applications also provide COM objects. For example, you can use Excel capabilities from QM. You can also use standalone ActiveX components that can be downloaded from the Internet.

In general, a COM object is made up of a set of data and the functions that manipulate the data. These function sets are called *interfaces*, and the functions of an interface are called *methods* and *properties*. These functions can be accessed only through a pointer to the interface (in QM known as *interface pointer* variable).

[COM support in QM \(162\)](#)

COM support in QM

See also: [about COM \(161\)](#), [using COM components \(163\)](#).

In QM you can:

1. Use [type libraries \(164\)](#). See available type libraries, declare them in code, and use identifiers (interfaces, types, constants, etc) declared in type libraries.
2. Explicitly [declare COM interfaces \(165\)](#).
3. Use [COM interface pointer variables \(166\)](#).
4. [Create COM object \(167\)](#).
5. [Call COM object functions \(168\)](#) (methods and properties).
6. Use early binding (VTBL or id), or late binding. By default, if possible, QM uses VTBL binding, which is fastest. You can use [#opt dispatch \(176\)](#) directive to use id binding, or [IDispatch](#) variable to use late binding (without declaring).
7. Enumerate [collections \(128\)](#).
8. Receive [events \(169\)](#).
9. Use ActiveX controls in [dialogs \(63\)](#).
10. Use [OLE Automation variable types \(145\)](#).
11. Use [type info \(46\)](#).
12. Execute VBScript code. Use [VbsExec](#) and other functions from System\Functions\Scripting folder.

QM does not support some features used in VB and other languages: default member, implicit application object, etc.

Using COM components

About COM components

COM components (also known as ActiveX components or ActiveX controls) provide various useful functions. For example, the Wsh (Windows Script Host) component provides many file, drive, network and other functions that are not available as intrinsic QM functions. In the web browser control, for example, you can display web pages in dialogs. Working with COM components requires minimum programming knowledge. You can simply select functions from the list, without even reading help files.

There are two main types of COM components - ActiveX controls, and non-control components. ActiveX controls provide graphical user interface and are used in dialogs. Other components are invisible but provide various useful functions. Controls are added to dialogs using the [Dialog Editor \(63\)](#). Non-controls are created using [_create \(167\)](#) or other functions.

Most COM components provide type libraries. Type libraries contain declarations of coclasses, interfaces and functions provided by these components, making available them to other programs, including QM.

See also: [COM support in QM \(162\)](#)

The 'COM Libraries (Type Libraries, ActiveX Controls)' dialog

Here you can see registered (installed) type libraries. You also can register new components and separate type libraries. The main purpose of the dialog - find a component's type library and insert [typelib \(164\)](#) declaration statement in current macro. Declaration is needed to use the component in macros.

- *Find* - search the list of type library names or control names. Selects next item that contains the above text.
- *Insert declaration from list* - insert type library declaration in current macro. What to do with it - read the "Using COM components" chapter below in this topic.
- *Use file path* - if checked, the declaration will use [syntax2 \(164\)](#). By default is used syntax1.
- *Insert declaration from file...* - locate a type library file and insert declaration statement in current macro. The library may or may not be in the list. This does not register the type library or component.
- *Register..., Unregister...* - register (install) or unregister (uninstall) a component or separate type library. Registers/unregisters only dll and ocx components. Registers/unregisters type libraries in any files that contain them.
- *Libraries* - view registered type libraries. The list includes type libraries for ActiveX controls and other COM components, and other type libraries.
- *Controls* - view registered ActiveX controls. When you select a control, is shown some info from its type library, if available. When you click Insert Declaration From List, is inserted its type library declaration.
- *Preview* - see how looks selected control. Not all controls will look as they look at run time because some features must be activated by calling their functions at run time (in macro).
- *Add control to dialog* - adds selected control to dialog. Available when opened from the dialog editor.

The path of the file that contains the selected type library is displayed at the bottom of the dialog. If there are separate type libraries for several languages, you can select language.

At the right side is displayed some information from the type library: name (although you can use any name), coclasses (types of COM objects) and interfaces (collections of functions that can be used with these COM objects).

Finding and installing COM components

Many COM components are installed and used by Windows and various applications. Some of them also can be useful in QM. Many other COM components are on the Internet.

It is not easy to find a good component on the Internet, especially if you want a freeware component. Many components are too simple to be useful, or have bugs, or are too big, or too expensive, or depend on various runtime files.

New (downloaded) COM components must be registered (installed). Some downloaded components have setup programs and are installed like any other software. Then you can find component's type library in the COM Libraries dialog. Other components came without a setup program, as an ocx or dll file in a zip file. To install a component that does not have a setup program, use the Register button in the COM Libraries dialog. It is possible to use COM components without registering; read next chapter.

Usually type libraries are embedded in component files (dll, ocx) and are registered together with component. If you have downloaded a separate type library (tlb, olb, dll), register it using the Register button.

If you move a component file to a different location, you have to Register it again (it will not be reregistered automatically).

Registered type libraries can be declared using GUID ([syntax1 \(164\)](#)) or file path ([syntax2](#)). Unregistered - only by path.

Distributing macros that use COM components

If you want to distribute macros that use a COM component, make sure that users have the component. Also, they must have compatible version. The macro will work if they have type library with same major version as in [typelib](#) statement, and same or higher minor version. This is actual even if you distribute [exe file \(51\)](#). You can distribute the component with your macros.

If you distribute a COM component with your macros, users or your setup macro/program must register it on that computer. COM components can be registered/unregistered using the COM Libraries dialog (see above) or function [RegisterComComponent \(114\)](#). Usually, when registering a component, it also registers its type library. To register .NET COM components, instead use regasm.exe; it is in .NET runtime folder; more info is on the internet.

It is possible to use COM components without registering. There are 2 ways:

- QM 2.3.4. Specify dll/ocx path with [_create \(167\)](#) or in [dialog definition \(63\)](#).
- QM 2.3.5. Use manifest and [__ComActivator](#) class.

The first way is easier, faster and works on all Windows versions, but does not work with .NET COM components and some other components. The second way works with all components, but requires at least Windows XP SP2.

Example of downloading automatically.

Components can be downloaded automatically, using function [DownloadComponent](#). Example:

```
typelib MSScript {0E59F1D2-1FBE-11D0-8FF2-00A0D10038BC} 1.0 0 1
#err MSScriptDownload
```

Function MSScriptDownload:

```
lpstr s="$system$\msscript.ocx"
if(FileExists(s)) ret
int fa=&RegisterComComponent
mac "DownloadComponent" "" s "http://www.quickmacros.com/com/msscript.zip" "Microsoft
Windows Script Control" 43 fa 6
ret -1
```

It runs at compile time. The typelib statement throws error if the COM component is not installed. The next statement handles the error and calls function that downloads and registers the component. However this cannot be used in exe; will need to somehow detect unavailable component at run time, for example if [_create](#) fails.

Using COM components

To use a component, its type library must be [declared \(164\)](#). The declaration statement can be in the same macro, or in other macro, e.g., in [init2](#) function or some other function that runs at startup (has "QM file loaded" trigger). Several type libraries are already declared by QM, and you can see them at the bottom of the list when you type dot.

When just evaluating new component, you can simply create new macro, insert the type library declaration statement (using the COM Libraries dialog), and run the macro, or just compile (Ctrl+Shift+R). Then you can type the library name in the next line, type dot, and see what classes are available. Classes are listed at the top of the list. When you double click a class in the list, its name is inserted in the macro. If it is an ActiveX control class, you usually can see ";;ActiveX control" in QM status bar. Some controls also can be used as non-control components, that is, created using [_create \(167\)](#), not in a dialog.

The following are two examples how new COM components are typically installed and used. Steps 1 and 2 are omitted if you have found the component on your computer (in the COM Libraries dialog).

Example1: installing and using a non-control component

1. Find it on the Internet, and download. When searching, you can include keywords "COM component", "ActiveX component", "ActiveX DLL", "ActiveX control".
2. Install the component using the setup program or the Register button in the COM Libraries dialog.
3. Insert type library declaration: In the COM Libraries dialog, find component's type library and click Insert Declaration From List. If you cannot find it in the list, try Insert Declaration From File.
4. Run or compile the macro where you have inserted the declaration ([typelib](#) ...). If you have placed the declaration in the [init2](#) function or some other function that runs at startup (has "QM file loaded" trigger), it is compiled automatically. Normally,

163. Using COM components

you don't have to somehow separately compile just to run the macro, but while creating the macro you cannot see the available classes and functions until the declaration is not compiled.

5. Type the library name (it is the first word after [typelib](#)). Then type dot, and double click a coclass (coclasses are at the top of the list). If there are several coclasses, pick one that have a meaningful name, eg something similar to the component name.

6. Declare a variable of that class, and call [_create \(167\)](#).

7. Then you can call component's functions, and/or set events. Available functions are shown when you type dot after variable name. Those with green brick icon are methods. Gray icons - properties. Lightning icons - [events \(169\)](#).

Example:

```
Excel.Application a._create  
a.SomeMethod(arguments)  
somevariable=a.SomeProperty  
a.SomeProperty=somevalue
```

Example2: installing and using an ActiveX control

1. Find it on the Internet, and download. When searching, you can include keywords "ActiveX control".
2. Install. Same as above.
3. Create or open the smart dialog where you will place the ActiveX control.
4. In the Dialog Editor, click "ActiveX controls..." in the list. It opens the COM Libraries dialog that displays available (registered) ActiveX controls.
5. In the COM Libraries dialog, select the control and click Add Control To Dialog. Click Yes if prompts to insert type library declaration.
6. It adds the control or its placeholder to the dialog. You can move/resize/zorder it like any other control.
7. If you want to use events, click the control in the Dialog Editor and click Events. It inserts a statement that declares an interface pointer variable for the control, and another statement that calls [_getcontrol \(167\)](#) function. It also gives you instructions how to set events. The declaration is inserted under WM_INITDIALOG. If you will use the control under other case statements too, use the [_getcontrol](#) function everywhere to initialize the variable.
8. Then you can call component's functions, and/or set events. Same as step 7 above.

[Example \(63\)](#)

Declare (242) COM type library

Syntax1 - declare a type library by GUID

```
typelib libname guid vermajor.verminor [lcid] [flags]
```

Syntax2 - declare a type library by file name

```
typelib libname file [flags]
```

Syntax3 - use a previously declared type library

```
typelib libname
```

Parameters

libname - [name \(257\)](#) of type library. It can be any name that you want to use to identify the type library later in code.

guid - [globally unique identifier \(249\)](#) of type library.

vermajor.verminor - major and minor version. Digits.

lcid - locale identifier. Integer constant. Default: 0 (neutral).

flags (247):

1	load on demand. The type library will be loaded into memory only when actually needed.
2	global scope. Identifiers from this type library can be used anywhere without specifying libname .

(246)file - type library file.

Remarks

To insert this statement, use the [COM Libraries dialog \(163\)](#) (menu Tools -> COM Libraries).

Syntax1

Declares a registered type library.

Syntax2

Declares a type library from **file**. By default, is used the type library from the first resource in the file. To use other resource, append \ and resource index (e.g., "c:\abc.dll\2").

Syntax3

Makes identifiers from a previously declared type library available in the current macro or function without specifying **libname**. Also, if the type library is still not loaded, loads.

A type library usually contains information about one or more COM classes (coclasses) and associated interfaces. Also can contain other declarations, such as dll functions, constants and types. It makes programming easier, because you don't have to manually declare them ([type](#), [def](#), [dll](#) and [interface](#) statements). It is either separate file (.tlb, .olb) or part of the component (.dll, .exe, .ocx). Typically, OLE-Automation-enabled applications and ActiveX components provide type libraries.

Accessing identifiers declared in type library

If a type library is declared with global scope (flag 2), identifiers from it (types, constants, etc) can be used anywhere, without any special syntax. Otherwise use syntax `libname.identifier`, unless the type library is declared in the same macro/function. Also, when using `libname.identifier` syntax, the identifier is automatically declared and colored when typing or opening macro.

See also: [sub-function attribute r \(182\)](#), [application folders \(11\)](#).

Errors

If a type library cannot be loaded (file not found, not registered, etc), is generated error, and compilation stops. If you are not sure that the component exists on user's system, you can use [#err \(175\)](#) to mute the error, compile alternative code or call a user-defined function. Even if flag 1 (load on demand) is set, [#err function](#) works: if function returns nonzero, error is not shown.

Tips

To see what is inside type libraries, you can download and install a type library browser, such as Microsoft's OLE Object

Viewer (free).

See also: [ref \(160\)](#) [declarations \(242\)](#) [scope \(257\)](#)

Examples

```
typelib Word {00020905-0000-0000-C000-000000000046} 8.0 0x409
typelib mytypelib "c:\type libraries\mytl.tlb"

Word.Application wa
int i = mytypelib.MyFunction(0)
```

Declare (242) COM interface

Syntax

```
interface[@|#] typename :baseinterface memberfunctions [{guid}]
```

Parameters

typename - interface [name \(257\)](#).

baseinterface - interface from which this interface is inherited.

memberfunctions - list of member functions.

- Member functions and other parts after **typename** can be declared in multiple tab-indented lines.

guid - [GUID \(249\)](#) of the interface. It is implicitly used by [_create \(167\)](#) and in some other cases.

Options:

@	Declare dispinterface. <ul style="list-style-type: none"> • Functions will be called through IDispatch::Invoke. • Default: declare interface. Functions will be called directly.
#	All functions are Automation-compatible. <ul style="list-style-type: none"> • The return value in the function's C++ declaration actually has HRESULT type. In QM it is hidden. • If a function returns negative HRESULT value, QM generates run-time error.

Member function syntax:

```
[[attributes]] [membertype]membername [(parameters)]
```

attributes - one or more of the following literals:

f	method (default).
p	property-put function.
g	property-get function.
r	property-put byref function.
h	function is Automation-compatible. <ul style="list-style-type: none"> • Optional if used option # or attribute a or l. Cannot be used with dispinterfaces. • QM 2.2.0. If function's return type is specified, [h] is interpreted like [a]. In previous versions, if only [h] was specified, the function was interpreted as not returning a value.
a	function is Automation-compatible. It has a hidden parameter for return value (not specified in parameters). <ul style="list-style-type: none"> • Optional if used option #. • Cannot be used with dispinterfaces.
l	function is Automation-compatible. It has a hidden parameter for locale (not specified in parameters). <ul style="list-style-type: none"> • Cannot be used with dispinterface.

[\(266\)membertype](#) - type of the return value.

membername - function's [name \(257\)](#).

parameters - parameters. Same as with [dll \(153\)](#). Parentheses (maybe empty) must be used, unless parameters are not defined.

See also: [declarations \(242\)](#), [scope \(257\)](#)

Remarks

Declares a [COM \(162\)](#) interface. A COM interface defines functions that can be used with a COM object. [Read more.](#)

Physically, a COM interface is an array of function addresses. An interface pointer variable holds the address of a COM object, which holds the address of that array. A COM object can have one or more different interfaces. All COM interfaces are inherited from (i.e. includes functions of) base interface IUnknown (directly or through another base interface). IUnknown has 3 member functions: QueryInterface, AddRef and Release. QueryInterface gets pointer to some other supported interface, others control object's life time. IDispatch also has 4 member functions that allow calling other functions by name. Interfaces IUnknown and IDispatch are defined by QM. For all interfaces, QM implicitly calls functions

of IUnknown and IDispatch when needed. It is not error to call them explicitly, although they are not included in the popup list of member functions of other interfaces.

For interfaces, must be declared all member functions. A function is called by the position in the array, therefore functions must be declared in exact order (but you can change names). You can fully declare only functions that you will use. All other functions can be for example x. If a function is declared without parameters, then, when calling, you can pass any number of arguments, but argument types are not converted therefore must match expected. To explicitly specify that a function has 0 parameters, use empty parentheses.

For dispinterfaces, can be declared only some functions, in arbitrary order. A function is called by name therefore names cannot be changed.

Interfaces also can be declared in [reference files \(15\)](#) and [type libraries \(164\)](#), which allows you to use them without declaring explicitly. Many declarations are in WINAPI and WINAPIV reference files. Usage example:

```
WINAPI.ITaskScheduler ts
```

QM 2.3.0: Can be defined optional parameters, like with [dll \(153\)](#).

QM 2.3.0. Allowed comments, like with [type \(154\)](#).

QM 2.4.1. You can declare parameters as [reference instead of pointer \(147\)](#). For example, `VARIANT&p` is the same as `VARIANT*p`.

Examples

```
interface IDispatch :IUnknown
_#GetTypeInfoCount(*pctinfo)
_#GetTypeInfo(iTInfo lcid ITypeInfo*PTInfo)
_#GetIDsOfNames(GUID*riid word**rgszNames cNames lcid rgDispId)
_#Invoke(dispidMember GUID*riid lcid @wFlags DISPPARAMS*pDispParams VARIANT*pVarResult
EXCEPINFO*pExcepInfo *puArgErr)
_{00020400-0000-0000-C000-0000000000046}

interface IAccessible :IDispatch
_[ga]IAccessible'Parent()
_[ga]ChildCount()
_[ga]IAccessible'Child(VARIANT'varChild)
_[ga]BSTR'Name(VARIANT'varChild)
_[ga]BSTR'Value(VARIANT'varChild)
_[ga]BSTR'Description(VARIANT'varChild)
_[ga]VARIANT'Role(VARIANT'varChild)
_[ga]VARIANT'State(VARIANT'varChild)
_[ga]BSTR'Help(VARIANT'varChild)
_[ga]HelpTopic(BSTR*pszHelpFile VARIANT'varChild)
_[ga]BSTR'KeyboardShortcut(VARIANT'varChild)
_[ga]VARIANT'Focus()
_[ga]VARIANT'Selection()
_[ga]BSTR'DefaultAction(VARIANT'varChild)
_[h]Select(flagsSelect VARIANT'varChild)
_[h]Location(*pxLeft *pyTop *pcxWidth *pcyHeight VARIANT'varChild)
_[a]VARIANT'Navigate(navDir VARIANT'varStart)
_[a]VARIANT'HitTest(xLeft yTop)
_[h]DoDefaultAction(VARIANT'varChild)
_[ph]Name(VARIANT'varChild BSTR'szName)
_[ph]Value(VARIANT'varChild BSTR'szValue)
_{618736e0-3c3d-11cf-810c-00aa00389b71}

interface# ITask :IScheduledWorkItem
_SetApplicationName(@*pwszApplicationName)
_GetApplicationName(@**ppwszApplicationName)
_SetParameters(@*pwszParameters)
_GetParameters(@**ppwszParameters)
_SetWorkingDirectory(@*pwszWorkingDirectory)
_GetWorkingDirectory(@**ppwszWorkingDirectory)
_SetPriority(dwPriority)
```

165. interface

```
GetPriority(*pdwPriority)
SetTaskFlags(dwFlags)
GetTaskFlags(*pdwFlags)
SetMaxRunTime(dwMaxRunTimeMS)
GetMaxRunTime(*pdwMaxRunTimeMS)
{148BD524-A2AB-11CE-B11F-00AA00530503}
```

```
interface@ IExample :IDispatch #Method(VARIANT'v) [g]BSTR'Property() [p]Property(BSTR'name)
```

Interface pointer variables

An interface pointer variable is a reference to a COM object. You use it when calling COM object functions. Declaration syntax:

```
Interface ip
```

Here **Interface** is either interface name or coclass name. **ip** is variable name. If you use interface name, the variable has **Interface** type. If you use coclass name, the type of the variable is the type of the default interface of the coclass.

Usually, coclasses and interfaces are declared in type libraries. You should use type library name too, like in this example:

```
Excel.Application app
```

When you assign one interface pointer variable to another (for example, using operator =), it does not copy the object, but just increments its reference count. Both variables will refer to the same object. That is fast. QM may generate error on interface pointer type mismatch in an assignment operation. Use [operator + \(134\)](#) to compile without error. If that interface is not supported, error will be generated at run time.

```
Interface1 a._create
Interface2 b+=a
```

To explicitly delete the object, assign 0 to the interface pointer variable. It decrements object's reference count. It is not necessary because QM does it implicitly when the variable goes out of scope. You should not call AddRef or Release, because QM manages this automatically.

COM objects usually cannot be used by multiple threads, and therefore interface pointer variables should be [declared \(141\)](#) with local or thread [scope \(142\)](#).

An interface pointer variable itself is not an object. It is just pointer to the object, and double pointer to the interface. Like any other pointer, it is 4-byte integer.

COM object creation functions

When you declare an interface pointer variable, initially it is 0. That is, it does not refer to an object. Before using it to call COM object functions, need to create new object or get existing object. For this purpose, you can use functions described in this topic. Common syntax is:

```
ip.function(parameters)
```

Here **ip** is interface pointer variable. Each function returns **ip** itself.

```
ip._create([class] [dll])
```

Creates new COM object of **class** type and populates **ip** with object's address.

If **class** is omitted or 0, uses class of **ip** (finds it in the type library, if possible). As **class**, you can use CLSID or ProgId of object's class. It can be either GUID* (see [uuidof \(110\)](#)), or GUID string like "{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}", or ProgID string like "Application.Class".

QM 2.3.4. If the COM component is in a dll file, you can specify the **dll**. Then don't need to register the component. However it works not with all components. If dll not found, tries to create as registered. In exe, if **dll** begins with "\$qm\$", searches for the dll in exe folder; if not found there, searches in QM folder if QM is installed on that computer.

QM 2.3.5. To create COM components without registration, you can use manifest and **__ComActivator** class. It works with .NET COM components too.

The COM component must be [registered \(163\)](#), unless you use **dll** or **__ComActivator**. Else QM tries to register the component automatically, but this should be used only while testing a new component. To register automatically, need administrator privileges. For this reason, auto registration will not work on user accounts with limited privileges and on Windows Vista/7/8/10 if QM is running not as administrator. Also, it does not work in [exe \(20\)](#). Read [here \(163\)](#) about installing/registering and using COM components.

To create object that must exist while dialog is running, declare the variable in dialog procedure, under WM_INITDIALOG, as thread variable (with -), e.g., `Typelib.Class- obj. _create`. Or, you can declare it before [ShowDialog](#). If you declare it as local (without -) in dialog procedure, it will be quickly destroyed, because dialog procedure is called multiple times while the dialog is displayed. Actually, thread variables are destroyed when thread ends (macro ends), not when destroying dialog. If it is important to delete the object immediately when dialog is closed, under WM_DESTROY assign 0, e.g., `obj=0`.

Not all COM classes support creating objects using this function. Assume there are Class1 and Class2 in the type library. If you cannot create object of Class2, try to create object of Class1 and get object of Class2 using a function of Class1.

This function is not used to create [ActiveX controls \(63\)](#) for dialogs, but it can be used to create ActiveX controls that provide useful functions without being visible.

QM 2.2.1. Some ActiveX controls created using this function will not work anymore. Place them in a dialog. Some others will work better.

QM 2.3.4. Supports QM interfaces **IStringMap**, **ICsv** and **IXml**. Use without arguments.

```
ip._getactive([class] [flags] [moniker])
```

Gets currently active object of **class** type and populates **ip** with object's address.

class - object's class (see `_create`). If omitted or 0, uses class of **ip**.

flags (247):

0 (default)	If there is no active object, generate error.
1	If there is no active object, create new.
16	Display monikers of all running objects. Example:

```
IUnknown u._getactive(0 16 "."); err
```

moniker (QM 2.2.0) - object's name in the system Running Object Table (ROT).

- Can contain [wildcard characters \(196\)](#).
- For example, if it is a document object, its moniker string probably is file path.
- QM 2.3.3. Fixed bug: when wildcard used, may return object of incorrect type. For example, you could not use **moniker** "".

This function may fail on Windows Vista/7/8/10. There are several [workarounds \(277\)](#).

```
ip._getfile(file [class])
```

Gets object from **file**, and populates **ip** with object's address.

If **class** not specified (omitted or 0), uses CoGetObject. It starts associated application and opens the file. If the file is already open, gets object of the open file (like `_getactive`).

- **file** also can be moniker string. For example, to get WMI services use "winmgmts".
- Uses class of **ip** (see `_create`).

If **class** specified, uses different method. Creates object of **class** type (like `_create`) and loads the file using IPersistFile.Load. Does not try to get an open file object.

```
ip._getcontrol(hwndcontrol)
```

Retrieves ActiveX control object that is hosted by **hwndcontrol** in a [dialog \(63\)](#). An ActiveX control is created and hosted by a child window of class "ActiveX", so **hwndcontrol** must be handle of that child window. To get it, use function `id`. For example, if control id in dialog is 3, then you can use `id(3 hDlg)`. Usually it is not the id you see in QM status bar, because an ActiveX control also creates its own child window that covers the host child window. The id is shown in the dialog definition (it is the first number in the green line with "ActiveX"). This function is inserted when you click the Events button in the Dialog Editor.

```
ip._setevents([eventfolder] [flags])
```

Activates or deactivates object's [events \(169\)](#).

You also can use other ways to create/get COM objects. You can use universal or specialized API functions (example 4). Objects that are not externally creatable, usually are retrieved using properties (functions) of other objects of that component (example 3). Functions `acc`, `htm` and `web` also return interface pointer.

Examples

1. Create Excel Application object and make window visible:

```
typelib Excel {00020813-0000-0000-C000-000000000046} 1.2
Excel.Application a._create
a.Visible=TRUE; err
```

2. You can specify class, although usually this is not necessary. Several examples how can be specified class:

```
Excel.Application b c d
b._create("Excel.Application")
c._create(uuidof(Excel.Application))
d._create("{00024500-0000-0000-C000-000000000046}")
```

3. Get Excel Workbook object from file; get pointers to other interfaces; make window and document visible:

```
Excel.Workbook x._getfile("$desktop$\Book5.xls")
Excel.Application a=x.Application
a.Visible=TRUE; err
Excel.Windows ws=a.Windows
Excel.Window w=ws.Item(1)
```

```
w.Visible=TRUE
```

4. Use dll function to get interface pointer:

```
IAccessible a  
if (AccessibleObjectFromWindow(hwnd 0 uuidof(IAccessible) &a)) ret  
out a.accName(0)
```

Calling COM functions

Syntax

Methods, property-get functions and property-put functions are called using syntax:

```
[... = ]ip.method[(parameters)]
[... = ]ip.property[(parameters)]
ip.property[(parameters)] = value
```

Here **ip** is interface pointer variable; **method** and **property** are names of functions. Parameters often have BSTR or VARIANT type, but you can pass values of QM intrinsic types (str, etc) instead. QM converts them automatically.

Errors

Most COM functions actually return error code, which is 0 on success, negative on error, positive on partial success or if function returns some information. In C++ it is defined as integer of type HRESULT (int in QM). You don't use it when calling functions. Instead, as return value is used the last parameter if in type library it is marked as return value. If function returns negative HRESULT, QM generates error, which can be handled ([err \(129\)](#)). This behavior is only if function's return value is declared as HRESULT (near all functions are declared in this way). QM stores last called COM function's return value in special variable [_hresult \(144\)](#).

Multiple calls in single statement

A single statement can contain more than one function call. If a function returns interface pointer, you can immediately call a function on that interface pointer. For example, statement:

```
Excel.Worksheet xlSheet+=xlApp.Workbooks.Add(xlWBATWorksheet).ActiveSheet
```

does the same job as the following three statements:

```
Excel.Workbooks xlBooks=xlApp.Workbooks
Excel.Workbook xlBook=xlBooks.Add(xlWBATWorksheet)
Excel.Worksheet xlSheet+=xlBook.ActiveSheet
```

QM 2.3.0: This also can be used with interfaces declared using [interface \(165\)](#).

Arguments passed by reference

For "by reference" parameters (& in QM status bar), you must pass variable of that type, or address of variable of that type. You cannot pass simple number or string as you could do in Visual Basic. For example, VB statement

```
doc = Documents.Add("file")
```

would be the same in QM, but only if argument is passed by value (e.g., declared as VARIANT filename). If argument is passed by reference (e.g., declared as VARIANT&filename), it should be

```
VARIANT v="file"; doc = docs.Add(v)
or
VARIANT v="file"; doc = docs.Add(&v)
```

Default member

QM does not support default members. For example, Visual Basic statement

```
doc = docs(1) ;;get item #1 from collection
```

in QM could be

```
doc = docs.Item(1)
```

In the popup list of functions default member is blue.

Optional parameters

QM partially supports optional parameters.

In QM status bar, optional parameters are displayed with [], like [param]. Optional parameters that have a default value are displayed like [param=1].

Optional parameters of VARIANT type can be omitted. Or use @ instead. If there is a default value, use the value.

In the example, we call function Func. Assume it has 3 parameters, all optional. We pass x for parameter 2.

```
ip.Func (@ x)
```

Variable number of arguments

Some COM functions support variable number of arguments. For such functions QM displays "vararg" in status bar.

Store arguments into an ARRAY(VARIANT) variable and pass it as the last argument. Example:

Assume that QM status bar shows this: Func(BSTR's ARRAY(VARIANT)*a) . vararg

```
ARRAY (VARIANT) a.create (2)
a[0] = 1; a[1] = "string2"
ip.Func ("string1" &a)
```

When calling through IDispatch::Invoke, simply pass the arguments. IDispatch::Invoke is used when the variable is of type IDispatch, or #opt dispatch 1 is set, or the interface does not support vtbl binding. Example:

```
ip.Func ("string1" 1 "string2")
```

I don't have type library, and don't want to declare interface

You can also call functions of undeclared interfaces. Interface pointer variable must be of IDispatch type. Example:

```
IDispatch app._create ("Word.Application")
app.Visible = -1
```

Argument types should match the expected types. QM does implicit conversions only where it is obvious, such as str to BSTR. The type of the return value is always VARIANT.

QM cannot show type info (popup list of functions, and in status bar) for undeclared interfaces.

You can also use VBScript or JScript code in QM. Example:

```
lpstr code=
Set app=CreateObject ("Word.Application")
app.Visible=True
VbsExec code
```

Or, parts of code that manipulates single object can be in VBScript, and other parts in QM. Functions [VbsFuns](#) and [VbsEval](#) can be used to exchange values between VBScript and QM.

Examples

See [other \(167\)](#) topics and COM samples in the [forum](#).

COM events

Syntax

```
ip._setevents(["eventfolder"] [flags])
```

Parameters

ip - interface pointer variable.

eventfolder - where are event functions. Can be:

- QM 2.4.1. "sub" or "sub.prefix". Use it if event functions are [sub-functions \(182\)](#) in the same macro/function as the `_setevents` statement.
- A folder (in the list of macros) containing event functions. Not recommended.
- If **eventfolder** is omitted or "", disconnects events.

flags (247):

1	don't allow multiple connections on the same object.
---	--

Remarks

Usually you call COM object's functions ("methods" and "properties"). Some COM objects also support "events", that is, can call your functions (event functions). But at first need to connect object's events to your functions.

`_setevents` connects or disconnects object's events.

Example:

```
Excel.Application x._create
x._setevents("sub.x")
```

When object (in the example - variable x) fires an event, the corresponding function from **eventfolder** (in the example - a [sub-function \(182\)](#)) is called.

Creating a new event function:

- Type `.` after an interface pointer variable. It shows a popup list of functions that also may contain events (lightning icons ⚡). Double click an event. It adds `_setevents` (if does not exist) and creates an event-function (sub-function) with [correct name and parameters](#). Then you can add more code in the sub-function.
- You also can invoke the popup list from the Dialog Editor: select an ActiveX control (eg web browser) and click Events.

Names of event functions must consist of some prefix, underscore (`_`) and event name. If **eventfolder** is like "sub.prefix", names must have that prefix, else can have any prefix. For example, if **eventfolder** is "sub.x", will be connected sub-functions like "x_Event1", "x_Event2" etc. Other sub-functions can be used for other purposes, for example with other `_setevents` statements.

Event function parameters must match event's parameters. Event functions don't return a value. If your function returns some nonzero value, it is interpreted as an error code.

When using a QM folder for event functions, if there are several folders with same name, `_setevents` uses one that is nearest in the list of macros. At first it searches in the same folder (where function that calls `_setevents` is) and subfolders, then in its parent folder and its subfolders, then in other ancestors, and finally in whole list of macros. To avoid all this confusion, use sub-functions instead.

Usually COM objects work only in single thread (running macro). To receive events, thread where the object is created must be running. If it ends immediately, events have no sense. For example, you can use a dialog. Also you can use `wait`, but then need to insert `opt waitmsg 1` before, or events will be blocked. See example.

To access the event-source object variable from an event function, use one of:


- Declare it as the last parameter in the `function` statement. When the function is auto-created, the parameter is added as comments, need just to remove ";,;". Example:

```
function x y Excel.Application 'x
x.Function()
```

- Sub-functions with attribute v can use parent's variables.
- Declare the variable with [thread scope \(142\)](#) (with `-`). Don't use this with ActiveX controls in dialogs.

By default, multiple calls to `_setevents` create multiple connections. For example, you can connect two folders simultaneously. If flag 1 is set, previous connection is deleted before making new.

To disconnect, call `_setevents` without arguments. Usually it is not necessary because it happens automatically when the object is deleted.

Events can be used only with coclasses (icon ) from type libraries.

`_setevents` must be used directly with a variable. Code like `var.Function._setevents(...)` will not work.

Example - Excel, wait

```
ExcelSheet es.Init
Excel.Worksheet x=es.ws
x._setevents("sub.x")
opt waitmsg 1
10

#sub x_SelectionChange
function Excel.Range'Target ;;Excel._Worksheet'x
out 1
```

Example - dialog, web browser control

```
\Dialog_Editor

str controls = "3"
str ax3SHD
ax3SHD="http://www.quickmacros.com"
if(!ShowDialog("" &sub.DialogProcedure &controls)) ret

BEGIN DIALOG
0 "" 0x90C80AC8 0x0 0 0 224 136 "Dialog"
3 ActiveX 0x54030000 0x0 0 0 224 114 "SHDocVw.WebBrowser"
1 Button 0x54030001 0x4 116 116 48 14 "OK"
2 Button 0x54030000 0x4 168 116 48 14 "Cancel"
END DIALOG
DIALOG EDITOR: "" 0x2040104 "*" "" "" ""

#sub DialogProcedure
function# hDlg message wParam lParam
sel message
_case WM_INITDIALOG
_SHDocVw.WebBrowser we3
we3._getcontrol(id(3 hDlg))
we3._setevents("sub.we3")
_case WM_DESTROY
_case WM_COMMAND goto messages2
ret
messages2
sel wParam
_case IDOK
_case IDCANCEL
ret 1

#sub we3_DocumentComplete
function IDispatch'pDisp `&URL ;;SHDocVw.IWebBrowser2'we3
out URL
```

Variable types CURRENCY, DECIMAL, VARIANT

A variable of type CURRENCY is stored as 8-byte integer, scaled by 10000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. This provides a range of 922337203685477.5807 to -922337203685477.5808. Useful for calculations involving money, or for any fixed-point calculation where accuracy is particularly important.

A variable of type DECIMAL is stored as 12-byte integer, scaled by a power of 10.

A variable of type VARIANT can hold a value of type int, byte, word, double, long, BSTR, DATE, FLOAT, CURRENCY, DECIMAL, arrays, interface pointers and several other types. The first member (vt) of VARIANT holds [type information \(269\)](#). The next three words are used only by DECIMAL. Then follows 8-byte union that can hold value of other supported types. An union is a variable type whose members can have different types but they all share the same memory. Variables of type VARIANT are automatically [cleared \(265\)](#) when go out of scope.

All these types support operator = (assign) and [unary operators \(134\)](#). To assign a string containing a number, [val \(186\)](#) is not needed. Error if = cannot convert from another type.

Functions

All these types have the same set of arithmetic functions. These functions provide more precise results than calculations with operators. Common syntax:

```
var.function(parameters)
```

Here **var** is a variable of type CURRENCY, DECIMAL or VARIANT. Arguments can have any type, but before calculating they are converted to the type of **var**. Most functions return **var** itself.

```
var.add([left] right)
```

Adds **left** and **right** (var = left + right). If **left** is omitted, **var** is used (var = var + right).

```
var.sub([left] right)
```

Subtracts (var = left - right).

```
var.mul([left] right)
```

Multiplies (var = left * right).

```
var.div([left] right)
```

Divides (var = left / right).

```
var.round([number] [cDec])
```

Rounds **number** to **cDec** places after decimal point. Default for **cDec** is 0. Default for **number** is **var**.

```
var.fix([number])
```

Gets integer part of **number**. Default for **number** is **var**.

```
int var.cmp(numberorstring [flags])
```

Compares **var** and **numberorstring**. Returns -1 if **var** is less than **numberorstring**, 0 if equal, 1 if greater.

QM 2.3.5. Added **flags**. Flag 1 - case insensitive.

VARIANT functions

```
var.attach(a)
```

Here **a** is variable of type ARRAY, BSTR, VARIANT or interface pointer.

The function stores **a** into **var** without copying the associated data (as operator = does). Clears **a**.

Examples

```
VARIANT a
a.add(10 0.45)
now a is 10.45
a.sub(a.mul(5 10) 0.45)
now a is 49.55
a.round(2.52)
now a is 3
a.round(2.52 1)
now a is 2.5
a.fix(2.52)
now a is 2
a = -a
now a is -2
a = "string1"
a.add(" string2")
now a is "string1 string2"
```

Variable type BSTR

With COM functions usually are used strings of type BSTR. It is [Unicode \(267\)](#) UTF-16 string.

The BSTR type supports operator = (assign). You can assign values of intrinsic types (str, int, etc) and other OLE types (VARIANT, DATE, etc). They are automatically converted to BSTR. A BSTR variable itself can be assigned to a str or other variable. Read more about [converting to/from UTF-16 \(238\)](#).

Functions

Here **var** is a variable of type BSTR. Where the return type is not specified, the function returns **var** itself.

```
var.add([left] right)
```

Joins **left** and **right** (var = left + right). Arguments can have any type. If **left** is omitted, **var** is used (var = var + right).

```
int var.cmp(string [flags])
```

Compares **var** and **string**. Returns -1 if **var** is less than **string**, 0 if equal, 1 if greater.

Here "less" means that **var** would be above **string** in a sorted list.

QM 2.3.5. Added **flags**. Flag 1 - case insensitive. Other flags are rarely used and are the same as with [VarBstrCmp](#), documented in the MSDN Library.

```
var.alloc(nchar)
```

Allocates memory for a string of **nchar** characters, and appends [null character \(183\)](#).

If the variable was empty, the function sets the first character in the allocated memory to 0 (QM 2.3.0). The remaining part is not initialized.

If the variable was not empty, preserves the string, but no more than **nchar** characters. The remaining part is not initialized.

```
var.free
```

Frees string.

```
int var.len
```

Returns string length. It is number of 2-byte characters, not including the terminating null character.

To get UTF-16 string length also can be used function [len \(184\)](#) (QM 2.3.0) or [wcslen](#). They calculate string length by searching for the terminating null character, and therefore may return different value if the string contains binary data (null characters in middle).

Examples

```
BSTR a = "string1"
BSTR b.add(a " string2")
now b is "string1 string2"
str s = a
a = s
```

Compiler directives: **#if, #else, #endif**

Syntax

```
#if constexpr
statements
[#else
statements]
#endif]
```

Parameters

constexpr - constant integer expression.

- Also can be a global variable of type int.
- Also can be a user-defined function that returns int. It is executed while compiling, in QM main thread.
- Expression with operators is evaluated from left to right, regardless of operator priority.

statements - any statements.

Remarks

If value of **constexpr** is not 0, are compiled statements after **#if** and skipped statements after **#else**. Otherwise, are skipped statements after **#if** and compiled statements after **#else**.

Unlike [if \(123\)](#), compiler directives are evaluated only at [compile time \(47\)](#).

If **#endif** is omitted, are compiled (or skipped) all following statements until the end.

QM 2.4.1. **#if/#ifdef/#ifndef** can be nested in other **#if/#ifdef/#ifndef** or **#else** code block. Also now most directives can be tab-indented.

See also: [#ifdef \(173\)](#), [#ret \(181\)](#), [predefined variables and constants \(144\)](#), [make exe \(51\)](#).

Example

```
def TEST 5
#if !TEST
out "TEST is 0"
#else
rep 2
  _if TEST>=5
  out "TEST is >= 5"
  _else
  out "TEST is < 5"
  _endif
#endif
```

Compiler directives: **#ifdef**, **#ifndef**

Syntax

```
#if[n]def identifier
statements
[#else
statements]
#endif
```

Parameters

identifier - global identifier (named constant, type, function, global variable, etc).
statements - any statements.

Remarks

If **identifier** is defined (or not defined, if **#ifndef** used), are compiled statements after **#ifdef** (or **#ifndef**) and skipped statements after **#else**. Otherwise, are skipped statements after **#ifdef** (or **#ifndef**) and compiled statements after **#else**.

If **#endif** is omitted, are compiled (or skipped) all following statements until the end.

QM 2.4.1. **#if/#ifdef/#ifndef** can be nested in other **#if/#ifdef/#ifndef** or **#else** code block. Also now most directives can be tab-indented.

See also: [#if \(172\)](#).

Example

```
if function "MyFunction" exists, call it:
#ifdef MyFunction
MyFunction 100
#endif
```

Compiler directives: **#compile**

Syntax

```
#compile [+|*] "macro"
```

Parameters

macro - name of a macro or function. Also can be [sub-function \(182\)](#), like "sub.SubName".

Options:

+	macro is folder. Compiles all items in it. If macro is "sub", compiles all sub-functions.
*	compile and execute. <ul style="list-style-type: none"> • macro must be user-defined function or sub-function. It runs while compiling, in QM main thread. • This is useful if you want to set global or environment variables that are used later while compiling current macro (with #if, etc).

Remarks

[Compiles \(47\)](#) **macro**. It does not "include" or call it, but just compiles, if it is not already compiled. Usually it is used to compile macros or functions that contain various [declarations \(242\)](#).

See also: [ref \(160\)](#), [CompileAllItems \(114\)](#).

Example

```
#compile "toolhelp"
```

Compiler directives: **#err**

Syntax

#err

Syntax2

#err *nlines*

Syntax3

#err *function*

Parameters

nlines - number of lines to skip if error does not occur. Digits.

function - name of user-defined function.

- QM 2.4.3. Can be [sub-function \(182\)](#).

Remarks

Handles compile-time error generated in previous statement.

Syntax1: On error, continue. Error is not generated, and statement with error is excluded.

Syntax2: On error, compile following **nlines** lines. If error does not occur, following **nlines** lines are skipped. If **nlines** is 0, on error all following code is not compiled.

Syntax3: On error, call **function**. It runs synchronously (at compile time) in QM main thread. If it returns 0, error is generated as usually. If 1, error is not generated, and statement with error is excluded (same as **#err**). If -1, error is generated but not displayed. If -2, error is not generated, and all following code is not compiled (same as **#err 0**).

Special variable [_error \(129\)](#) is filled with information about error, and function **function** can access it. Currently, **code** is always 0 on compile-time errors.

See also: [errors \(48\)](#), [err \(129\)](#)

Examples

```
dll somedll DllFunctionThatMaybeMissing a b c
#err SayNoFunction

typelib VBScript_RegExp {3F4DACA7-160D-11D2-A8E9-00104B365C9F} 1.0
#err 1
typelib VBScript_RegExp {3F4DACA7-160D-11D2-A8E9-00104B365C9F} 0.0
```

Compiler directives: **#opt**

Syntax

#opt option value

Parameters

option - one of words in the table below.

value - one of values (0 or 1) in the table:

option	1 or other nonzero	0
err	On compile-time error continue. Skip statement with error. This option works not with all statements. Use it only where it has sense, like in the example below.	On compile-time error stop.
nowarnings	Don't display warnings.	Display warnings.
dispatch	Call COM functions through IDispatch.Invoke.	Call COM functions through VTBL (if available).
hidedecl (QM 2.3.2)	Hide declarations in popup lists (46) .	Don't hide declarations.

Remarks

Sets an option that is used when compiling current macro/function.

Initially all options are 0.

Tip: Other ways to hide something: 1. If name begins with "__". 2. If the user-defined function is in a [private folder \(11\)](#). 3. If the class member function is private or protected.

Examples

```
#opt err 1
FunctionThatMaybeMissing(a b c)
#opt err 0

#opt nowarnings 1
mac FunctionThatReturnsIdOfFunctionToRun
#opt nowarnings 0

typelib Excel {00020813-0000-0000-C000-000000000046}
#opt dispatch 1
Excel.Application xlApp._create
xlApp.Visible = -1

#opt hidedecl 1
class HiddenClass -m_x -m_y
int+ g_hiddenVariable
#opt hidedecl 0
```

Compiler directives: **#set**

Syntax

```
#set variable value
```

Parameters

variable - global variable of type `int`.

value - an integer value to assign. Can be simple number, global `int` variable, or user-defined function without arguments (runs at compile time in QM main thread). Sub-functions also supported.

Remarks

Sets value of a global `int` variable at compile time. The variable must exist.

See also: [#compile \(174\)](#)

Examples

create, set and use global variable at compile time

```
int+ g_var  
#set g_var GetGVar  
#if g_var
```

Compiler directives: **#out**, **#warning**, **#error**

Syntax

```
#out "text"  
#warning "text"  
#error "text"
```

Remarks

Displays **text** in the output while compiling the macro.

#out just displays the text.

#warning displays the text as warning.

#error generates compile-time error.

See also: [#if \(172\)](#)

Example

```
#if EXE  
#warning "this function is unavailable in exe"  
#else  
...  
#endif
```

Compiler directives: #exe

Syntax

#exe option value [resId] [resType]

Parameters

option	value	Description
addfunction	Function name.	Adds the user-defined function. Usually this is not necessary because all required functions are added automatically, but in some cases QM cannot know what function must be added. For example, with mac , if function name is variable, QM cannot know what function will be used, and displays a warning. Then use #exe .
addtextof	QM item name.	<p>Adds text of the QM item (macro, function, etc). Usually this is not necessary because all required items are added automatically, but in some cases QM cannot know what item must be added. For example, with str.getmacro, if macro name is variable, QM cannot know what macro will be used, and displays a warning. Then use #exe.</p> <p>Automatically adds text of items used with ShowDialog (item containing dialog definition or/and menu definition), scripting functions (item containing script), scan "macro:..." (item containing bitmap), wait S "macro:..." and some other functions.</p> <p>QM 2.3.5. Also automatically adds text of items from all string constants like "macro:MacroName". Previously this would work only with scan and wait. Multiple items can be added with code like this: <code>s="macro:Macro1[]macro:Macro2"</code>.</p> <p>QM 2.3.5. If value is "<script>", adds caller's text if it calls current function with first argument "", like VbsExec "".</p>
addactivex	ActiveX control class in form Typelib.Class.	Adds information about an ActiveX control (63) class. Usually this is not necessary because QM automatically adds information about ActiveX controls used in dialogs. You only need to use addactivex if you create ActiveX controls using CreateWindowEx or CreateControl (where window class is "ActiveX"), or create dialog definition at run time. QM 2.2.0: and only if class id is not included, ie window name is just Typelib.Class, without {clsid}.
addfile	File path. Can be full or relative to the folder where the exe file will be created.	<p>QM 2.2.0. Adds a file to exe resources.</p> <div> <p>resId - resource id, 1 to 0xFFFF. Must be unique in resources of this type. Can be string (QM 2.4.0).</p> <p>resType (QM 2.3.4) - resource type. Can be integer 1 to 0xFFFF (eg RT_BITMAP) or string (eg "WAVE"). Default (0): RT_RCDATA.</p> </div> <p>At run time you can extract the file with ExeExtractFile (114), or get file data with ExeGetResourceData (114), or use a Windows API function that supports resources.</p> <p>QM 2.3.0. Instead can be used syntax ":resourceid filepath" with file functions. Read more (51). Also, can be used str.getfile (217) to get data.</p> <p>QM 2.4.1. Can add a macro resource (261). See example.</p>

Remarks

Explicitly adds a function, text, ActiveX control class or a file to [exe \(51\)](#). Ignored if the macro runs in QM. Can be used in any place.

To disable warnings, use [#opt nowarnings 1 \(176\)](#).

Examples

```
#exe addfunction "Function5"
#exe addtextof "Macro5"
#exe addactivex "Typelib.Class"
```

```
#exe addfile "$my qm$\copy.bmp" 10 RT_BITMAP  
#exe addfile "resource:app.ico" 10 RT_GROUP_ICON
```

See also code in function [ExeQmGridDll](#). It adds a dll to exe, and extracts/loads at run time.

Compiler directives: **#region**

Syntax

```
#region [-] [name]
statements
[#endregion [comments]]
```

Parameters

name, comments - can be any text. Optional.
statements - any statements.

Options:

-	Don't nest this block in previous #region block. Adding this option is the same as inserting #endregion line before.
---	--

Remarks

Lets you specify a block of code that you can expand or collapse (hide, fold) in the code editor. Can be used in QM items of all types (macros, menus etc).

QM uses this directive only when displaying code in the code editor. When compiling macro or creating menu etc, #region and #endregion lines are ignored, like comments.

When first time displaying a #region line, QM adds a vertical bar with - and + boxes to collapse/expand regions. You also can use its right-click menu to collapse or expand all regions or create new region. Also you can use menu 'Edit -> Lines -> Hide selected' to create new region.

Added in QM 2.4.1.

Compiler directives: #ret**Syntax**

#ret

Remarks

Tells QM to not compile text that follows. Can be used to place any text in macro below QM code.

Added in QM 2.3.2.

Compiler directives: #sub

Syntax

#sub name [attributes]

Parameters

name - [name \(257\)](#) of the sub-function.

attributes - one or more characters to specify sub-function properties:

c	This sub-function is a class (157) member function of its parent function's class. Example: <code>#sub Member</code> c. To call: <code>sub.Member</code> or <code>var.sub.Member</code> .
m	Text of this sub-function is used as menu (22) item text in a menu, toolbar or autotext. The menu item must be like <code>Label :sub.SubName</code> . Cannot be called as function.
v	This sub-function can use parent's local and thread variables declared above the first sub call statement. Be careful with this, read more in Remarks -> More.
r	This sub-function sees parent's ref (160) and typlib (164) statements that are above the first sub call statement.
p	QM 2.4.3. Private. In a function containing shared sub-functions this sub-function is not shared. Sub-functions in other functions are always private, don't need this attribute.

Remarks

Begins a sub-function.

Added in QM 2.4.1.

Sub-functions are user-defined functions that are embedded in text of their *parent* [QM item \(19\)](#) (macro etc). Unlike [usual QM user-defined functions \(21\)](#), they aren't separate QM items that you could manage in the list of macros. Sub-functions can be used in macros, functions, member functions, menus, toolbars and autotexts.

To call a sub-function, use its name with "sub." prefix. Such sub-functions can be called from parent and its sub-functions, but not from other QM items. QM 2.4.3 also supports [shared sub-functions](#).

Sub-functions can be used like usual functions:

- Called directly, like `sub.SubName(1 2)` or `classVar.sub.SubName(1 2)`.
- As callback function, for example with `atend`, `win`, `ShowDialog`. To get address: `&sub.SubName`.
- Run in separate thread with `mac (100)`, like `mac "sub.SubName"`.
- As timer function with `tim (92)`, like `tim 1 sub.SubName`.
- With [COM events \(169\)](#).
- With directives [#compile \(174\)](#) and [#set \(177\)](#). QM 2.4.3: also with [#err \(175\)](#).

When you may want to use sub-function instead of usual function:

- You need a private function that cannot be used in other macros.
- You don't want to have the function in the list of macros.
- You want to edit it together (in same text) with parent function.
- When menu item text is multiline and you don't want to create macro/function for it.
- To use features specific to sub-functions, for example attribute v (use parent variables).

When you cannot use sub-function:

- The function is used by more than one macro/function/etc.
- The function must have a trigger or other properties of QM items that sub-functions don't have.
- Where sub-functions are not supported (error when trying to use).
- You want to make your macro compatible with older QM versions (< 2.4.1).

Sub-functions cannot be nested. This directive ends current code (parent or another sub-function).

All sub-functions can access private/protected members of [class \(157\)](#) of parent function, even if they don't have c attribute.

Like in all functions, default speed ([spe \(90\)](#)) in sub-functions is 0. In sub-functions with m attribute it is like in macros and menu items, default 100.

[More](#)

When compiling a sub-function, QM creates a temporary hidden QM item for it. These QM items have own QM item id and name (see [getopt \(98\)](#)), but don't have own text, trigger and most other properties of QM items. The names are like "<00001>SubName", where the number is parent QM item id, although usually displayed like "ParentName:SubName" or just "SubName". Most QM functions that find QM item by name support both "<00001>SubName" and "ParentName:SubName", but can find only if the temporary QM item already exists.

When a sub-function is specified, in most cases QM uses its parent instead:

- [str.getmacro \(219\)](#) gets parent text (all, including text of sub-functions).
- [qmitem](#) gets parent properties.
- [dis](#) disables/enables parent.
- [str.setmacro](#) replaces text of parent.
- All functions that use [macro resources and settings \(261\)](#).
- Functions that use the above functions. For example, [ShowDialog \(" " . . .\)](#) gets dialog definition from caller's text. If caller is a sub-function, looks in all parent text, not just in its part containing the sub-function. Similarly, [CsExec " " "](#) etc look for script code in all parent text.
- When creating exe, if need to add sub-function's text to exe, adds parent text.

When a sub-function is specified, QM does not use its parent with [IsThreadRunning](#), [EnumQmThreads](#), [shutdown -6](#) (end thread), [EndThread](#), [WaitForThreads](#), trigger "End thread". Example: [IsThreadRunning \("ParentName:SubName"\)](#).

Some other features that sub-functions don't have:

- Cannot be encrypted separately from parent.
- Cannot run in other process than parent.
- Cannot be the main function in exe.
- Cannot be a class special function - constructor, destructor, operator=.

Be careful with attribute v (use parent variables):

- When the sub-function is a callback function, in some cases it may be called when parent function is already returned, or even run in other thread. Then will be run-time error "cannot access variable". This attribute is not allowed with [mac](#) and in menus.
- When the parent function is called recursively (calls itself), its sub-functions with v attribute use variables of the nearest parent function instance in the call stack. If it is not what you need, the parent function should not be recursive.

Sub-function threads and timers are hidden by default in the 'Running items' pane. Right-click a folder to show.

Examples

```
sub.Test

#sub Test
out "sub-function Test"
out sub.Test2(3 4)

#sub Test2
function# a b
out "sub-function Test2"
ret a+b
```

Shared sub-functions

By default, sub-functions can be called only from parent and its sub-functions, but not from other QM items. QM 2.4.3 and later also supports shared sub-functions. They can be global and class.

Global shared sub-functions can be called from anywhere. They must be in functions named "__sub_X", where X can be any name. They are called like [sub_X.Func \(1 2\)](#).

Class shared sub-functions can be called from any member functions of that class, and from their sub-functions. They must be in class member functions named "Class.__subX", where Class is class name and X can be any name. They are called like [subX.Func \(1 2\)](#) or [classVar.subX.Func \(1 2\)](#). Use attribute c for class member sub-functions, like [#sub Member c](#). Functions without this attribute are like global functions, but they can be called only from functions of that class, and they can call any functions of that class (public, private and protected).

Shared sub-functions cannot have attributes v, r and m. Can have attribute p that makes the sub-function non-shared.

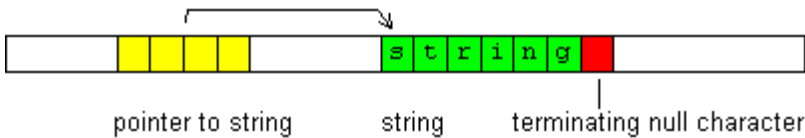
The `__sub_X` and `Class.__subX` functions can be used only as containers of shared sub-functions. They cannot be called etc. If you try to cal, run or compile such a function, instead are compiled all its shared sub-functions, which can be useful in debugging (eg to check for errors all at once).

Strings

See also: [string constants \(137\)](#), [using strings with functions \(150\)](#), [Unicode and UTF-8 \(267\)](#), [strings with variables \(138\)](#)

A string is an array of characters. Used to store text. Each character is stored in memory as a single byte (in UTF-8 - 1 to 4 bytes). A byte is an amount of memory that can have 256 different values. A byte with value 0 ("terminating null character") is appended to the end. Other values are used for [characters \(239\)](#).

A string [variable \(140\)](#) generally is (or includes) a pointer to a string. The pointer holds the address of the first character. The picture shows how whole string variable is stored in memory.



There are several types of string variables:

A variable of type `str` manages its string memory.

A variable of type `lpstr` does not manage string memory.

Variables of type [BSTR \(171\)](#) are used for Unicode UTF-16 strings. They are used mostly with COM functions. They manage string memory (like `str`).

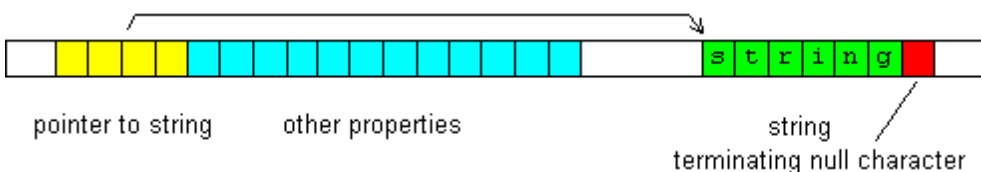
Sometimes variables of type `word*` also are used as Unicode UTF-16 strings. They don't manage string memory (like `lpstr`).

Variables of type [VARIANT \(170\)](#) are interpreted as strings when they contain BSTR. They are used mostly with COM functions.

In C++ programming language and in MSDN Library, a pointer to a string is often called `LPSTR`, `LPCSTR`, `char*`, `const char*`, etc. A pointer to an Unicode UTF-16 string is often called `LPWSTR`, `LPCWSTR`, `WCHAR*`, `OLECHAR*`, etc. Also can be `LPTSTR` and similar.

Variables of type `str`

Variables of type `str` are used to store and manipulate strings. A `str` variable holds a string in a variable-length automatically (re)allocated string buffer. The buffer is an array of bytes that the variable allocates somewhere in other memory location. Beside the pointer to the string buffer, a `str` variable has several other [properties \(234\)](#). All properties are automatically adjusted when performing string operations.



When you assign a string or numeric value to a `str` variable, that value is copied to the string buffer. Numeric values are automatically converted. Examples:

```
str s = "Cat"
now variable s is "Cat"
s = 5.75
now variable s is "5.75"
```

When you assign a `str` variable to a variable of other [intrinsic type \(141\)](#), the other variable receives the pointer value. The same is when you pass a `str` variable as a function argument. To convert a `str` variable to a useful numeric value, use [val \(186\)](#) function:

```
str s = 45
now s is "45"
int i = val(s)
now i is 45
```

A single character can be accessed using [] operator:

```
str s = "Cat"
int char = s[1]
now char is 97 ('a' character code)
s[0] = 66
now s is "Bat" (66 is 'B' character code)
s[2] = 'r'
now s is "Bar"
```

The str type supports [operators \(133\)](#) = (assign), + (append) and - (prepend):

```
s = "Cat"
now s is "Cat"
s + " and Mouse"
now s is "Cat and Mouse"
s - "Dog, "
now s is "Dog, Cat and Mouse"
```

Expressions like `s1 = s2 + s3` cannot be used to join strings, because s2 and s3 in such expression are interpreted as numeric (pointers). Instead use `s1.from(s2 s3)` or `s1=F"{s2}{s3}"`.

The str type also supports comparison operators = (equal, case sensitive comparison), ! (not equal, case sensitive comparison) and ~ (equal, case insensitive comparison):

```
if(s = "Cat") out "s is 'Cat'"
```

The str type has several member functions. Common syntax:

```
s.func(...)
```

Here **s** is a str variable; func is name of a member function; ... are parameters. You can declare variable and call function:

```
str s.function(...)
```

Most str functions return the str variable for which were called. Example:

```
str s.fix(GetWindowText(hwnd s.all(100) 100))
```

is equivalent to:

```
str s
s.all(100)
int i=GetWindowText(hwnd s 100)
s.fix(i)
```

Member functions:

lcase ucase (226)	Make string lowercase or uppercase.
unicode ansi (238)	Convert string to/from Unicode UTF-16.
escape (210)	Replace escape sequences with characters, or vice versa.
encrypt decrypt (209)	Encrypt, decrypt.
trim ltrim rtrim (237)	Remove characters from the beginning or end.
set (232)	Set part of string.
insert (225)	Insert other string.
remove (228)	Cut part of string.
replace (229)	Replace part of string.
findreplace (211)	Find and replace.
replacex (230)	Find and replace using regular expression.
addline (204)	Append string as new line.
left right get geta (227)	Get part of other string.

183. Strings

gett (223)	Find and get n-th part of other string.
getl (218)	Find and get n-th line.
getpath getfilename (221)	Extract path or filename from full file path.
from (214)	Create string from several parts (join).
fromn (215)	Create string from several string or binary parts of specified length.
format, formata (213)	Format string (create string that will contain values of variables).
all (205)	Allocate or free string buffer.
fix (212)	Set string length.
getwintext setwintext getwinclass getwinexe (224)	Get window text, class name, program, set window text.
getclip setclip getsel setsel (216)	Clipboard, copy, paste.
getfile setfile (217)	Read or write file.
searchpath, expandpath (231)	Find file and get full path; expand special folder.
dospath (208)	Get short (DOS) path from long path or vice versa.
getmacro (219)	Get macro text.
setmacro (220)	Set macro text.
dllerror (207)	Get dll error description.
timeformat (236)	Format date/time string.
beg begi end endi mid midi (206)	Compare part of string.
getstruct setstruct (222)	Get/set variable of user-defined type.
swap (233)	Exchange string with another variable.

Variables of type lpstr

Although in most cases you will use str variables, in some cases it is better to use lpstr variables. Unlike str variables, a lpstr variable has no string buffer. It is only a pointer to a null-terminated string (see the first picture above). It does not manage (which means allocating memory) the string to which it points.

Like str and pointers, lpstr supports [] operator to access a single character.

Also supports operators =, + and -, but they modifies only the pointer, but not the string to which it points:

```
lpstr s = "abcd"
now s is (points to) "abcd"
s + 2
now s is (points to) "cd"
s - 1
now s is (points to) "bcd"
```

You cannot use str member functions with lpstr variables.

You cannot assign a numeric value, except 0.

Null

A str or lpstr variable can be null. It means that the pointer to string is 0, and string memory is not allocated. This happens:

- When the variable is just declared and still not assigned a value.
- After calling [all \(205\)](#) without arguments.
- After the variable receives a str or lpstr expression that is null.
- In several other cases.

Note that it is not the same as "". When a str variable is "", its length also is 0, but it has string buffer which is required to hold terminating null character.

Common functions

Beside str member functions (listed above), you can also use global string functions. They can be used with str and lpstr variables. They are called like other global (non-member) functions. Example:

```
int lens
lpstr s="abc"
lens=len(s)
```

Functions:

len (184)	Get number of characters.
empty (185)	Is string empty?
val (186)	Get numeric value.
numlines (187)	Get number of lines.
find (188) , findw (189)	Find substring, find whole word.
findt (190) , findl (191)	Find n-th token, find n-th line.
tok (192)	Split.
findc, findcr (194)	Find character, find character from right.
findcs, findcn (194)	Find character that is or isn't in the specified set of characters.
findb (195)	Find substring in binary data.
findrx (197)	Find one or all substrings using regular expression.
matchw (196)	Compare strings using wildcards.

Global string functions are added to the [string category](#). There you also can find several useful functions from dlls. To compare strings, you can use [StrCompare \(114\)](#) and other functions. To get information about a dll function, press [F1 \(245\)](#).

Notes

String functions are not thread-safe. Don't use a variable in multiple threads simultaneously. It can damage data and make QM unstable. You can use [lock \(112\)](#) to prevent it.

Get string length

Syntax

```
int len(s)
```

Parameters

s - string.

- Can be of type str or lpstr.
- QM 2.3.0. Also can be BSTR, word*, VARIANT.

Remarks

Returns number of characters in string. To calculate it, searches for [terminating null character \(183\)](#). Returns 0 if **s** is null. Returns 0 if **s** is VARIANT of type other than VT_BSTR.

For str and lpstr, this function always returns number of bytes. In [Unicode \(267\)](#) mode, some characters consist of 2-4 bytes. For other types, always returns number of 2-byte characters.

Tips

For str and BSTR variables, use len property instead of this function. Example: `length = s.len`.

See also: [empty \(185\)](#).

Examples

```
lpstr s = "String"
int i = len(s)
now i is 6

str s = "one two"; s[3]=0
out s.len
out len(s)
Output:
7
3
```

Is string empty?

Syntax

```
int empty(s)
```

Parameters

s - string. Can be of type lpstr, str, BSTR, word*, VARIANT.

Remarks

Returns 1 if **s** is empty, or 0 if not.

Added in QM 2.3.0.

Type	s is empty if
lpstr	null (183) or "".
str	The len property is 0.
BSTR	The len property is 0.
word*	null or L"".
VARIANT	vt is VT_BSTR and the len property of bstrVal is 0. Or vt is VT_EMPTY or VT_NULL. Note that this function always returns 1 for other types, even if the value is 0.

Tips

Don't use `if(s=="")` to check if **s** is empty. It will not work if **s** is null. Also it will not work if **s** is not str. Instead use `if(empty(s))`.

See also: [len \(184\)](#)

Example

```
/
function lpstr's

if(empty(s)) ret ;;exit the function if s is empty ("" or 0)
...
```

Convert string to number

Syntax

```
int|long|double val(s [type] [length])
```

Parameters

s - string that contains a number at the beginning.

type - return type: 0 - int, 1 - long, 2 - double. Default: 0. Constant.

length - int variable that receives the number of characters from the beginning of **s** to the end of the part of **s** that contains the number. Receives 0 if **s** does not begin with a number or the number is too big.

Remarks

Returns numeric value of string **s**. Returns 0 if the string does not begin with a number or the number is too big.

The string can contain integer or double (floating-point) [number \(137\)](#). Integer number can be decimal (like 10) or hexadecimal (like 0x1C). Double number can be in form

```
[sign] [digits] [.digits] [(d|D|e|E) [sign] digits]
```

Whole string can begin with spaces or tabs and sign - or +.

Regardless of **type**, the function always parses whole number in all supported formats. For example, val("1.5E3") returns 1500. It uses **type** only to know function's return type and check bounds.

QM 2.3.0. If number is 0x80000000-0xFFFFFFFF when **type** is 0, or 0x8000000000000000-0xFFFFFFFFFFFFFFFF when **type** is 1, the function returns a negative value. In older QM versions would return 0 (number too big).

Tips

To convert number to string, use operator = with a str variable. Example: `int i=5; str s; s=i ;;now s is "5"`.

Example

```
str s1 s2; int i length; double d
s1="-100"
i=val(s1); out i
s2="4.757E2"
d=val(s2 2 length); out d; out length
```

Output:

-100

475.7

7

Get number of lines in string

Syntax

```
int numlines(s)
```

Parameters

s - string.

Remarks

Returns number of lines in **s**. Does not include last empty line. Lines can be delimited by carriage-return/new-line characters (`[]`) or only new-line characters (`[10]`).

See also: [findl \(191\)](#), [getl \(218\)](#), [foreach \(128\)](#), [line break characters \(253\)](#)

Examples

```
out numlines("one[]two")
out numlines("one[]two[] ")
out numlines("")
```

Output:

```
2
2
0
```

Find string in string

Syntax

```
int find(string substring [from] [flags])
```

Parameters

string - string to search in.

substring - substring to find.

from - 0-based character index, from which to start search. Default 0.

flags (247):

1	case insensitive.
---	-------------------

Remarks

Returns 0-based index of first character of **substring** in **string**. If **substring** not found, returns -1.

Tips

To make more precise search, use [findw \(189\)](#) or [findrx \(197\)](#).

Example

```
str s = "Notepad.exe"  
int i = find(s ".exe")  
now i is 7
```

Find string in string as whole word

Syntax

```
int findw(string substring [from] [delim] [flags])
```

Parameters

string - string to search in.

substring - word to find.

from - 0-based character index, from which to start search. Default 0.

delim - [delimiters \(263\)](#).

flags (247):

1	Case insensitive.
64	QM 2.3.3. Match always if substring begins/ends with a delimiter.
0x100	delim is table of delimiters.
0x200	QM 2.3.3. Add blanks to delim .

Remarks

Returns 0-based index of first character of **substring** in **string**. If not found, returns -1. Searches substrings delimited by characters in **delim**.

Tips

If **string** can be multiline, don't forget to include line break characters (`\n`) in **delim**. Or flag 0x200.

See also: [findrx \(197\)](#).

Example

```
str s = "my.exe /p"
int i = findw(s "exe" 0 " .:\\/'"[])
out i ;;3
```

Find n-th token (part, word) of string

Syntax

```
int findt(string [n] [delim] [flags])
```

Parameters

string - string to search in.

n - 0-based index of token. Default: -1.

delim - [delimiters \(263\)](#).

flags (247):

0x100	delim is table of delimiters.
0x200	QM 2.3.3. Add blanks to delim .

Remarks

Returns 0-based index of first character of **n**-th token (substring delimited by characters in **delim**) in **string**. If there are less than **n**+1 tokens, returns -1.

If **n** is omitted or negative, finds next token. It works in same function only.

Tips

To find lines, use [findl \(191\)](#).

See also: [gett \(223\)](#) [findrx \(197\)](#)

Examples

```
find the second token
str s = "my.exe /cl"
int i = findt(s 1 " .:\/'"[])
out i ;;3

for each word
int t
for t 0 1000000000
  i=findt(s -t)
  if(i<0) break
  out i
```

Find n-th line in string

Syntax

```
int findl(string [n])
```

Parameters

string - string to search in.

n - 0-based index of line. Default: -1.

Remarks

Returns 0-based index of first character of **n**-th line in **string**. If there are less than **n**+1 lines, returns -1.

If **n** is omitted or negative, finds next line. It works in same function only.

QM 2.3.3. Fixed bug: does not work if **n** omitted.

See also: [getl \(218\)](#), [numlines. \(187\)](#) [foreach \(128\)](#), [line break characters \(253\)](#)

Examples

```
find the second line (line index 1)
str s = "line0[]line1[]line2"
int i = findl(s 1)
out i ;;7 (5 characters in the first line, and 2 characters in the line break)

for each line
int li
for li 0 1000000000
  i=findl(s -li)
  if(i<0) break
  out i
```

Tokenize (split) string

Syntax

```
int tok(string arr [n] [delim] [flags] [arr2])
```

Parameters

string - string to tokenize. Usually str variable.

arr - receives tokens. Variable of type ARRAY(str) or ARRAY(lpstr). Also can be [pointer-based array \(193\)](#) of str or lpstr. Can be 0 if don't need.

n - max number of tokens required. If omitted or -1, gets all.

delim - [delimiters \(263\)](#).

flags (247):

1	Modify string : substitute first delimiter character after each token to 0. <ul style="list-style-type: none"> It is useful when arr is array of lpstr. string must be of type str, not lpstr.
2	If there are more than n tokens, get whole right part as last (n -1 th) token. <ul style="list-style-type: none"> For example, if string is "a b c" and n is 2, you will get "a" and "b c" instead of "a" and "b".
4	Don't split parts enclosed in " " (double quotation marks). <ul style="list-style-type: none"> For example, <code>tok "a, 'b, c'" a -1 ", '" 4</code> gets "a" and "b, c", not "a", "b", "c".
8	Don't split parts enclosed in ().
16	Don't split parts enclosed in [].
32	Don't split parts enclosed in { }.
64	Don't split parts enclosed in < >.
128	Don't split parts enclosed in ' '.
0x100	delim is table of delimiters.
0x200	QM 2.3.1. Recursive parsing of parts enclosed in ()[]{}<>. <ul style="list-style-type: none"> For example, when parsing string "<a (b > c) d>" with flags 8 64, you would get 3 tokens: "a (b ", "c" and "d". With flags 8 64 0x200 will be 1 token: "a (b > c) d".
0x400	QM 2.3.1. Don't apply this default behavior of parsing parts enclosed in ()[]{}<>: <ol style="list-style-type: none"> Characters)]]> in parts enclosed in "" are ignored. A single character)]]> enclosed in ' ' is ignored.
0x1000	QM 2.3.3. Delimiters are blanks (space, tab, new line, control characters) and delim characters.
0x2000	QM 2.3.5. Always trim blanks around tokens. Also removes blank tokens. <ul style="list-style-type: none"> For example, <code>tok " a , b " a -1 ", " 0x2000</code> gets "a" and "b", not " a " and " b ".

arr2 - array for parts between (after) tokens. Will have same length as **arr**. Can be 0 if don't need.

Remarks

Parses **string** and stores tokens in **arr**. Returns number of tokens.

If **arr** is array of str, it receives copies of tokens. If it is array of lpstr, it receives pointers to tokens within **string**; it is faster.

QM 2.3.5. Applies flags 4-128 even if **delim** does not contain these characters. Then tokens include these characters.

QM 2.3.5. Fixed bug: flags 4-128 ignored when the enclosed part is preceded by a non-delimiter character.

Tips

Although **tok** can be used to get lines of a multiline string, there are simpler ways. See example3, [foreach \(128\)](#), [findl \(191\)](#), [str.getl \(218\)](#).

To parse strings also can be used [regular expressions \(198\)](#) ([findrx \(197\)](#), [str.replacex \(230\)](#)) and other string functions, like [find](#), [findc](#), [findw](#).

Example1

```
str s = "one two three"
ARRAY(str) arr
int i nt
```

```
nt = tok(s arr)
for(i 0 nt) out arr[i]
```

Output:

```
one
two
three
```

Example2

```
str s = "one, (two + three) four five"
ARRAY(str) arr arr2
int i nt
nt = tok(s arr 3 ", ()" 8 arr2)
for(i 0 nt) out "'%s' '%s'" arr[i] arr2[i]
```

Output:

```
'one', ('
'two + three' ) '
'four' ''
```

Example3

```
str s = "one[]two[]three"
ARRAY(str) arr = s
for(int i 0 arr.len) out arr[i]
```

Output:

```
one
two
three
```

Example4

```
str s="abcdef"
int i
Split s into characters as strings:
ARRAY(str) a.create(s.len)
for(i 0 a.len) a[i].get(s i 1)
Split s into characters as character codes:
ARRAY(int) b.create(s.len)
for(i 0 b.len) b[i]=s[i]
```

tok and pointer-based arrays

Function [tok \(192\)](#) splits string and stores tokens (parts of the string) into array **arr**. It can be safe array (ARRAY(str) or ARRAY(lpstr)) or [pointer-based array](#). Using safe array is easier. Using pointer-based array makes [tok](#) faster.

arr can be lpstr array (fastest) or str array. If **arr** is lpstr array, [tok](#) stores pointers to tokens (within **string**) in it. If **arr** is str array - copies tokens to it.

When using pointer-based array, [tok](#) don't know how many elements it has, therefore **n** must be equal or less than number of elements in array. If number of required tokens is unknown, use safe array.

If **n** is negative, [tok](#) populates **-n** elements of array, but it parses whole **string** and returns number of tokens that can be more than **-n**.

Array can be created in various ways. Examples:

1. Safe array:

```
ARRAY(str) arr.create(50)
tok somestring &arr[0] 50
```

2. Declare **n** local variables:

```
str s0 s1 s2 s3
tok somestring &s0 4 " "
```

3. Define variable type with embedded array:

```
type STRARRAY50 str's[50]
LPSTRARRAY50 a
tok somestring &a 50
```

Also can be used some memory allocation function.

Example1

```
str s = "one two three"
ARRAY(str) arr.create(20)
int nt = tok(s &arr[0] arr.len)
for(int'i 0 nt) out arr[i]
Output:
one
two
three
```

Example2

```
type TOK50 $s[50] ~ss[50]
TOK50 t
str s = "func1(57 func2(a b) hoo) 'Some string' r.top"
int nt = tok(s &t 10 "" 1|4|8 &t.ss)
for int'i 0 nt
_out "'%s' '%s'" t.s[i] t.ss[i]
Output:
'func1' '('
'57 func2(a b) hoo' ')'
'Some string' ''
'r' '.'
'top' ''
```

Find character in string

Syntax

```
int findc(string char [from])
int findcr(string char [fromr])
int findcs(string charset [from])
int findcn(string charset [from])
```

Parameters

string - string to search in.

char - character to find. Integer [character code \(239\)](#) or single-character string.

charset - characters to find or skip. String.

from - 0-based character index from which to start search. Default: 0.

fromr - 0-based character index from which to start search towards beginning. Default: -1 (string length).

Remarks

Finds character **char** in **string**. If not found, returns -1.

findc returns 0-based index of **char**.

findcr returns 0-based index of **char**, but searches from right to left.

findcs returns 0-based index of first character that is in **charset**.

findcn returns 0-based index of first character that is not in **charset**.

In [Unicode \(267\)](#) mode, **string** can contain any characters, but **char** must be an ASCII character, and **charset** must contain only ASCII characters. To find an Unicode character, use [find \(188\)](#).

Examples

```
str s = "c:\windows\calc.exe"
int i = findc(s ':')
now i is 1
i = findcr(s '\')
now i is 10
i = findcs(s ":\./." 11)
now i is 15
```

Find string in binary data

Syntax

```
int findb(string substring [len] [from])
```

Parameters

string - string to search in. Must be str variable.

substring - substring to find. Can be string or pointer to binary data.

len - number of bytes in **substring**. Default: if **substring** is string - **substring** length, otherwise this parameter is required.

from - 0-based character index, from which to start search. Default 0.

Remarks

Returns 0-based index of first character of **substring** in **string**. If not found, returns -1. Both strings can contain binary data (null characters).

Example

```
str s="one two"; s[3]=0  
now s contains null character at position 3  
int i=findb(s "e[0]t" 3)  
now i is 2
```

Compare string using wildcard characters

Syntax

```
int matchw(string pattern [flags])
```

Parameters

string - string.

pattern - string with wildcard characters.

flags (247):

1	case insensitive.
---	-------------------

Remarks

Returns 1 if **string** matches **pattern**, or 0 if not.

Wildcard characters:

? matches any single character. Note that in [Unicode \(267\)](#) mode it can be more than 1 byte because non ASCII characters consist of several bytes.

* matches 0 or more characters.

?* matches 1 or more characters.

** matches *.

*? matches ?.

See also: [findrx \(197\)](#), [StrCompare \(114\)](#).

Example

```
str s="file.txt"
if(matchw(s "*.txt"))
...

```

Find or compare string using regular expression

[About regular expressions \(198\)](#)

[Regular expression syntax \(199\)](#)

[str.replacex \(230\)](#)

Syntax

```
int findrx(string pattern [from] [flags] [result] [submatch])
```

or

```
int findrx(string pattern rf [flags] [result] [submatch])
```

Parameters

string - string to search in.

pattern - regular expression that matches substring to find. String.

from - 0-based character index, from which to start search. Default 0.

rf - variable of type [FINDRX \(202\)](#). You can use it to set the part of **string** where to search, and a callout callback.

flags (247):

1	Case insensitive.
2	Whole word. This adds \b to the beginning and end of pattern .
4	Find all. Valid only if result is array.
8	Multiline. If this flag is set (or (?m) is used in pattern), ^ and \$ match the beginning and end of line. Default: ^ and \$ match the beginning and end of whole string.
16	Don't need submatches. This flag makes this function faster when result is array.
32	QM 2.3.0. Convert pattern from UTF-8 to ANSI. Used when QM is running in Unicode mode (ignored otherwise). Set this flag if pattern contains non ASCII (239) characters, but string is ANSI (not UTF-8). It is needed because these characters in pattern normally consist of 2 or 3 bytes, whereas characters in string consist of 1 byte.
128	Only compile pattern .
pcre flags (200)	

result - variable of type str, int, ARRAY(str) or ARRAY(CHARRANGE).

submatch - submatch to find. Integer. If 0 (default), finds whole match. Before QM 2.4.3 - not used if **result** is array or 0.

Remarks

Finds a substring in **string**. To specify the substring, is used regular expression (**pattern**). The function can find a whole match, a submatch, or all matches and submatches. A *match* is the part of **string** that matches **pattern**. A *submatch* is the part of the match that matches a captured subpattern. A captured subpattern is the part of pattern that is enclosed in parentheses and does not begin with ?.

The return value depends on flags and other arguments:

default	0-based index of first character of the match in string , or -1 if not found.
nonzero submatch	0-based index of first character of the submatch in string , or -1 if not found.
flag 4	number of matches, or 0 if not found.
flag 128	not used.

result can be used to get more information about the found match and submatches. This table shows what the function stores in **result** variable depending on its type. Assume that flag 4 is not used.

str	result receives the match or submatch (if submatch is nonzero). If flag 128, receives the compiled pattern.
int	result receives length of the match or submatch.
ARRAY(str)	result receives the match in element 0 and submatches in subsequent elements. If flag 16 - only match. QM 2.4.3: if submatch not 0 - only the submatch.
ARRAY(CHARRANGE)	result receives start and end offsets of the match and submatches. If flag 16 - only match. QM 2.4.3: if submatch not 0, receives only the submatch. QM 2.4.3: ARRAY(POINT) can be used too (POINT is the same as CHARRANGE, just shorter member names).

The CHARRANGE type is used to store start and end positions of a substring in a string.

```
type CHARRANGE cpMin cpMax
```

cpMin - start of substring (match or submatch) in string. It is 0-based index of first character of substring in string.

cpMax - end of substring.

If flag 4 is set and **result** is array, finds all matches. It creates two-dimensional array. To access an element, use `result[x y]`, where y is match index (0 - first match, 1 - second match, ...), and x is 0 or submatch index (0 - whole match, 1 - first submatch, ...). For example, `result[0 0]` contains first match, `result[0 1]` - second match, `result[1 0]` - first submatch of first match.

If flag 128 (only compile) is set, and **result** is str variable, the function does not search. It only compiles **pattern** and stores compiled data into **result** variable. You can use that variable later with functions `findrx` and `str.replacex` as **pattern**. If multiple operations are performed with the same pattern, using compiled pattern is about 2 times faster, because then pattern does not have to be compiled each time. To compile pattern, are used only **pattern**, **flags** and **result**. You should use same **flags** value when compiling and later.

Examples

Find digits (10)

```
str subject="abc10 100 def"
out findrx(subject "\d+")
```

Find digits as whole word (100), and store into s

```
str subject="abc10 100 def"
str s
if(findrx(subject "\d+" 0 2 s)>=0) out s
```

Extract HTML tags (simplified; useful only as "find all" example)

```
str html
IntGetFile("http://www.google.com" html)
str pattern="<(.*?)>.*<\/\1>" ;;matches a HTML tag
ARRAY(str) a
findrx(html pattern 0 4 a)
int i
for(i 0 a.len)
_out "submatch=%s, whole=%s" a[1 i] a[0 i]
```

Extract URL components

```
str subject="http://msdn.microsoft.com:80/scripting/default.htm"
str pattern="(\w+):\/\:\/\/([^\:]+)(:\d*)?([^\# ]*)"
int i; ARRAY(str) a
if(findrx(subject pattern 0 0 a)<0) out "does not match"; ret
for i 0 a.len
_out a[i]
```

Extract URL components; show offsets and lengths

```
str subject="http://msdn.microsoft.com:80/scripting/default.htm"
str pattern="(\w+):\/\:\/\/([^\:]+)(:\d*)?([^\# ]*)"
int i; ARRAY(CHARRANGE) a
if(findrx(subject pattern 0 0 a)<0) out "does not match"; ret
for i 0 a.len
_int offset(a[i].cpMin) length(a[i].cpMax-a[i].cpMin)
_str s.get(subject offset length)
_out "offset=%i length=%i %s" offset length s
```

Extract only server from URL

```
str subject="http://msdn.microsoft.com:80/scripting/default.htm"
str pattern="(\w+):\/\:\/\/([^\:]+)(:\d*)?([^\# ]*)"
str server
if(findrx(subject pattern 0 0 server 2)>=0) out server
```

Regular expressions

Reference (199)

Regular expressions are used to find, find-replace and split strings. QM has two functions that use regular expressions: [findrx \(197\)](#) and [str.replacex \(230\)](#). Regular expressions also can be used with commands and functions that use [window name \(274\)](#), with [window triggers \(32\)](#), and in the Find dialog.

A regular expression is a pattern of text that consists of ordinary characters (for example, letters a through z) and special characters, known as *metacharacters*. The pattern describes one or more strings to match when searching text. It is like a string containing wildcard characters * and ? that you use when searching for files, but regular expressions are much more powerful. The following table contains some metacharacters and their behavior in the context of regular expressions.

Character	Description
\	Marks the next character as either a special character, a literal (removes special meaning from next metacharacter), a backreference, anchor, or an octal hex escape. For example, 't' matches the character "t". '\t' matches a tab character. The sequence '\\' matches "\", and \" matches \".
.	Matches any single character except new line character (\n). If you use '(?s)' somewhere before, then . matches all characters. Note: A Windows new line consists of two characters - carriage return followed by new line (\r\n). Since . matches carriage return (\r), sometimes you may get unexpected results. Then use '[^r\n]' instead of .
[xyz]	A character set. Matches any one of the enclosed characters. For example, '[abc]' matches the 'a' in "plain". Note: [digits] in strings is a QM escape sequence (137) and is replaced to a character code. If you need literal [digits] in regular expression, use [91] instead of [. For example, for [135] use [91]135]. Escape sequences are processed before processing regular expression.
[^xyz]	A negative character set. Matches any character not enclosed. For example, '[^abc]' matches the 'p' in "plain".
[a-z]	A range of characters. Matches any character in the specified range. For example, '[a-z]' matches any lowercase alphabetic character in the range 'a' through 'z'.
[^a-z]	A negative range characters. Matches any character not in the specified range. For example, '[^a-z]' matches any character not in the range 'a' through 'z'.
*	Matches the preceding subexpression zero or more times. For example, 'zo*' matches "z" and "zoo".
+	Matches the preceding subexpression one or more times. For example, 'zo+' matches "zo" and "zoo", but not "z".
?	Matches the preceding subexpression zero or one time. For example, "do(es)?" matches the "do" in "do" or "does".
{n}	Matches exactly <i>n</i> times (<i>n</i> is a nonnegative integer). For example, 'o{2}' does not match the 'o' in "Bob," but matches the two o's in "food".
{n,}	Matches at least <i>n</i> times. For example, 'o{2,}' does not match the "o" in "Bob" and matches all the o's in "fooooo".
{n,m}	Matches at least <i>n</i> and at most <i>m</i> times. For example, "o{1,3}" matches the first three o's in "fooooo". Note that you cannot put a space between the comma and the numbers.
?	When this character immediately follows any of the other quantifiers (*, +, ?, { <i>n</i> }, { <i>n</i> , }, { <i>n</i> , <i>m</i> }), the matching pattern is non-greedy. A non-greedy pattern matches as little of the searched string as possible, whereas the default greedy pattern matches as much of the searched string as possible. For example, in the string "oooo", 'o+?' matches a single "o", while 'o+' matches all 'o's.
^	Matches the position at the beginning of the input string. If multiline flag is set, ^ also matches the position at beginning of line.
\$	Matches the position at the end of the input string. If multiline flag is set, \$ also matches the position at end of line.
\b	Matches a word boundary, that is, the position between a word and a space (or other delimiter, e.g. comma). For example, 'er\b' matches the 'er' in "never" but not the 'er' in "verb".
\B	Matches a nonword boundary. 'er\B' matches the 'er' in "verb" but not the 'er' in "never".
(pattern)	Matches <i>pattern</i> and captures (remembers) the match ("submatch"). To match parentheses characters (), use \"(' or \").
(?:pattern)	Matches <i>pattern</i> but does not capture the match, that is, it is a non-capturing match that is not stored for possible later use. This is useful for combining parts of a pattern with the "or" character (). For example, 'industr(?:y ies)' is a more economical expression than 'industry industries'.
(?pattern)	Positive lookahead matches the search string at any point where a string matching <i>pattern</i> begins. This is a

198. Regular expressions

=pattern)	non-capturing match, that is, the match is not captured for possible later use. For example 'Windows (?=95 98 NT 2000)' matches "Windows" in "Windows 2000" but not "Windows" in "Windows 3.1". Lookaheads do not consume characters, that is, after a match occurs, the search for the next match begins immediately following the last match, not after the characters that comprised the lookahead.
(?!pattern)	Negative lookahead matches the search string at any point where a string not matching <i>pattern</i> begins. This is a non-capturing match, that is, the match is not captured for possible later use. For example 'Windows (?!95 98 NT 2000)' matches "Windows" in "Windows 3.1" but does not match "Windows" in "Windows 2000". Lookaheads do not consume characters, that is, after a match occurs, the search for the next match begins immediately following the last match, not after the characters that comprised the lookahead.
x y	Matches either <i>x</i> or <i>y</i> . For example, 'z food' matches "z" or "food". 'colo(r ur)' matches "color" or "colour".
\cx	Matches the control character indicated by <i>x</i> . For example, \cM matches a Control-M or carriage return character. The value of <i>x</i> must be in the range of A-Z or a-z. If not, <i>c</i> is assumed to be a literal 'c' character.
\d	Matches a digit character. Equivalent to [0-9].
\D	Matches a nondigit character. Equivalent to [^0-9].
\f	Matches a form-feed character. Equivalent to \x0c and \cL.
\n	Matches a newline character. Equivalent to \x0a and \cJ.
\r	Matches a carriage return character. Equivalent to \x0d and \cM.
\s	Matches any whitespace character including space, tab, form-feed, etc. Equivalent to [\f\n\r\t\v].
\S	Matches any non-whitespace character. Equivalent to [^\f\n\r\t\v].
\t	Matches a tab character. Equivalent to \x09 and \cI.
\v	Matches a vertical tab character. Equivalent to \x0b and \cK.
\w	Matches any word character including underscore. Equivalent to '[A-Za-z0-9_]'.
\W	Matches any nonword character. Equivalent to '[^A-Za-z0-9_]'.
\xn	Matches <i>n</i> , where <i>n</i> is a hexadecimal escape value. Hexadecimal escape values must be exactly two digits long. For example, '\x41' matches "A". '\x041' is equivalent to '\x04' & "1". Allows ASCII codes to be used in regular expressions.
\num	Matches <i>num</i> , where <i>num</i> is a positive integer. A reference back to captured matches. For example, '(.)\1' matches two consecutive identical characters.
\n	Identifies either an octal escape value or a backreference. If \n is preceded by at least <i>n</i> captured subexpressions, <i>n</i> is a backreference. Otherwise, <i>n</i> is an octal escape value if <i>n</i> is an octal digit (0-7).
\nm	Identifies either an octal escape value or a backreference. If \nm is preceded by at least <i>nm</i> captured subexpressions, <i>nm</i> is a backreference. If \nm is preceded by at least <i>n</i> captures, <i>n</i> is a backreference followed by literal <i>m</i> . If neither of the preceding conditions exist, \ nm matches octal escape value <i>nm</i> when <i>n</i> and <i>m</i> are octal digits (0-7).
\nml	Matches octal escape value <i>nml</i> when <i>n</i> is an octal digit (0-3) and <i>m</i> and <i>l</i> are octal digits (0-7).

Tutorials

Shows how to use function `findrx()` to find and extract text.

`findrx()` is similar to `find()`.

```
int i
i=find("Sunday Monday Tuesday" "Monday") ;;find Monday
out i
i=findrx("Sunday Monday Tuesday" "Monday") ;;find Monday
out i
```

But `findrx()` can find not only exact text.

The second argument (regular expression) can contain special characters (metacharacters) that match certain characters or conditions.

```
i=findrx("file578.txt" "\d+") ;;find a number. Here \d matches a digit, and + means "one or more". So, \d+
matches one or more digits.
out i
```

How to extract the found text?

```
str s
i=findrx("file578.txt" "\d+" 0 0 s) ;;pass a str variable as 5-th argument, and it will be populated with the
match
out i
out s
```

What if not found?

```
i=findrx("file.txt" "\d+") ;;if not found, returns -1
if(i<0) out "not found"
```

When whole string must match:

```
i=findrx("578" "^\d+$") ;;here ^ and $ mean "beginning" and "end"
out i
i=findrx("file578.txt" "^\d+$") ;;does not match
out i
```

Find a number where it is whole word:

```
i=findrx("file123 456.txt" "\b\d+\b" 0 0 s) ;;here \b means "word boundary". Word characters are
alphanumeric ASCII characters and _.
```

```
out i
out s
```

Find a date in a filename:

```
i=findrx("file01-02-2000.txt" "\d{2}-\d{2}-\d{4}" 0 0 s) ;;here \d{2} means "2 digits"
out i
out s
```

If need only year:

```
i=findrx("file01-02-2000.txt" "\d{2}-\d{2}-(\d{4})" 0 0 s 1) ;;6-th argument tells which submatch
(part enclosed in ()) to get into s
```

```
out i
out s
```

If need month, day and year:

```
ARRAY(str) a
i=findrx("file01-02-2000.txt" "(\d{2})-(\d{2})-(\d{4})" 0 0 a)
```

```
out a[0] ;;whole match
out a[1] ;;submatch 1 (month)
out a[2] ;;submatch 2 (day)
out a[3] ;;submatch 3 (year)
```

Find substring consisting of specified characters:

```
i=findrx("xxx 0x3ea5 yyy" "0x[\dabcdef]+" 0 0 s) ;;here [] is used to specify characters, \d is digit, +
means "1 or more". So it finds 0x followed by 1 or more digits and characters abcdef.
```

```
out i
out s
```

How to use `\^$()[].+?` and other metacharacters as ordinary characters? (all metacharacters are listed in "Regular expression syntax" topic in QM Help)

```
i=findrx("file.txt" ".+\.txt" 0 0 s) ;;insert \ before each such character. Here the first . means "any
character", however the second . is just . because preceded by \.
```

```
out s
i=findrx("xxx file.txt yyy" "\Qfile.txt\E" 0 0 s) ;;or enclose part of regular expression in \Q and \E
out s
```

Shows how to use function `str.replacrx()` to find-replace text.

`str.replacrx()` is similar to `str.findreplace()`.

```
str x="Sunday Monday Tuesday"
x.findreplace("Monday" "(here was Monday)")
out x
str y="Sunday Monday Tuesday"
y.replacrx("Monday" "(here was Monday)")
out y
```

But it can find not only exact text. It can find text like `findrx()`.

```
y="file123 456.txt"
y.replacrx("\d+" "(here was a number)")
out y
```

What if not found?

```
y="file.txt"
int n=y.replacerx("\d+" "(here was a number)") ;;replacerx returns number of found matches
if(n=0) out "not found"
```

How to use submatches in the replacement string?

```
y="file01-02-2000.txt"
y.replacerx("(\d{2})-(\d{2})-(\d{4})" "$3-$1-$2") ;;in the replacement string, $number can be used for a
submatch
out y ;;converted date from mm-dd-yyyy to yyyy-mm-dd format
```

On the web:

[Tutorial](#) (VBScript regular expressions)

[More info, often used regular expressions, etc](#)

[Regular expression tools](#)

PCRE

Regular expression support is provided by the [PCRE library](#), which is open source software, written by Philip Hazel, and copyright by the University of Cambridge, England.

The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl, with just a few differences. QM uses PCRE version 4.4. It corresponds approximately with Perl 5.8.

UTF-8

When QM is running in Unicode mode, it stores text in UTF-8 format. When text is in UTF-8 format, non [ASCII \(239\)](#) characters consist of 2-4 bytes.

QM does not support UTF-8 mode (option RX_UTF8) in regular expressions. Non ASCII characters can be used, but they are interpreted as 2-4 separate bytes, not as single character. For example, . matches single byte, not whole character. Also, with non ASCII characters are not recognized word boundaries (\b), character classes (\w, \s, etc), not supported case insensitivity, etc.

In most cases all this will not be a problem. However in some cases, when there are non ASCII characters in regular expression or subject string, you have to build the regular expression differently, and maybe use [flag 32 \(197\)](#).

Regular expression syntax

- [PCRE REGULAR EXPRESSION DETAILS](#)
- [BACKSLASH](#) \
- [CIRCUMFLEX AND DOLLAR](#) ^\$
- [DOT](#) .
- [MATCHING A SINGLE BYTE](#)
- [SQUARE BRACKETS](#) []
- [POSIX CHARACTER CLASSES](#)
- [VERTICAL BAR](#) |
- [INTERNAL OPTION SETTING](#)
- [SUBPATTERNS](#)
- [NAMED SUBPATTERNS](#)
- [REPETITION](#)
- [ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS](#)
- [BACK REFERENCES](#)
- [ASSERTIONS](#)
- [CONDITIONAL SUBPATTERNS](#)
- [COMMENTS](#)
- [RECURSIVE PATTERNS](#)
- [SUBPATTERNS AS SUBROUTINES](#)
- [CALLOUTS](#)

See also:

- [findrx \(197\)](#), [str.replacex \(230\)](#)
- [Regular expression options \(flags\) \(200\)](#)
- [About regular expressions; code examples \(198\)](#)
- [Regular expressions in Unicode mode \(UTF-8\) \(198\)](#)

PCRE REGULAR EXPRESSION DETAILS

PCRE - Perl-compatible regular expressions. The syntax and semantics of the regular expressions supported by PCRE are described below. Regular expressions are also described in the Perl documentation and in a number of other books, some of which have copious examples. Jeffrey Friedl's "Mastering Regular Expressions", published by O'Reilly, covers them in great detail. The description here is intended as reference documentation.

The basic operation of PCRE is on strings of bytes. However, there is also support for UTF-8 character strings. To use this support, call regular expression functions with the `RX_UTF8` option. How this affects the pattern matching is mentioned in several places below.

Note: QM does not support UTF-8 mode (option `RX_UTF8`) in regular expressions. [Read more \(198\)](#).

A regular expression is a pattern that is matched against a subject string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the subject. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a subject string that is identical to itself. The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of *meta-characters*, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of meta-characters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the meta-characters are as follows:

\	general escape character with several uses
^	assert start of string (or line, in multiline mode)
\$	assert end of string (or line, in multiline mode)
.	match any character except newline (by default)
[start character class definition
	start of alternative branch
(start subpattern
)	end subpattern
?	extends the meaning of (
	also 0 or 1 quantifier
	also quantifier minimizer
*	0 or more quantifier

```

+      1 or more quantifier
      also "possessive quantifier"
{      start min/max quantifier

```

Part of a pattern that is in square brackets is called a "character class". In a character class the only meta-characters are:

```

\      general escape character
^      negate the class, but only if the first character
-      indicates character range
[      POSIX character class (only if followed by POSIX
      syntax)
]      terminates the character class

```

The following sections describe the use of each of the meta-characters.

BACKSLASH \

The backslash character has several uses.

Firstly, if it is followed by a non-alphameric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a `*` character, you write `*` in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a meta-character, so it is always safe to precede a non-alphameric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write `\\`.

If a pattern is compiled with the `RX_EXTENDED` option, whitespace in the pattern (other than in a character class) and characters between a `#` outside a character class and the next newline character are ignored. An escaping backslash can be used to include a whitespace or `#` character as part of the pattern.

If you want to remove the special meaning from a sequence of characters, you can do so by putting them between `\Q` and `\E`. This is different from Perl in that `$` and `@` are handled as literals in `\Q...\E` sequences in PCRE, whereas in Perl, `$` and `@` cause variable interpolation. Note the following examples:

Pattern	PCRE matches	Perl matches
<code>\Qabc\$xyz\E</code>	<code>abc\$xyz</code>	<code>abc</code> followed by the contents of <code>\$xyz</code>
<code>\Qabc\ \$xyz\E</code>	<code>abc\ \$xyz</code>	<code>abc\ \$xyz</code>
<code>\Qabc\E\ \$\Qxyz\E</code>	<code>abc\$xyz</code>	<code>abc\$xyz</code>

The `\Q...\E` sequence is recognized both inside and outside character classes.

See also: [QM escape sequences \(137\)](#).

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

```

\a      alarm, that is, the BEL character (hex 07)
\cx     "control-x", where x is any character
\e      escape (hex 1B)
\f      formfeed (hex 0C)
\n      newline (hex 0A)
\r      carriage return (hex 0D)
\t      tab (hex 09)
\ddd    character with octal code ddd, or backreference
\xhh    character with hex code hh
\x{hhh..} character with hex code hhh... (UTF-8 mode only)

```

The precise effect of `\cx` is as follows: if `x` is a lower case letter, it is converted to upper case. Then bit 6 of the character (hex 40) is inverted. Thus `\cz` becomes hex 1A, but `\c{` becomes hex 3B, while `\c;` becomes hex 7B.

After `\x`, from zero to two hexadecimal digits are read (letters can be in upper or lower case). In UTF-8 mode, any number of hexadecimal digits may appear between `\x{` and `}`, but the value of the character code must be less than 2^{31} (that is, the maximum hexadecimal value is 7FFFFFFF). If characters other than hexadecimal digits appear between `\x{` and `}`, or if there is no terminating `}`, this form of escape is not recognized. Instead, the initial `\x` will be interpreted as a basic hexadecimal

escape, with no following digits, giving a byte whose value is zero.

Characters whose value is less than 256 can be defined by either of the two syntaxes for `\x` when PCRE is in UTF-8 mode. There is no difference in the way they are handled. For example, `\xdc` is exactly the same as `\x{dc}`.

After `\0` up to two further octal digits are read. In both cases, if there are fewer than two digits, just those that are present are used. Thus the sequence `\0\x07` specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, PCRE reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a *back reference*. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, PCRE re-reads up to three octal digits following the backslash, and generates a single byte from the least significant 8 bits of the value. Any subsequent digits stand for themselves. For example:

```
\040    is another way of writing a space
\40     is the same, provided there are fewer than 40
        previous capturing subpatterns
\7      is always a back reference
\11     might be a back reference, or another way of
        writing a tab
\011    is always a tab
\0113   is a tab followed by the character "3"
\113    might be a back reference, otherwise the
        character with octal code 113
\377    might be a back reference, otherwise
        the byte consisting entirely of 1 bits
\81     is either a back reference, or a binary zero
        followed by the two characters "8" and "1"
```

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single byte value or a single UTF-8 character (in UTF-8 mode) can be used both inside and outside character classes. In addition, inside a character class, the sequence `\b` is interpreted as the backspace character (hex 08). Outside a character class it has a different meaning (see below).

The third use of backslash is for specifying generic character types:

```
\d      any decimal digit
\D      any character that is not a decimal digit
\s      any whitespace character
\S      any character that is not a whitespace character
\w      any "word" character
\W      any "non-word" character
```

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.

In UTF-8 mode, characters with values greater than 255 never match `\d`, `\s`, or `\w`, and always match `\D`, `\S`, and `\W`.

For compatibility with Perl, `\s` does not match the VT character (code 11). This makes it different from the POSIX "space" class. The `\s` characters are HT (9), LF (10), FF (12), CR (13), and space (32).

A "word" character is any letter or digit or the underscore character, that is, any character which can be part of a Perl "word". The definition of letters and digits is controlled by PCRE's character tables, and may vary if locale-specific matching is taking place. For example, in the "fr" (French) locale, some character codes greater than 128 are used for accented letters, and these are matched by `\w`.

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the subject string, all of them fail, since there is no character to match.

The fourth use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the subject string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are

```

\b    matches at a word boundary
\B    matches when not at a word boundary
\A    matches at start of subject
\Z    matches at end of subject or before newline at end
\z    matches at end of subject
\G    matches at first matching position in subject

```

These assertions may not appear in character classes (but note that `\b` has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the subject string where the current character and the previous character do not both match `\w` or `\W` (i.e. one matches `\w` and the other matches `\W`), or the start or end of the string if the first or last character matches `\w`, respectively.

The `\A`, `\Z`, and `\z` assertions differ from the traditional circumflex and dollar (described below) in that they only ever match at the very start and end of the subject string, whatever options are set. Thus, they are independent of multiline mode.

They are not affected by the `RX_NOTBOL` or `RX_NOTEOL` options. If the **from** argument is non-zero, indicating that matching is to start at a point other than the beginning of the subject, `\A` can never match. The difference between `\Z` and `\z` is that `\Z` matches before a newline that is the last character of the string as well as at the end of the string, whereas `\z` matches only at the end.

The `\G` assertion is true only when the current matching position is at the start point of the match, as specified by the **from** argument. It differs from `\A` when the value of *startoffset* is non-zero. By calling the function multiple times with appropriate arguments, you can mimic Perl's `/g` option, and it is in this kind of implementation where `\G` can be useful.

Note, however, that PCRE's interpretation of `\G`, as the start of the current match, is subtly different from Perl's, which defines it as the end of the previous match. In Perl, these can be different when the previously matched string was empty. Because PCRE does just one match at a time, it cannot reproduce this behavior.

If all the alternatives of a pattern begin with `\G`, the expression is anchored to the starting match position, and the "anchored" flag is set in the compiled regular expression.

CIRCUMFLEX AND DOLLAR ^ \$

Outside a character class, in the default matching mode, the circumflex character is an assertion which is true only if the current matching point is at the start of the subject string. If the **from** argument is non-zero, circumflex can never match if the `RX_MULTILINE` option is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the subject, it is said to be an "anchored" pattern. (There are also other constructs that can cause a pattern to be anchored.)

A dollar character is an assertion which is true only if the current matching point is at the end of the subject string, or immediately before a newline character that is the last character in the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the `RX_DOLLAR_ENDONLY` option at compile time. This does not affect the `\Z` assertion.

The meanings of the circumflex and dollar characters are changed if the `RX_MULTILINE` option is set. When this is the case, they match immediately after and immediately before an internal newline character, respectively, in addition to matching at the start and end of the subject string. For example, the pattern `/^abc$/` matches the subject string "defnabc" in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with `^` are not anchored in multiline mode, and a match for circumflex is possible when the **from** argument is non-zero. The `RX_DOLLAR_ENDONLY` option is ignored if `RX_MULTILINE` is set.

Note that the sequences `\A`, `\Z`, and `\z` can be used to match the start and end of the subject in both modes, and if all branches of a pattern start with `\A` it is always anchored, whether `RX_MULTILINE` is set or not.

Usually, end of line includes carriage return character (`\r\n` instead of `\n` only). In QM, `$` and `\Z` match in both cases.

DOT

Outside a character class, a dot in the pattern matches any one character in the subject, including a non-printing character, but not (by default) newline. If the `RX_DOTALL` option is set, dots match newlines as well. The handling of dot is entirely independent of the handling of circumflex and dollar (`^` `$`), the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

MATCHING A SINGLE BYTE

Outside a character class, the escape sequence `\C` matches any one byte, both in and out of UTF-8 mode. Unlike a dot, it always matches a newline. The feature is provided in Perl in order to match individual bytes in UTF-8 mode. Because it breaks up UTF-8 characters into individual bytes, what remains in the string may be a malformed UTF-8 string. For this reason it is best avoided.

PCRE does not allow `\C` to appear in lookbehind assertions (see below), because in UTF-8 mode it makes it impossible to calculate the length of the lookbehind.

SQUARE BRACKETS []

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the subject. In UTF-8 mode, the character may occupy more than one byte. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the subject character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.

For example, the character class `[aeiou]` matches any lower case vowel, while `^[aeiou]` matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters which are in the class by enumerating those that are not. It is not an assertion: it still consumes a character from the subject string, and fails if the current pointer is at the end of the string.

In UTF-8 mode, characters with values greater than 255 can be included in a class as a literal string of bytes, or by using the `\x{}` escaping mechanism.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless `[aeiou]` matches "A" as well as "a", and a caseless `^[aeiou]` does not match "A", whereas a careful version would. PCRE does not support the concept of case for characters with values greater than 255.

The newline character is never treated in any special way in character classes, whatever the setting of the `RX_DOTALL` or `RX_MULTILINE` options is. A class such as `^[a]` will always match a newline.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, `[d-m]` matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as `[W-]46]` is interpreted as a class of two characters ("W" and "-") followed by a literal string "46]", so it would match "W46]" or "-46]". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so `[W-\]46]` is interpreted as a single class containing a range followed by two separate characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example `[\000-\037]`. In UTF-8 mode, ranges can include characters whose values are greater than 255, for example `[\x{100}-\x{2ff}]`.

If a range that includes letters is used when caseless matching is set, it matches the letters in either case. For example, `[W-c]` is equivalent to `[[\^_`wxyzabc]`, matched caselessly, and if character tables for the "fr" locale are in use, `[\xc8-\xcb]` matches accented E characters in both cases.

The character types `\d`, `\D`, `\s`, `\S`, `\w`, and `\W` may also appear in a character class, and add the characters that they match to the class. For example, `[\dABCDEF]` matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class `^[^W_]` matches any letter or digit, but not underscore.

All non-alphanumeric characters other than `\`, `-`, `^` (at the start) and the terminating `]` are non-special in character classes, but it does no harm if they are escaped.

POSIX CHARACTER CLASSES

Perl supports the POSIX notation for character classes, which uses names enclosed by `[:` and `:]` within the enclosing square brackets. PCRE also supports this notation. For example,

```
[01[:alpha:]]%
```

matches `"0"`, `"1"`, any alphabetic character, or `"%"`. The supported class names are

<code>alnum</code>	letters and digits
<code>alpha</code>	letters
<code>ascii</code>	character codes 0 - 127
<code>blank</code>	space or tab only
<code>cntrl</code>	control characters
<code>digit</code>	decimal digits (same as <code>\d</code>)
<code>graph</code>	printing characters, excluding space
<code>lower</code>	lower case letters
<code>print</code>	printing characters, including space
<code>punct</code>	printing characters, excluding letters and digits
<code>space</code>	white space (not quite the same as <code>\s</code>)
<code>upper</code>	upper case letters
<code>word</code>	"word" characters (same as <code>\w</code>)
<code>xdigit</code>	hexadecimal digits

The "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to `\s`, which does not include VT (for Perl compatibility).

The name "word" is a Perl extension, and "blank" is a GNU extension from Perl 5.8. Another Perl extension is negation, which is indicated by a `^` character after the colon. For example,

```
[12[:^digit:]]
```

matches `"1"`, `"2"`, or any non-digit. PCRE (and Perl) also recognize the POSIX syntax `[.ch.]` and `[=ch=]` where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

In UTF-8 mode, characters with values greater than 255 do not match any of the POSIX character classes.

VERTICAL BAR |

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```

matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

INTERNAL OPTION SETTING

The settings of the `RX_CASELESS`, `RX_MULTILINE`, `RX_DOTALL`, and `RX_EXTENDED` options can be changed from within the pattern by a sequence of Perl option letters enclosed between `"(?"` and `")"`. The option letters are

<code>i</code>	for <code>RX_CASELESS</code>
<code>m</code>	for <code>RX_MULTILINE</code>
<code>s</code>	for <code>RX_DOTALL</code>
<code>x</code>	for <code>RX_EXTENDED</code>

For example, `(?im)` sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unsetting such as `(?im-sx)`, which sets `RX_CASELESS` and `RX_MULTILINE` while

unsetting `RX_DOTALL` and `RX_EXTENDED`, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows. If the change is placed right at the start of a pattern, PCRE extracts it into the global options.

An option change within a subpattern affects only that part of the current pattern that follows it, so

```
(a(?i)b)c
```

matches `abc` and `aBc` and no other strings (assuming `RX_CASELESS` is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
(a(?i)b|c)
```

matches `"ab"`, `"aB"`, `"c"`, and `"C"`, even though when matching `"C"` the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behavior otherwise.

The PCRE-specific options `RX_UNGREEDY` and `RX_EXTRA` can be changed in the same way as the Perl-compatible options by using the characters `U` and `X` respectively. The `(?X)` flag setting is special in that it must always occur earlier in the pattern than any of the additional features it turns on, even when it is at top level. It is best put at the start.

SUBPATTERNS

Subpatterns are delimited by parentheses (round brackets), which can be nested. Marking part of a pattern as a subpattern does two things:

1. It localizes a set of alternatives. For example, the pattern

```
cat(aract|erpillar|)
```

matches one of the words `"cat"`, `"cataract"`, or `"caterpillar"`. Without the parentheses, it would match `"cataract"`, `"erpillar"` or the empty string.

2. It sets up the subpattern as a capturing subpattern (as defined above). When the whole pattern matches, it is that portion of the subject string that matched the subpattern. Opening parentheses are counted from left to right (starting from 1) to obtain the numbers of the capturing subpatterns.

For example, if the string `"the red king"` is matched against the pattern

```
the ((red|white) (king|queen))
```

the captured substrings are `"red king"`, `"red"`, and `"king"`, and are numbered 1, 2, and 3, respectively.

The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string `"the white queen"` is matched against the pattern

```
the ((?:red|white) (king|queen))
```

the captured substrings are `"white queen"` and `"queen"`, and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535, and the maximum depth of nesting of all subpatterns, both capturing and non-capturing, is 200.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the `"?"` and the `":"`. Thus the two patterns

```
(?i:saturday|sunday)
(?: (?i) saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match `"SUNDAY"` as well as `"Saturday"`.

NAMED SUBPATTERNS

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with the difficulty, PCRE supports the naming of subpatterns, something that Perl does not provide. The Python syntax (?P<name>...) is used. Names consist of alphanumeric characters and underscores, and must be unique within a pattern.

Named capturing parentheses are still allocated numbers as well as names.

REPETITION

Repetition is specified by quantifiers, which can follow any of the following items:

- a literal data character
- the . metacharacter
- the \C escape sequence
- escapes such as \d that match single characters
- a character class
- a back reference (see next section)
- a parenthesized subpattern (unless it is an assertion)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

```
z{2,4}
```

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

```
[aeiou]{3,}
```

matches at least 3 successive vowels, but may match many more, while

```
\d{8}
```

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {,6} is not a quantifier, but a literal string of four characters.

In UTF-8 mode, quantifiers apply to UTF-8 characters rather than to individual bytes. Thus, for example, \x{100}{2} matches two UTF-8 characters, each of which is represented by a two-byte sequence.

The quantifier {0} is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience (and historical compatibility) the three most common quantifiers have single-character abbreviations:

- * is equivalent to {0,}
- + is equivalent to {1,}
- ? is equivalent to {0,1}

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

```
(a?)*
```

Earlier versions of Perl and PCRE used to give an error at compile time for such patterns. However, because there are cases where this can be useful, such patterns are now accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are "greedy", that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between the sequences /* and */ and within the sequence, individual * and / characters may appear. An attempt to match C comments by applying the pattern

```
/\*. *\*/
```

to the string

```
/* first command */ not comment /* second comment */
```

fails, because it matches the entire string owing to the greediness of the `.*` item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*. *?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the `RX_UNGREEDY` option is set (an option which is not available in Perl), the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behavior.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more store is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with `.*` or `{0,}` and the `RX_DOTALL` option (equivalent to Perl's `/s`) is set, thus allowing the `.` to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the subject string, so there is no point in retrying the overall match at any position after the first. PCRE normally treats such a pattern as though it were preceded by `\A`.

In cases where it is known that the subject string contains no newlines, it is worth setting `RX_DOTALL` in order to obtain this optimization, or alternatively using `^` to indicate anchoring explicitly.

However, there is one situation where the optimization cannot be used. When `.*` is inside capturing parentheses that are the subject of a backreference elsewhere in the pattern, a match at the start may fail, and a later one succeed. Consider, for example:

```
(.*)abc\1
```

If the subject is "xyz123abc123" the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[dume]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|(b))+/
```

matches "aba" the value of the second captured substring is "b".

ATOMIC GROUPING AND POSSESSIVE QUANTIFIERS

With both maximizing and minimizing repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern `\d+foo` when applied to the subject line

```
123456bar
```

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the `\d+` item, and then with 4, and so on, before ultimately failing. "Atomic grouping" (a term taken from Jeffrey Friedl's book) provides the means for specifying that once a subpattern has matched, it is not to be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher would give up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with `(?>` as in this example:

```
(?>\d+)bar
```

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the subject string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both `\d+` and `\d+?` are prepared to adjust the number of digits they match in order to make the rest of the pattern match, `(?>\d+)` can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional `+` character following a quantifier. Using this notation, the previous example can be rewritten as

```
\d++bar
```

Possessive quantifiers are always greedy; the setting of the `RX_UNGREEDY` option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning or processing of a possessive quantifier and the equivalent atomic group.

The possessive quantifier syntax is an extension to the Perl syntax. It originates in Sun's Java package.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

```
(\D+|<\d+>)*[!?]
```

matches an unlimited number of substrings that either consist of non-digits, or digits enclosed in `<>`, followed by either `!` or `?`. When it matches, it runs quickly. However, if it is applied to

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

it takes a long time before reporting failure. This is because the string can be divided between the two repeats in a large number of ways, and all have to be tried. (The example used `[!?]` rather than a single character at the end, because both PCRE and Perl have an optimization that allows for fast failure when a single character is used. They remember the last single character that is required for a match, and fail early if it is not present in the string.) If the pattern is changed to

```
((?>\D+)|<\d+>)*[!?]
```

sequences of non-digits cannot be broken, and failure happens quickly.

BACK REFERENCES

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. See the section entitled "Backslash" above for further details of the handling of digits following a backslash.

A back reference matches whatever actually matched the capturing subpattern in the current subject string, rather than anything matching the subpattern itself (see "[Subpatterns as subroutines](#)" below for a way of doing that). So the pattern

```
(sens|respons)e and \1libility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If careful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?i)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

Back references to named subpatterns use the Python syntax `(?P=name)`. We could rewrite the above example as follows:

```
(?<p1>(?!i)rah)\s+(?P=p1)
```

There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|(bc))\2
```

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the `RX_EXTENDED` option is set, this can be whitespace. Otherwise an empty comment can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, `(a\1)` never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern

```
(a|b\1)+
```

matches any number of "a"s and also "aba", "ababbaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

ASSERTIONS

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as `\b`, `\B`, `\A`, `\G`, `\Z`, `\z`, `^` and `$` are described above. More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the subject string, and those that look behind it.

An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed. Lookahead assertions start with `(?=` for positive assertions and `(?!` for negative assertions. For example,

```
\w+(?=;)
```

matches a word followed by a semicolon, but does not include the semicolon in the match, and

```
foo(?!bar)
```

matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

```
(?!foo)bar
```

does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion `(?!foo)` is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve this effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with `(?!)` because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

Lookbehind assertions start with `(?<=` for positive assertions and `(?<!` for negative assertions. For example,

```
(?<!foo)bar
```

does find an occurrence of "bar" that is not preceded by "foo". The contents of a lookbehind assertion are restricted such that all the strings it matches must have a fixed length. However, if there are several alternatives, they do not all have to have the same fixed length. Thus

```
(?<=bullock|donkey)
```

is permitted, but

```
(?<!dogs?|cats?)
```

causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. This is an extension compared with Perl (at least for 5.8), which requires all branches to match the same length of string. An assertion such as

```
(?<=ab(c|de))
```

is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

```
(?<=abc|abde)
```

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed width and then try to match. If there are insufficient characters before the current position, the match is deemed to fail.

PCRE does not allow the `\C` escape (which matches a single byte in UTF-8 mode) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind.

Atomic groups can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

```
abcd$
```

when applied to a long string that does not match. Because matching proceeds from left to right, PCRE will look for each "a" in the subject and then see if what follows matches the rest of the pattern. If the pattern is specified as

```
^.*abcd$
```

the initial `.*` matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

```
^(?>.*)(?<=abcd)
```

or, equivalently,

```
^.*+(?<=abcd)
```

there can be no backtracking for the `.*` item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Several assertions (of any sort) may occur in succession. For example,

```
(?<=\d{3})(?<!999)foo
```

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the subject string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does *not* match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn't match "123abcfoo". A pattern to do that is

```
(?<=\d{3}...)(?<!999)foo
```

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

```
(?<=(?<!foo)bar)baz
```

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

```
(?<=\d{3}(?!999)... )foo
```

is another pattern which matches "foo" preceded by three digits and any three characters that are not "999".

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

CONDITIONAL SUBPATTERNS

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two

possible forms of conditional subpattern are

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are three kinds of condition. If the text between the parentheses consists of a sequence of digits, the condition is satisfied if the capturing subpattern of that number has previously matched. The number must be greater than zero. Consider the following pattern, which contains non-significant white space to make it more readable (assume the `RX_EXTENDED` option) and to divide it into three parts for ease of discussion:

```
( \ ( ) ?      [ ^ ( ) ] +      ( ? ( 1 ) \ ) )
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if subject started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

If the condition is the string (R), it is satisfied if a recursive call to the pattern or subpattern has been made. At "top level", the condition is false. This is a PCRE extension. Recursive patterns are described in the next section.

If the condition is not a sequence of digits or (R), it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:

```
(? ( ? = [ ^ a - z ] * [ a - z ] )
 \ d { 2 } - [ a - z ] { 3 } - \ d { 2 }   |   \ d { 2 } - \ d { 2 } - \ d { 2 } )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the subject. If a letter is found, the subject is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

COMMENTS

The sequence `(?#` marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the `RX_EXTENDED` option is set, an unescaped `#` character outside a character class introduces a comment that continues up to the next newline character in the pattern.

RECURSIVE PATTERNS

Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth. Perl has provided an experimental facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern to solve the parentheses problem can be created like this:

```
$re = qr{\( (? : (?>[ ^ ( ) ] +) | (?p{$re}) ) * \) }x;
```

The `(?p{...})` item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears. Obviously, PCRE cannot support the interpolation of Perl code. Instead, it supports some special syntax for recursion of the entire pattern, and also for individual subpattern recursion.

The special item that consists of `(?` followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a "subroutine" call, which is described in the next section.) The special item `(?R)` is a recursive call of the entire regular expression.

For example, this PCRE pattern solves the nested parentheses problem (assume the `RX_EXTENDED` option is set so that white space is ignored):

```
\( ( (?>[^( )]+) | (?R) ) * \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (that is a correctly parenthesized substring). Finally there is a closing parenthesis.

If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```
( \ ( ( (?>[^( )]+) | (?1) ) * \ ) )
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern. In a larger pattern, keeping track of parenthesis numbers can be tricky. It may be more convenient to use named parentheses instead. For this, PCRE uses `(?P>name)`, which is an extension to the Python syntax that PCRE uses for named parentheses (Perl does not provide named parentheses). We could rewrite the above example as follows:

```
(?<pn> \ ( ( (?>[^( )]+) | (?P>pn) ) * \ ) )
```

This particular example pattern contains nested unlimited repeats, and so the use of atomic grouping for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa())
```

it yields "no match" quickly. However, if atomic grouping is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the subject, and all have to be tested before failure can be reported.

At the end of a match, the values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If you want to obtain intermediate values, a callout function can be used (see below and the [callout \(201\)](#) documentation). If the pattern above is matched against

```
(ab(cd)ef)
```

the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving

```
\( ( ( (?>[^( )]+) | (?R) ) * ) \)
  ^               ^
  ^               ^
```

the string they capture is "ab(cd)ef", the contents of the top level parentheses. If there are more than 15 capturing parentheses in a pattern, PCRE has to obtain extra memory to store data during a recursion. If no memory can be obtained, the match fails.

Do not confuse the `(?R)` item with the condition `(R)`, which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

```
< (?: (? (R) \d++ | [^<>]++) | (?R) ) * >
```

In this pattern, `(?R)` is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. The `(?R)` item is the actual recursive call.

SUBPATTERNS AS SUBROUTINES

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. An earlier example pointed out that the pattern

```
(sens|respons)e and \libility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

```
(sens|respons)e and (?1)ibility
```

is used, it does match "sense and responsibility" as well as the other two strings. Such references must, however, follow the subpattern to which they refer.

CALLOUTS

Perl has a feature whereby using the sequence `(?{...})` causes arbitrary Perl code to be obeyed in the middle of matching a regular expression. This makes it possible, amongst other things, to extract different substrings that match the same pair of parentheses when there is a repetition.

PCRE provides a similar feature, but of course it cannot obey arbitrary Perl code. The feature is called "callout". The caller of PCRE provides an external function by putting its address into `FINDRX` or `REPLACERX` variable.

Within a regular expression, `(?C)` indicates the points at which the external function is to be called. If you want to identify different callout points, you can put a number less than 256 after the letter C. The default value is zero. For example, this pattern has two callout points:

```
(?C1) dabc (?C2) def
```

During matching, when PCRE reaches a callout point (and callout function is set), the external function is called. It is provided with the number of the callout, and more data originally supplied by the caller.

The callout function may cause matching to backtrack, or to fail altogether. A complete description of the callout function is given [here \(201\)](#).

Last updated: 03 February 2003

Copyright © 1997-2003 University of Cambridge.

Regular expression options

The following [flags \(247\)](#) bits can be used with functions [findrx \(197\)](#) and [str.replacex \(230\)](#):

RX_CASELESS 0x100 (same as flag 1)

If this bit is set, letters in the pattern match both upper and lower case letters. It is equivalent to Perl's /i option, and it can be changed within a pattern by a (?i) option setting.

RX_MULTILINE 0x200 (same as flag 8)

By default, PCRE treats the subject string as consisting of a single "line" of characters (even if it actually contains several newlines). The "start of line" metacharacter (^) matches only at the start of the string, while the "end of line" metacharacter (\$) matches only at the end of the string, or before a terminating newline (unless RX_DOLLAR_ENDONLY is set). This is the same as Perl.

When RX_MULTILINE it is set, the "start of line" and "end of line" constructs match immediately following or immediately before any newline in the subject string, respectively, as well as at the very start and end. This is equivalent to Perl's /m option, and it can be changed within a pattern by a (?m) option setting. If there are no "\n" characters in a subject string, or no occurrences of ^ or \$ in a pattern, setting RX_MULTILINE has no effect.

RX_DOTALL 0x400

If this bit is set, a dot metacharacter in the pattern matches all characters, including newlines. Without it, newlines are excluded. This option is equivalent to Perl's /s option, and it can be changed within a pattern by a (?s) option setting. A negative class such as [^a] always matches a newline character, independent of the setting of this option.

RX_EXTENDED 0x800

If this bit is set, whitespace data characters in the pattern are totally ignored except when escaped or inside a character class. Whitespace does not include the VT character (code 11). In addition, characters between an unescaped # outside a character class and the next newline character, inclusive, are also ignored. This is equivalent to Perl's /x option, and it can be changed within a pattern by a (?x) option setting.

This option makes it possible to include comments inside complicated patterns. Note, however, that this applies only to data characters. Whitespace characters may never appear within special character sequences in a pattern, for example within the sequence (? (which introduces a conditional subpattern.

RX_ANCHORED 0x1000

If this bit is set, the pattern is forced to be "anchored", that is, it is constrained to match only at the first matching point in the string which is being searched (the "subject string"). This effect can also be achieved by appropriate constructs in the pattern itself, which is the only way to do it in Perl.

RX_DOLLAR_ENDONLY 0x2000

If this bit is set, a dollar metacharacter in the pattern matches only at the end of the subject string. Without this option, a dollar also matches immediately before the final character if it is a newline (but not before any other newlines). The RX_DOLLAR_ENDONLY option is ignored if RX_MULTILINE is set. There is no equivalent to this option in Perl, and no way to set it within a pattern.

RX_EXTRA 0x4000

This option was invented in order to turn on additional functionality of PCRE that is incompatible with Perl, but it is currently of very little use. When set, any backslash in a pattern that is followed by a letter that has no special meaning causes an error, thus reserving these combinations for future expansion. By default, as in Perl, a backslash followed by a letter with no special meaning is treated as a literal. There are at present no other features controlled by this option. It can also be set by a (?X) option setting within a pattern.

RX_NOTBOL 0x8000

The first character of the string is not the beginning of a line, so the circumflex metacharacter should not match before it. Setting this without RX_MULTILINE (at compile time) causes circumflex never to match.

RX_NOTEOL 0x10000

200. Regular expression options

The end of the string is not the end of a line, so the dollar metacharacter should not match it nor (except in multiline mode) a newline immediately before it. Setting this without `RX_MULTILINE` (at compile time) causes dollar never to match.

`RX_UNGREEDY 0x20000`

This option inverts the "greediness" of the quantifiers so that they are not greedy by default, but become greedy if followed by "?". It is not compatible with Perl. It can also be set by a (?U) option setting within the pattern.

`RX_NOTEMPTY 0x40000`

An empty string is not considered to be a valid match if this option is set. If there are alternatives in the pattern, they are tried. If all the alternatives match the empty string, the entire match fails.

`RX_UTF8 0x80000` (currently not supported in QM)

This option causes to regard both the pattern and the subject as strings of UTF-8 characters instead of single-byte character strings.

`RX_NO_AUTO_CAPTURE 0x100000`

If this option is set, it disables the use of numbered capturing parentheses in the pattern. Any opening parenthesis that is not followed by ? behaves as if it were followed by ?: but named parentheses can still be used for capturing (and they acquire numbers in the usual way). There is no equivalent of this option in Perl.

`RX_NO_UTF8_CHECK 0x200000` (currently not supported in QM)

Don't check if UTF-8 is valid. Faster but can be dangerous.

Callout callback function

[PCRE \(199\)](#) provides a feature called "callout", which is a means of temporarily passing control to the caller of PCRE in the middle of pattern matching. Within a regular expression, (?C) indicates the points at which the external function is to be called. Different callout points can be identified by putting a number less than 256 after the letter C. The default value is zero. For example, this pattern has two callout points:

```
(?C1) \dabc (?C2) def
```

Types FINDRX (used with [findrx \(197\)](#)) and REPLACERX (used with [str.replacerx \(230\)](#)) have member **fcallout**. If **fcallout** is address of user-defined function, that function is called each time a callout is encountered in pattern.

A template is available in menu -> File -> New -> Templates.

The function must begin with:

```
/
function[c] # CALLOUT&x
```

x is variable of type CALLOUT.

```
type CALLOUT version number CHARRANGE*vec $subject length start_match current_position
capture_top capture_last FINDRX*frx []REPLACERX*rrx
```

version - version number of the type.

number - number of the callout, as compiled into the pattern (that is, the number after ?C).

vec - array of offsets in subject string. The content can be inspected in order to extract substrings that have been matched so far.

subject - subject string (**s** in str.replacerx, **string** in [findrx](#)).

length - length of subject string.

start_match - offset within the subject at which the current match attempt started. If the pattern is not anchored, the callout function may be called several times for different starting points.

current_position - offset within the subject of the current match pointer.

capture_top - one more than the number of the highest numbered captured substring so far (that is, **vec** length). If no substrings have been captured, the value of **capture_top** is one.

capture_last - number of the most recently captured substring.

rf - address of variable that was passed to [findrx](#).

rr - address of variable that was passed to str.replacerx.

Return values

0	matching proceeds as normal.
>	matching fails at the current point, but backtracking to test other possibilities goes ahead, just as if a lookahead assertion had failed.
-1	matching fails (not found).
< -1	generate error with this error number. Should be < -100.

FINDRX type

A variable of FINDRX type can be used with [findrx \(197\)](#). Definition:

```
type FINDRX ifrom ito fcallout paramc
```

ifrom - start from here. 0-based character index in **string**.

ito - until here. If 0, until **string** length.

fcallout - [callout callback function \(201\)](#).

paramc - some value to pass to the callout callback function.

Before passing the variable to the function, you can set some its members, and leave unused members uninitialized, that is 0. For example, if you don't need a callout, you can set only **ifrom** and/or **ito**.

Example

```
str s="One two three"
str rx="[a-z]+(?C)o"
FINDRX r
r.fcallout=&sub.CalloutCallbackFunction
int i=findrx(s rx r)

#sub CalloutCallbackFunction
function[c]# CALLOUT*p
out p.start_match
```

REPLACERX; replacement callback function

REPLACERX type can be used with [str.replacex \(230\)](#) instead of replacement string. Definition:

```
type REPLACERX ifrom ito fcallout paramc frepl paramr $repl
```

ifrom - start from here. 0-based character index in **s** (subject string). Default: 0.

ito - do until here. Default: **s** length.

fcallout - [callout callback function \(201\)](#).

paramc - some value or pointer to be accessed by callout callback function.

frepl - replacement callback function.

paramr - some value or pointer to be accessed by replacement callback function.

repl - replacement string. Ignored if **frepl** is nonzero.

Before passing the variable to the function, you can set one or more members that you use, leaving other members uninitialized, that is 0.

If **frepl** and **fcallout** are 0, str.replacex behavior is same as usually (when using **replaceto** string), plus you can set **s** portion (**ifrom** to **ito**) where replacements must occur.

Replacement callback function

If **frepl** is address of a user-defined function, the function is called each time when a match is found during replacement process. It provides replacement string.

A template is available in menu -> File -> New -> Templates.

The function must begin with:

```
/
function# REPLACERXCB&x
```

x is variable of type REPLACERXCB. This variable is filled by str.replacex. Callback function can (and should) modify **match** and maybe some other members. In most cases you will only set **match** to be replacement string, and return 0.

```
type REPLACERXCB ~match ~strnew $subject CHARRANGE*vec lenv number REPLACERX*rrx
```

match - copy of matched substring. Function can modify it. It will become replacement string.

strnew - string that is being formatted from replacements and nonmatching portions of **s**. Finally it will replace **s**. Now it contains the portion of **s** from the beginning to the current match, with previous replacements.

subject - unmodified subject string (**s**).

vec - array of offsets in **s**. First element (vec[0]) contains offsets (beginning and end) of match. Subsequent elements contain offsets for submatches. If n-th subexpression not found then vec[n].cpMin and vec[n].cpMax are -1.

lenv - number of elements in **vec** array. It can be 1 (if no submatches) or more.

number - number of replacements (1 on first replacement, 2 on second and so on).

rr - address of variable that was passed to str.replacex.

Return values

0	indicates that match is replacement string. It will be appended to string being formatted. If match is not modified, matched substring will not be replaced.
> 0	indicates that nothing should be appended to string being formatted. This return value can be used either to remove matched substring, or when callback function itself appends replacement to strnew .
- 1	stop replacement process. str.replacex will return immediately. It returns number of replacements not including current, or -1 for single replacement mode. To stop replacement process and include current replacement, set x.rr.ito = 0.
< - 1	generate error with this error number. Should be < -100.

Example

```
multiply all decimal numbers in s by 5:
str s="A2 B100"
REPLACERX r
r.frepl=&sub.replacerx_callback
s.replacerx("\d+" &r)
out s ;;A10 B500

#sub replacerx_callback
function# REPLACERXCB&x
x.match = val(x.match) * 5
```

Append string as new line

Syntax

```
s.addline(ss [flags])
```

Parameters

s - str variable.

ss - string to append.

flags (247):

1	Don't add new line characters at the end.
---	---

Remarks

Added in QM 2.2.0.

Allocate or free string buffer

Syntax1 - allocate

```
s.all(length [flags] [fillchar])
```

Syntax2 - compact

```
s.all(-1)
```

Syntax3 - free

```
s.all
```

Parameters

s - str variable.

length - number of bytes to allocate, not including the terminating null character.

flags (247):

1	Preserve previous string (max length bytes).
2	Set string length (<i>len</i> property (234)) to length . If this flag not used, <i>len</i> will be 0 or length of preserved string.

fillchar - [character \(239\)](#) to fill allocated string (except preserved part). If omitted, allocated string content is undefined. In [Unicode \(267\)](#) mode **fillchar** must be an ASCII character.

Remarks

Syntax1

Allocates string buffer (memory).

Sets *nc* [property \(234\)](#) to the number of allocated bytes, not including the terminating null character.

For better performance, may allocate more than **length** bytes.

Syntax2

Compacts string buffer. It frees extra memory that may be added or not freed by other string functions. Extra memory is used to avoid frequent reallocations.

Syntax3

Frees string buffer. Sets *lpstr*, *len* and *length* [properties \(234\)](#) to 0.

See also: [str.fix \(212\)](#).

Tips

Function [all](#) can be used to allocate buffer before calling a dll function that requires pointer to buffer. When dll function returns, use function [fix](#) without **length** or with **length** = dll function's return value.

Example

```
int hwnd = win()
str s
s.all(256)
int i = GetWindowText(hwnd s 256)
s.fix(i)
or:
str s.fix(GetWindowText(win() s.all(256) 256))
```

Compare beginning, end or middle

Syntax

```
int s.beg(string [lens])
int s.begi(string [lens])
int s.end(string [lens])
int s.endi(string [lens])
int s.mid(string from [lens])
int s.midi(string from [lens])
```

Parameters

s - str variable.

string - string to compare with.

lens - number of characters to compare. Default: -1 (function itself finds **string** length).

from - 0-based character index in **s** from where to begin comparison.

Remarks

beg returns 1 if **s** begins with **string**, or 0 if not.

end returns 1 if **s** ends with **string**, or 0 if not.

mid returns 1 if **s** contains **string** at **from** position, or 0 if not.

begi, **endi** and **midi** perform case insensitive comparison.

Example

```
str s = "Edit\Copy"
if(s.beg("Edit")) out "True"; else out "False"
if(s.midi("cop" 5)) out "True"; else out "False"
```

Get dll error description

Syntax

```
s.dllerror([message] [dll] [errorcode])
```

Parameters

s - str variable.

message - string to prepend. Default: "".

dll - where to search for error description. Default: "". Possible values:

"" or omitted	search in Windows system dlls (mostly used).
a dll filename or full path	search in that dll.
"C"	get C run-time library (MSVCRT) error description.

errorcode - numeric error code. If omitted or 0, calls [GetLastError](#); it gets error code set by the last called dll function (or explicitly with [SetLastError](#)) in this thread.

Remarks

Many Windows dll functions and other functions set an error code (an integer number) when they fail. If you read function documentation, often it says "If the function fails ... call GetLastError.". Some other functions instead return a nonzero value when they fail, and it usually is an error code. Many error codes are known and have text descriptions. This function gets the description (calls [FormatMessage](#)). If description not found, gets "".

Example

```
if CopyFile("nosuchfile" "tonosuchfile" 1) = 0
    _str s.dllerror("Last dll error: ")
    _out s
```

Output:

Last dll error: The system cannot find the file specified.

Get short (DOS) path or long path

Syntax

```
s.dospath([file] [flags])
```

Parameters

s - str variable.

file - full path of file or folder. Default: **s**.

flags (247):

1	Get long path. If flag 1 is not used, gets short path.
2	QM 2.3.1. If fails to get short path, and the path contains spaces, enclose it in double quotes.

Remarks

Added in QM 2.2.0.

A file path can have two formats - long and short. Usually are used long paths, but also are supported short paths that were used in DOS. For example, short path version of "c:\program files\some folder\some file.txt" would be "c:\progra~1\somefo~1\somefi~1.txt" or similar. Some programs don't support long paths in command line, so you have to use short path. This function does conversion between long and short path versions.

If **file** begins with a \$spacial folder\$ or %environment variable%, this function expands it.

The file/folder must exist, or the function will be unable to get short or long path. If it does not exist, or the function fails for some other reason, **s** receives **file** (possibly expanded).

Note: On some computers short path generation is disabled.

Encrypt, decrypt

Syntax1 - BlowFish

```
s.encrypt(1 src key [flags])
s.decrypt(1 src key)
```

Syntax2 - MD5

```
s.encrypt(2 [src] [src2] [flags])
```

Syntax3 - Base64

```
s.encrypt(4 [src] ["" flags])
s.decrypt(4 [src])
```

Syntax4 - Hex

```
s.encrypt(8 [src] ["" flags])
s.decrypt(8 [src])
```

Syntax5 - decrypt password

```
s.encrypt(16 password function [flags])
s.decrypt(16 password key [flags])
```

Syntax6 - LZO compression

```
s.encrypt(32 [src])
s.decrypt(32 [src])
```

Parameters

s - str variable. Receives result.
1,2,4,8,16,32 - encryption algorithm. Also can be 1|4 (BlowFish+Base64), 1|8 (BlowFish+Hex), 2|4 (MD5+Base64) and 2|8 (MD5+Hex), 32|4 (LZO+Base64) and 32|8 (LZO+Hex).
src - string to encrypt or decrypt. If **src** is str variable, it can contain binary data. Default: **s** itself.
key - encryption key. String of 1 to 56 characters length.
src2 - second string to hash with **src**. Optional.
password - password to encrypt or decrypt.
function - name of the function to which the password will be passed. String.
[flags \(247\)](#) - read in remarks.

Remarks

Encrypts or decrypts string or binary data using one of standard algorithms. The algorithms are used for different purposes. You can find more info about them on the Internet.

Syntax1 - BlowFish (algorithms 1, 1|4, 1|8)

Encrypts/decrypts **src** using BlowFish algorithm.

BlowFish is used to encrypt data for security purposes.

Uses encryption key. To decrypt, use same key as when encrypting.

Error if **src** or **key** is invalid and cannot be encrypted or decrypted.

By default, the result is binary. If algorithm is 1|4 or 1|8, converts it to text in Base64 or Hex format.

Syntax2 - MD5 (algorithms 2, 2|4, 2|8)

encrypt generates hash ("digest", "checksum") of **src** using MD5 algorithm.

MD5 is used to check whether two strings (or files, passwords, other data) are identical without comparing the strings. Instead you compare their MD5 hash values. The hash is a 16-byte value. It is different for different data. It cannot be decrypted.

The purpose is similar as of CRC (see [Crc32 \(114\)](#)), which generates a 4-byte value. MD5 is more reliable. The speed is similar.

By default, the result is binary. If algorithm is 2|4 or 2|8, converts it to text in Base64 or Hex format.

flags:

0x100 (QM 2.3.2)	src is file. <ul style="list-style-type: none"> Error if fails to open. To get error info, call GetLastError or _s.dllerror.
------------------------	--

Syntax3 - Base64 (algorithm 4)

Encodes/decodes **src** using Base64 algorithm.

Base64 is used to convert binary data to text.

When encoding, is generated string of approximately 4/3 of **src** length and consisting of alphanumeric and several other characters.

flags - flags that can be used with [encrypt](#):

1	don't add padding (default: if encoded data is not 4-multiple, adds 1 to 3 characters =).
2	don't add line breaks (default: if encoded data is long enough, adds line breaks).

These flags also can be used with algorithms involving Base64: 1|4, 2|4 and 32|4.

Syntax4 - Hex (algorithm 8)

Encodes/decodes **src** using Hex algorithm.

Hex is used to convert binary data to text. Faster than Base64, but the result is bigger.

When encoding, generates string of **src** length * 2 and consisting of hexadecimal characters (0 - 9, A - F).

flags - flags that can be used with [encrypt](#):

1	add spaces to separate bytes, like "AA BB CC DD".
2, 4, 8, 16, 32, 64	add line breaks. The number is the number of bytes in a line.

Syntax5 - decrypt password (algorithm 16)

[decrypt](#) decrypts password.

An encrypted password has format "[*XXXXXXXXXXXXXXXXXXXX*]". It also can be string containing one or more encrypted passwords, e.g. "pwd=[*0123456789ABCDEF*];".

If **password** does not contain an encrypted password, simply assigns **password** to **s**. But if flag 1 is used, generates error. Does not generate other errors.

Passwords can be encrypted in [Options->Security \(14\)](#) or using [encrypt](#) with algorithm 16. [Read more \(150\)](#).

Syntax5 not available in exe. RT error if used.

Syntax6 (QM 2.3.2) - LZO compression (algorithms 32, 32|4, 32|8)

Compresses/decompresses using LZO (<http://www.oberhumer.com/opensource/lzo/>).

Fast compression and extremely fast decompression. However compression rate is not high. Good for non-compressed images (bmp, ico).

By default, the result is binary. If algorithm is 1|4 or 1|8, converts it to text in Base64 or Hex format.

See also: [str.escape \(210\)](#), [str.ansi \(238\)](#), [inpp \(61\)](#), [Crc32 \(114\)](#).

Examples

Set password. Save it in registry (MD5-encrypted).

```
str s
if(!inp(s "New password:" "" "*")) ret
s.encrypt(10)
rset s "Password11" "\\Test"
```

Ask for password that previously was saved in registry (MD5-encrypted).

```
str ss sss
if(rget(ss "Password11" "\\Test"))
  if(!inp(sss "Password:" "" "*")) ret
  sss.encrypt(10)
  if(sss!=ss) mes- "Password is incorrect" "" "!"
mes "OK"
```

Replace escape sequences with characters, or vice versa

Syntax

`s.escape([flags])`

Parameters

s - str variable.

flags - one of:

0	Replace escape sequences (137) (' ', [], [digits]) to characters (", new line, character).
1	Replace unsafe characters (", new line, tab, control characters, [, ' ') to escape sequences.
8	Urldecode.
9	Urlencode. Use for URL parameters containing spaces and other unsafe characters. It replaces all characters except 0-9, A-Z, a-z, _, - and . to escape sequences in form %xx, where xx is character code in hexadecimal format.
10	same as 8, but decode + to space.
11	same as 9, but encode space to +.
16	QM 2.3.2. Skip parts enclosed in { }, like with operator F (138) . This flag can be used with flags 0 and 1. With flag 1, in { } replaces " with `.

Remarks

Replaces escape sequences with characters, or vice versa.

QM 2.3.0. Removed flag 2 (don't escape characters above 127). Now does not escape these characters. It is because of possible problems with [UTF-8 \(267\)](#).

See also: [str.encrypt \(209\)](#), [str.ansi \(238\)](#)

Examples

```
str s=
  line 1
  "line 2"
s.escape(1) ;;escape
out s
s.escape(0) ;;unescape
out s

str s2="one, two"
s2.escape(9) ;;urlencode
out s2
s2.escape(8) ;;urldecode
out s2
```

Find and replace

Syntax

```
int s.findreplace(findwhat [replaceto] [flags] [delim] [from])
```

Parameters

s - str variable.

findwhat - string to find.

replaceto - replacement string. Default: "".

flags (247):

1	case insensitive.
2	whole word.
4	single replacement.
8	repeat replacement until s will not contain substrings that match findwhat . Without this flag, the result string can contain new such substrings. For example, if in string "abbcc" is replaced "bc" to "", the result is "abc". With this flag, the result is "a".
16	replace with [0] character. replaceto is not used and can be "".
32	replace [0] character (QM 2.2.0). findwhat is not used and can be "".
64	QM 2.3.3. Match always if findwhat begins/ends with a delimiter. Can be used with flag 2.
0x100	delim is table of delimiters.
0x200	QM 2.3.3. Add blanks to delim .

delim - [delimiters \(263\)](#).

from - start from here. 0-based character index in **s**. Default: 0.

Remarks

Searches in **s** for **findwhat**, and replaces with **replaceto**. If flag 2 is set, searches for substrings delimited by characters in **delim**, else **delim** is ignored.

If flag 4 is not set, replaces all found **findwhat** and returns the number of replacements. If flag 4 is set, replaces only the first found **findwhat**, and returns 0-based character index of **findwhat** in **s**.

Examples

```
str s="one two one twoo one.two"
s.findreplace("two" "THREE" 2 " ")
now s is "one THREE one twoo one.THREE"
```

```
s="one two one two one two"
s.findreplace("two" "THREE" 4 "" 5)
now s is "one two one THREE one two"
```

remove empty lines:

```
s="one[] [] [] [] []two"
s.findreplace("[ ][]" "[]" 8)
```

Set string length

Syntax

```
s.fix([length] [flags])
```

Parameters

s - str variable.

length - new length.

- Default: -1 - find string length by searching for null character.
- Must not exceed the number of bytes in the string buffer (*nc* [property \(234\)](#)).

flags (247):

1	Don't free extra memory. Without this flag, if length < <i>nc</i> , may free extra memory.
2	Limit s length. It changes s length only if length is less than current s length.
4	QM 2.3.3. Correct length so that string would not end with an invalid byte. If length is in middle of newline characters, remove whole newline. If in middle of a multibyte UTF-8 character, remove whole character.

Remarks

Sets **s** length.

This function can be used to:

- Set correct string length after allocating string buffer ([str.all \(205\)](#)). Usually used with dll functions that require a string buffer.
- To make string shorter. That is, you can use `s.fix(x)` instead of `s.left(s x)`. Or `s.fix(0)` instead of `s=""`.

length is number of bytes, not characters. In Unicode mode, non-ASCII characters have more than 1 byte. To limit length to a number of characters, use [LimitLen](#) instead.

Example

```
int hwnd = win()
str s
s.all(256)
int i = GetWindowText(hwnd s 256)
s.fix(i)
or:
str s.fix(GetWindowText(win() s.all(256) 256))
```

Format string

See also: [variables in strings \(138\)](#), [string constants and escape sequences \(137\)](#)

Syntax

```
s.format(formatstring ...)
s.formata(formatstring ...)
```

Parameters

s - str variable.

formatstring - format-control string.

... - arguments. Variables or other values that will replace format fields in **formatstring**.

- The order and type of arguments must match corresponding format fields in **formatstring**.
- Arguments must be of [intrinsic types \(140\)](#) (str, lpstr, int, double, byte, word, long). Don't use [OLE Automation types \(145\)](#).

Remarks

format creates string that can include values of variables. The **formatstring** consists of ordinary characters that are copied unchanged, and [format fields \(248\)](#) that are replaced with values of arguments.

formata differs from **format** only that it writes to the end of **s** (appends).

Tips

The mostly used format fields: %i for integer numbers (int, byte, word), %s for strings (str, lpstr), %G for double, %I64i for long, %c for a single character, %X for integers in hexadecimal.

To insert this function, can be used the Text dialog from the floating toolbar.

out and some other functions also support formatting. Example: `int i=5; out "i=%i" i`

Example

```
int i=50
str s="stringvar"
double d=3.1415926535897932384626433832795
str f
f.format("variables: i=%i, s='%s', d=%.10G" i s d)
out f ;;variables: i=50, s="stringvar", d=3.141592654
```

Create string from several parts

Syntax

```
s.from(s1 [s2 ...])
```

Parameters

s - str variable.

s1, s2, ... - string or numeric values.

Remarks

str does not support expression `s = s1 + s2` to join strings. For this purpose use function [from](#).

Tip: To append other string to the end or beginning of **s**, you can use operator `+` or `-`. See example.

See also: [str.format \(213\)](#), [str.fromn \(215\)](#), [str.addline \(204\)](#), [strings with variables \(138\)](#).

Examples

```
str s ("notepad") ss
ss.from(s ".exe") ;;now ss is "notepad.exe"
ss=s; ss+" .exe" ;;the same
```

Create string from several string or binary parts of specified length

Syntax

```
s.fromn(ptr len [ptr2 len2 ...])
```

Parameters

s - str variable.

ptr - string, or pointer to binary data.

len - number of bytes to copy from **ptr**. If -1, the function finds **ptr** length (**ptr** must be null-terminated string).

... - more **ptr/len** pairs.

Remarks

This function can be used to join several strings (full or of specified length) or/and binary data. Max number of **ptr/len** pairs is 16.

Although str variables can contain binary data, many str functions work only with null-terminated strings.

See also: [str.format \(213\)](#) with [%m \(248\)](#).

Example

```
str s1="123456789"
str s2="ABCDEFGHI"
str s.s.fromn(s1 4 " " 1 s2 -1) ;;4 bytes from s1, space and s2
```

the same with format. If s1 is not binary data, can use s instead of m.

```
s.format("%.4m %s" s1 s2)
```

or

```
s.format("%.*m %s" 4 s1 s2)
```

Clipboard, copy, paste

Syntax

```
s.getclip([format])
s.setclip([format])
s.getsel([cut] [format] [control])
s.setsel([format] [control])
```

Parameters

s - str variable.

format - clipboard format. Integer (for predefined formats) or string (for registered formats). Default or 0: text.

cut - if nonzero, use Ctrl+X (cut). Default or 0: use Ctrl+C (copy).

control - handle of control to work with. Default or 0: the focused control.

- If used, sends a message instead of keys. Read more in remarks.

Remarks

[getclip](#) copies clipboard data to **s**. It is text, unless other format specified.

[setclip](#) copies **s** to the clipboard.

[getsel](#) copies selected text to **s**.

[setsel](#) pastes **s**. If **format** and **control** not used, it is the same as [paste \(58\)](#) ([paste s](#)).

All these functions use the clipboard. After [getsel](#) and [setsel](#), QM restores previous clipboard content (text only), unless [runtime option \(97\)](#) `opt clip 1` is set.

For [setsel](#), the speed depends on [spe \(90\)](#).

To empty the clipboard, use [setclip](#) with empty string:

```
str s.setclip
```

or

```
s.all; s.setclip
```

In [Unicode \(267\)](#) mode these functions automatically support Unicode text. Don't use CF_UNICODETEXT format, unless **s** contains UTF-16 text. QM text format is UTF-8 (in Unicode mode) or ANSI.

str variables can store text or binary data. These functions recognize clipboard formats CF_TEXT, CF_OEMTEXT, CF_UNICODETEXT and CF_BITMAP. Data of other formats is transferred to/from the clipboard without processing. Also can be used [registered formats](#); then **format** must be format name (eg "HTML Format"), or numeric value returned by [RegisterClipboardFormat](#).

Copy something and run this macro. It shows clipboard formats currently in the clipboard. Those with strings are registered formats.

```
int f; str s
OpenClipboard 0
rep CountClipboardFormats
  f=EnumClipboardFormats(f)
  s.fix(GetClipboardFormatName(f s s.all(100)))
  out "%i %s" f s
CloseClipboard
```

If **format** is CF_BITMAP, **s** must be picture file path. Instead of using **s** to store clipboard data, is used that file:

[getclip](#) saves clipboard data to file **s**. File must be bmp.

[setclip](#) copies file **s** to the clipboard. File can be bmp, jpg or gif. QM 2.3.4: also can be png.

[getsel](#) saves selected picture to file **s**. File must be bmp.

[setsel](#) pastes file **s**. File can be bmp, jpg or gif. QM 2.3.4: also can be png.

Functions `getsel` and `setsel`, if **control** is omitted or 0, use keys (Ctrl+C, Ctrl+X, Ctrl+V) to copy/cut/paste. If **control** is a child window handle, these functions instead send messages (WM_COPY, etc) to it. These messages are supported by most edit and rich edit controls. Messages can be sent even to controls in inactive windows.

Examples

```
str s1 = "string1"
str s2
s1.setclip
s2.getclip
now s2 is "string1"

str s.getfile("unicodefile")
s.setsel(CF_UNICODETEXT)

str ss="$My Pictures$\test.bmp"
ss.setclip(CF_BITMAP)

str s.getclip("HTML Format")
```

Read or write file

Syntax

```
s.getfile(file [from] [nbytes])
s.setfile(file [from] [nbytes] [flags])
```

Parameters

s - str variable.

file - [file \(246\)](#).

from - offset in file. Default: 0.

nbytes - number of bytes to read or write. Default: -1.

flags (247) (used only with [setfile](#)):

1	Append new line characters.
2	When writing to existing file (OPEN_ALWAYS), set end of file where writing ends. Read more in remarks.
4	QM 2.3.2. Create new or open existing file. Read more in remarks.
0x100	QM 2.4.0. Safe/atomic saving. The file will never be corrupted on power failure etc. Writes to a temporary file, flushes its buffers, and renames the temporary file to file , replacing if exists.
0x200	QM 2.4.0. Safe saving and backup. Same as 0x100, but also creates a backup file, named file-backup .

Remarks

[getfile](#) loads file content to **s**.

Error if the file does not exist.

If **from** and **nbytes** are not used or are 0 and -1, loads whole file.

If **from** is >0 and **nbytes** not used or -1, loads whole file starting from **from**.

Not error if **from+nbytes** is more than file length. Then **s** will contain less than **nbytes** or will be empty.

QM 2.4.1. [getfile](#) supports [macro resources \(261\)](#). If **file** is resource name (eg "resource:data.bin"), gets resource data. When creating [exe \(51\)](#), QM adds the macro resource to exe resources, and in exe [getfile](#) gets data from the exe resource. Don't use **from** and **nbytes**.

QM 2.3.0. [getfile](#) supports [exe resources \(51\)](#). If **file** begins with : and an integer number 1 to 0xffff (eg ":5 c:\data.bin"), in exe [getfile](#) gets data from exe resource whose id is the number and type is RT_RCDATA (10). When the macro runs in QM, it gets data from the file. When creating exe, if "Auto add files ..." is checked, QM adds the file to exe resources. Don't use **from** and **nbytes**.

Don't use this function to load large files, because then may fail to allocate memory for **s**. Instead read parts of the file. It is more efficient with Windows API functions; you can use class [__HFile](#). Or use a database, for example [Sqlite](#) class.

[setfile](#) writes **s** to file.

If the file does not exist, creates. Uses Windows API function [CreateFile](#) with creation mode CREATE_ALWAYS or OPEN_ALWAYS.

- By default uses CREATE_ALWAYS. If the file exists, deletes it and creates new.
- Uses OPEN_ALWAYS if flag 4 or **from** is not 0. If the file exists, opens it for writing. It preserves old data that is not overwritten with new data. You can use flag 2 to truncate old data if need.

If **from** is -1, appends.

If **nbytes** is < 0 or > **s.len**, writes whole **s**.

Safe saving is slower, mostly because immediately writes to disk (else Windows would write later). Then just renames file, which is fast. However if OPEN_ALWAYS used, copies old file to the temporary file, which can be slow if the file is big.

Some possible [setfile](#) errors:

The filename contains invalid filename characters. To avoid it, you can use [str](#) function [ReplaceInvalidFilenameCharacters](#). The file is opened for exclusive access by another process or some QM function.

QM is running not as administrator and the file is in Program Files, Windows, or some other protected folder. The file has read-only attribute. When using safe saving, the function removes this attribute and does not fail.

QM 2.3.5. Creates parent folder if does not exist. In older QM would be error.

Note: In [Unicode \(267\)](#) mode, other string functions interpret text as UTF-8. If you use [getfile](#) to load an ANSI text file that contains non ASCII characters, and pass the text to other string functions, you may have to [convert \(238\)](#) ANSI to UTF-8. You also may have to do conversions in some other cases.

Tips

You can use the My QM folder (in My Documents) to save various files used by your macros. Use [special folder \(246\)](#) string \$my qm\$.

These functions open the file, write/read, and close. To perform multiple write/read operations without reopening, instead use Windows API functions; you can use class [__HFile](#).

If it is possible that several threads (running functions) access the same file simultaneously, use [lock \(112\)](#) to prevent it. If several processes or computers write to the file simultaneously, use [CFileInterlocked](#) class, it is in the forum.

You can use functions [rget \(106\)](#) and [rset](#) to write to or read from ini files.

See also: [CSV \(116\)](#), [XML \(117\)](#), [Sqlite](#) class, other file category functions and classes.

Example

```
str s1 s2 f
f="$my qm$\test.txt"
s1="test data"
s1.setfile(f)
s2.getfile(f)
out s2
```

Find and get n-th line of other string

Syntax

```
int s.getl(ss [n] [flags])
```

Parameters

s - str variable.

ss - source string.

n - 0-based index of line. Default: -1.

flags (247):

2	get all right part.
---	---------------------

Remarks

Gets **n**-th line from **ss**. Returns 0-based index of first character of the line in **ss**. If there are less than **n+1** lines in **ss**, returns -1.

If **n** is omitted or negative, gets next line. It works in same function only.

See also: [findl \(191\)](#), [numlines \(187\)](#), [foreach \(128\)](#), [line break characters \(253\)](#)

Example

```
str s ss="one[]two[] []four[]"
int i
for i 0 2000000000
  if(s.getl(ss -i)<0) break ;;no more
  if(s.len=0) continue ;;skip empty
out s
```

Get macro text or other property

Syntax

```
s.getmacro([macro] [what])
```

Parameters

s - str variable.

macro - [QM item \(19\)](#) name (full, case insensitive) or [id \(107\)](#) (integer).

- If omitted or "" - QM item that is currently open in the code editor.
- Can be QM item path. For example, "\\Mouse\\Next" is macro "Next" in folder "Mouse".
- If [sub-function \(182\)](#), QM uses its parent function. To get name of current sub-function, use [getopt \(98\)](#).

what - what to get:

0	text.												
1	name. See also getopt itemname (98) .												
2	trigger. It is encoded (264) trigger string. It consists of trigger (possibly with escape sequences), programs and trigger filter function. To get parsed components or description, use qmitem (107) .												
3-8	obsolete , use qmitem (107) instead. <table border="1"> <tr> <td>3</td><td>type. s will be "0" if the item is macro, "1" if function, "2" if menu, "3" if toolbar, "4" if T. S. menu, "5" if folder, "6" if member function, "7" if file link.</td></tr> <tr> <td>4</td><td>disabled state. s will be "1" if the item is disabled, or "0" if not.</td></tr> <tr> <td>5</td><td>encrypted state. s will be "1" if the item is encrypted, or "0" if not.</td></tr> <tr> <td>6</td><td>shared state. s will be "1" if the item is in a shared folder, or "0" if not.</td></tr> <tr> <td>7</td><td>id.</td></tr> <tr> <td>8</td><td>read-only state. s will be "0" if the item is not read-only, "2" if the item is encrypted, "1" if the item is in read-only folder and not encrypted.</td></tr> </table>	3	type. s will be "0" if the item is macro, "1" if function, "2" if menu, "3" if toolbar, "4" if T. S. menu, "5" if folder, "6" if member function, "7" if file link.	4	disabled state. s will be "1" if the item is disabled, or "0" if not.	5	encrypted state. s will be "1" if the item is encrypted, or "0" if not.	6	shared state. s will be "1" if the item is in a shared folder, or "0" if not.	7	id.	8	read-only state. s will be "0" if the item is not read-only, "2" if the item is encrypted, "1" if the item is in read-only folder and not encrypted.
3	type. s will be "0" if the item is macro, "1" if function, "2" if menu, "3" if toolbar, "4" if T. S. menu, "5" if folder, "6" if member function, "7" if file link.												
4	disabled state. s will be "1" if the item is disabled, or "0" if not.												
5	encrypted state. s will be "1" if the item is encrypted, or "0" if not.												
6	shared state. s will be "1" if the item is in a shared folder, or "0" if not.												
7	id.												
8	read-only state. s will be "0" if the item is not read-only, "2" if the item is encrypted, "1" if the item is in read-only folder and not encrypted.												

Remarks

Gets QM item text or some other property.

Error if **macro** is invalid at run time.

Some features not available in [exe \(51\)](#).

Examples

```
str s ss
s.getmacro("Untitled")
ss.getmacro("\\Folder\\Macro")

int i = qmitem("Untitled")
_s.getmacro(i)
```

Replace macro text

Not available in [exe \(51\)](#).

Syntax

```
s.setmacro ([macro])
```

Parameters

s - str variable.

macro - [QM item \(19\)](#) name (full, case insensitive) or [id \(107\)](#) (integer).

- If omitted or "" - QM item that is currently open in the code editor.
- Can be QM item path. For example, "\\Mouse\\Next" is macro "Next" in folder "Mouse".
- If [sub-function \(182\)](#), QM uses its parent function.

Remarks

Replaces QM item text.

The item must not be read-only.

The user can Undo if the item is currently open (is in the "Open items" list).

See also: [getmacro \(219\)](#), [newitem \(108\)](#)

Get path or filename from full path

Syntax

```
s.getpath([file] [append])  
s.getfilename([file] [ext])
```

Parameters

s - str variable.

file - full path of a file or folder. Default: **s**.

append - some string to append. Use "" to remove "\".

ext - if nonzero, get filename with extension.

Remarks

[getpath](#) extracts path part from **file**.

[getfilename](#) extracts filename part from **file**.

QM 2.3.4. Supports file streams, like "c:\a\b.txt:stream1:\$DATA".

Tips

There are many Windows API functions that can be used to work with paths and URLs. For example, [PathFindExtension](#). Documented in [MSDN Library](#).

Example

```
str s1.getpath("c:\folder\test.txt")  
str s2.getfilename("c:\folder\test.txt")  
str s3.getfilename("c:\folder\test.txt" 1)  
now s1 is "c:\folder\", s2 is "test", s3 is "test.txt"
```

Convert variable of a user-defined type to/from string

Syntax

```
s.getstruct(var [flags])
s.setstruct(var [flags2])
```

Parameters

s - str variable.

var - variable of a [user-defined type \(154\)](#).

flags (247):

1	Add member names. If this flag not used, the string will contain only values.
---	---

flags2 (247):

1	Use this flag if s includes member names. When using this flag, the order of members in s is not important. Some members in s may be missing. Some lines in s may be empty or contain any data. This is useful when the string may be edited by the user. If this flag not used, the string must contain only values, exactly as generated by getstruct without flag 1.
2	Member names in the string don't have to match case.
4	Don't clear var members that are missing in s .
8	Error if some members are missing in s .

Remarks

[getstruct](#) converts and stores values of all members of a variable of a user-defined type (**var**) into a single string (**s**).

[setstruct](#) extracts values from **s** (which is previously generated by [getstruct](#) or other code) and populates **var**.

For example, if **var** is of a type that has two int members x and y, an example of **s** may be:

```
10
20
```

If using flag 1, it will be:

```
x 10
y 20
```

Member name and value are separated by one or more spaces.

The variable also can contain strings. For example, if **var** is of a type that has four str members s1, s2, s3 and s4, an example of **s** may be:

```
s1 string1
s2 "string that contains spaces[]or ''escape sequences''"
s3
s4 ""
```

If a string contains spaces or characters that must be [escaped \(210\)](#), in **s** it is or must be escaped and enclosed in double quotes, like s2 in the example. Otherwise enclosing is not necessary. [setstruct](#) does not unescape strings that are not enclosed. Empty strings are like s3 in the example. Zero length strings are like s4 in the example.

Embedded arrays look like:

```
array_of_int_word_byte_long_double 2 -5 184 0
array_of_byte_also_can_be "string"
array_of_str[0] string1
array_of_str[1] string2
array_of_POINT[0].x 10
array_of_POINT[0].y 20
```

```
array_of_POINT[1].x 30
array_of_POINT[1].y 40
```

If an int/word/byte member name contains "flags", [getstruct](#) formats its value in hexadecimal format.

The functions are slightly slower when using flag 1.

The string generated by [getstruct](#) with flag 1 can be passed to AddList function of [IStrinMap \(115\)](#). Don't forget to remove quotes and unescape enclosed values.

Not all types are supported. Restrictions:

1. **var** cannot be of an intrinsic type (int, byte, word, str, lpstr, long, double), pointer, interface pointer
2. **var** cannot contain lpstr. Cannot be or contain ARRAY, BSTR, VARIANT, pointer, interface pointer.

QM 2.3.2. The restriction 2 is not applied to [getstruct](#). It allows you to convert these types for debugging purposes.

Added in QM 2.2.1.

Example

```
out
type T1 byte'x word'y str's
type T2 int'i str's T1't double'd long'k byte'b[2]

T2 t tt
t.i=-5
t.s=""'string'[]line2"
t.t.x=200
t.t.y=50000
t.t.s="string2"
t.d=1.55
t.k=0x100000000
t.b[0]=3
t.b[1]=4

str s
s.getstruct(t 1)
out s
out "---"
s.setstruct(tt 1)

out tt.i
out tt.s
out tt.t.x
out tt.t.y
out tt.t.s
out tt.d
out tt.k
out tt.b[0]
out tt.b[1]
```

Output:

```
i -5
s ""'string'[]line2"
t.x 200
t.y 50000
t.s string2
d 1.55
k 4294967296
b[0] 3
b[1] 4
```

```
-5
"string"
```

```
line2  
200  
50000  
string2  
1.55  
4294967296  
3  
4
```

Find and get n-th token (part) of other string

Syntax

```
int s.gett(ss [n] [delim] [flags])
```

Parameters

s - str variable.

ss - source string.

n - 0-based index of token. Default: -1.

delim - [delimiters \(263\)](#).

flags (247):

2	Get all right part.
0x100	delim is table of delimiters.
0x200	QM 2.3.3. Add blanks to delim .

Remarks

Gets **n**-th token (substring delimited by characters in **delim**) from **ss**. Returns 0-based index of first character of the token. If there are less than **n+1** tokens in **ss**, returns -1.

If **n** is omitted or negative, finds next token. It works in same function only.

Tip: to get lines, use [getl \(218\)](#).

See also: [findt \(190\)](#) [findrx \(197\)](#)

Examples

```
get the second token
str s = "my.exe /cl"
str st.gett(s 1 " .:\/'"[])
out st

for each word
int t
for t 0 1000000000
if(st.gett(s -t)<0) break
out st
```

Get window text, class name, program, set window text

Syntax

```
s.getwintext(window)
s.setwintext(window)
s.getwinclass(window)
s.getwinexe(window [full])
```

Parameters

s - str variable.

(274)window - top-level or child window. Usually it is window handle.

full - if 0, get filename (e.g., "notepad"), else get full path (e.g., "C:\Windows\notepad.exe"). Default: 0.

Remarks

getwintext populates **s** with **window** text. It can be window title, button label, edit control text, static control text, etc.

setwintext sets **window** text.

getwinclass populates **s** with **window** class name.

getwinexe populates **s** with **window** executable file name or full path. Cannot get full path of a process running as other user.

See also: [GetProcessExename \(114\)](#)

Example

```
int h
str s
h=win("Notepad")
s.getwintext(h)
h=id(15 "Notepad")
s.setwintext(h)
```

Insert other string

Syntax

```
s.insert(ss [from] [nc])
```

Parameters

s - str variable.

ss - insert this string. Can be string or numeric value.

from - where to insert. 0-based character index in **s**. Default: 0.

nc - number of characters to copy. Default: -1 (whole **ss**).

Remarks

Inserts string **ss** into **s**.

Example

```
str s = "My dog cat"  
s.insert("and " 7)  
now s is "My dog and cat"
```

Convert to lowercase or uppercase

Syntax1

```
s.lcase (/ss/)  
s.ucase (/ss/)
```

Syntax2

```
s.lcase(from [length])  
s.ucase(from [length])
```

Parameters

s - str variable.

ss - a string to assign to **s** before converting.

from, len (QM 2.3.5) - range of characters (bytes).

- If **len** omitted or -1, converts all starting from **from**.
- If the range ends in a middle of a multibyte UTF-8 character, includes whole character.

Remarks

lcase converts to lowercase.

ucase converts to uppercase.

Examples

```
str s1 = "String"  
s1.ucase  
out s1 ;;STRING  
  
str s2 = "sTrInG"  
s2.ucase(0 1)  
s2.lcase(1)  
out s2 ;;String
```

Get part of other string

Syntax

```
s.left(ss nc)
s.right(ss nc)
s.get(ss from [nc])
s.geta(ss from [nc])
```

Parameters

s - str variable.
ss - source string.
nc - number of characters to copy.
from - 0-based index of first character to copy.

Remarks

Get part of **ss**.

left gets **nc** characters from the beginning of **ss**.

right gets **nc** characters from the end of **ss**.

get gets **nc** characters from middle of **ss**, starting from **from**. If **nc** is omitted or negative, gets all right part of **ss**.

geta - gets and appends.

Example

```
lpstr lp = "notepad.exe"
str s
s.left(lp 2)
now s is "no"
s.right(lp 3)
now s is "exe"
s.get(lp 7)
now s is ".exe"
s.get(lp 1 2)
now s is "ot"
s.geta(lp 7 1)
now s is "ot."
```

Remove part of string

Syntax

```
s.remove(from [nc])
```

Parameters

s - str variable.

from - 0-based index of first character to remove.

nc - number of characters to remove. Default: -1 (remove all characters at right (like [str.fix\(212\)](#))).

Remarks

Removes **nc** characters from middle of **s**.

Example

```
str s = "My dog and cat"  
s.remove(2 8)  
now s is "My cat"
```

Replace part of string

Syntax

```
s.replace(ss [from] [nc])
```

Parameters

s - str variable.

ss - replacement string. Can be string or numeric expression.

from - where to do the replacement. 0-based character index in **s**. Default: 0.

nc - number of characters to replace. Default: -1 (all characters at right from **from**).

Remarks

Replaces range of characters in **s**.

Example

```
str s = "one two three"  
s.replace("four" 4 3)  
now s is "one four three"
```

Find and replace using regular expression

[About regular expressions \(198\)](#)

[Regular expression syntax \(199\)](#)

[findrx \(197\)](#)

Syntax

```
int s.replacex(pattern [replaceto|rr] [flags])
```

Parameters

s - str variable.

pattern - regular expression that matches the substring to find. String.

replaceto - replacement string. Default: "". Some sequences of characters in **replaceto** have special meaning:

\$n, \$nn, \${n}	n-th or nn-th captured submatch (197) (1 to 99).
\$0, \${0}	entire match.
\$\$	\$
\$^	part of s that precedes match
\$'	part of s that follows match

rr - address of variable of type [REPLACERX \(203\)](#).

flags (247):

1	Case insensitive.
2	Whole word. This adds \b to the beginning and end of pattern .
4	Single replacement.
8	Multiline. If this flag is set (or (?m) is used in pattern), ^ and \$ match beginning and end of line. Default: ^ and \$ match beginning and end of whole string.
32	QM 2.3.0. Convert pattern from UTF-8 to ANSI. Used when QM is running in Unicode mode (ignored otherwise). Set this flag if pattern contains non ASCII (239) characters, but s is ANSI (not UTF-8). It is needed because these characters in pattern normally consist of 2 or 3 bytes, whereas characters in s consist of 1 byte.
pcre flags (200)	

Remarks

Finds parts of **s** that match **pattern** and replaces them with **replaceto**.

If flag 4 is not set, replaces all found matches, and returns the number of replacements. If flag 4 is set, replaces only the first found match, and returns 0-based character index of it in **s**, or -1 if not found.

Example

Replace two consecutive identical words with single

```
str s="Is is the cost of of gasoline going up up?"
```

```
out s.replacex("([a-z]+) \1" "$1" 1|2)
```

```
out s
```

Output:

```
3
```

```
Is the cost of gasoline going up?
```

Find file and get full path; expand special folder name

Syntax

```
s.searchpath([file] [path])
s.expandpath([file] [flags])
```

Parameters

s - str variable.

file - full path or filename (with extension) of file or folder. Also can be drive (should end with \). Default: **s**.

- Full path can begin with [special folder name \(246\)](#) enclosed in \$, or [environment variable \(143\)](#) enclosed in %.

path - folder where to search. If **file** contains full path, this part is ignored. Default: "".

flags (247):

1	Expand all (and only) environment variables. Without this flag, environment variables are expanded only if file begins with %.
2	QM 2.2.0. Unexpand path. For example, if file is "c:\windows\system32\file.exe", s will be "\$system\$\file.exe".

Remarks

searchpath searches for **file** in **path**. If **path** is omitted and **file** is not full path, searches in [these places \(246\)](#). If **file** is full path, only tests file existence. If **file** exists, **s** will receive full path, else **s** will be empty.

expandpath only expands special folder name or environment variable, but does not search. All QM file functions support special folders, so you don't have to use this function before calling them. You can use it before calling dll and COM functions.

See also: [FileExists \(114\)](#)

Examples

If file exists, get full path:

```
str s.searchpath("c:\folder\file.txt")
if(s.len) out s ;; "c:\folder\file.txt"
else out "not found"
```

Search for "notepad.exe" and get full path:

```
str s.searchpath("notepad.exe")
if(s.len) out s ;;e.g. "c:\windows\system32\notepad.exe"
else out "not found"
```

Get full path when the given path begins with special folder name:

```
str s.expandpath("$Desktop$\new file.txt")
out s ;;e.g. "c:\windows\desktop\new file.txt"
```

Set part of string

Syntax1

```
s.set(ss [from] [nc])
```

Syntax2

```
s.set(char [from] [nc])
```

Parameters

s - str variable.
ss - source string.
char - source character. Integer [character code \(239\)](#).
from - where to copy (0-based character index in **s**). Default: 0.
nc - number of characters to copy.

Remarks

Syntax1: Copies **nc** characters from **ss** (or whole **ss**, if **nc** omitted) to **s**, starting from **from**.

Syntax2: If **from** and **nc** are omitted, fills **s** with **char**. Otherwise, copies **nc** (default is 1) **char** characters to **s**, starting from **from**. In [Unicode \(267\)](#) mode, don't use non ASCII characters.

Common: If **from+nc>s.len**, **s** is automatically expanded. If **from>s.len**, area from **s.len** to **from** is filled with spaces.

Examples

```
str s = "My dog"
s.set("cat" 3)
now s is "My cat"

str s.all(6 2)
now s has allocated 6 bytes. String content is undefined.
s.set(' ' 7)
now s is " " (6 spaces)

str s = "Number"
s.set('1' 7)
now s is "Number 1"
```

Exchange string with another variable

Syntax

```
s.swap(s2)
```

Parameters

s - str variable.

s2 - another str variable.

Remarks

Exchanges strings of two str variables **s** and **s2** without copying.

Exchanges [str member variables \(234\)](#) **lpstr**, **len**, **nc**, but not **flags**.

Added in: QM 2.4.1.

str properties (lpstr, len, nc, flags)

str member variables:

lpstr - pointer to string.

- The string is in a string buffer (a block of memory) that is automatically allocated/freed.
- **lpstr** is address of first character of the string.
- **lpstr** is 0 if there is no buffer allocated.

len - length of string (number of bytes), not including [terminating null character \(183\)](#).

nc - size of string buffer.

- It is number of allocated bytes, not including terminating null character.
- To avoid frequent reallocations (it slows down), str variables may allocate more memory than need for string. Then **nc** is > **len**. The difference is called *extra bytes*.

flags (247):

1 (optimization)	str functions should never free extra bytes. It means that allocated string memory can grow but not shrink. Use this flag to avoid frequent reallocations. However some functions may ignore it.
2 (optimization)	str functions should allocate more extra bytes to avoid frequent reallocations.
4 (security)	QM 2.3.3. Erase string memory before freeing it. Use this flag for passwords and other confidential data. It makes much harder to retrieve the string from memory. To set this flag you can use this: <code>str s; s.set_secure</code> . In older QM versions you would have to erase explicitly: <code>str s="password"; ... s.set(0)</code> .

Remarks

When a str variable is just declared without initializing, all data members are 0.

To get string length of a str variable, use its **len** property.

When various operations are performed with a str variable, it's data members are managed automatically. You should not modify them. There are two cases when you can modify them:

1. You can modify **flags** to improve performance when multiple operations are performed with a str variable.
2. If you want to steal string buffer from a str variable, set **lpstr** to 0. To free the stealed memory, use [q_free \(114\)](#).

In C++, str definition would look like:

```
class str
{
public:
    LPSTR lpstr;
    int len, nc;
    BYTE flags;
//member functions
...
};
```

In QM, str definition would look like:

```
class str lpstr'lpstr len nc byte'flags
```

Example

```
str s = "string"
int i = s.len
now i is 6
s.flags = 1
now s buffer can grow, but not shrink
```

Format date/time string

This function is obsolete. The C style formatting is slow and has other problems. Use [str.timeformat \(236\)](#).

Syntax1 - standard date/time string

```
s.time([date])
```

Syntax2 - C style date/time string

```
s.time([date] cfrm [locale] [plusnseconds])
```

Syntax3 - Windows style date/time string

```
s.time([date] winfrmdate [winfrmtime])
```

Parameters

s - str variable.

date - variable of type [DATE \(93\)](#). Default: use current date.

cfrm - format-control string in C style.

locale - desired language. String. Default: "" (current locale).

plusnseconds - number of seconds to add. Can be negative. Default: 0.

winfrmdate - date part of format-control string in Windows style. If "", uses default format. If "-", uses only time part.

winfrmtime - time part of format-control string in Windows style. If "", uses default format. If "-" or omitted, uses only date part.

Remarks

To insert this function, can be used the Text dialog from the floating toolbar.

Syntax1

Formats date/time string for current locale.

Syntax2

Formats date/time string in C style. This style is selected at run time if format-control string contains % characters. The **cfrm** consists of one or more codes, preceded by a percent sign (%). Characters that do not begin with % are copied unchanged. The formatting codes are listed below:

%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for locale
%d	Day of month as decimal number (01 - 31)
%H	Hour in 24-hour format (00 - 23)
%I	Hour in 12-hour format (01 - 12)
%j	Day of year as decimal number (001 - 366)
%m	Month as decimal number (01 - 12)
%M	Minute as decimal number (00 - 59)
%p	Current locale's A.M./P.M. indicator for 12-hour clock
%S	Second as decimal number (00 - 59)
%U	Week of year as decimal number, with Sunday as first day of week (00 - 51)
%w	Weekday as decimal number (0 - 6; Sunday is 0)
%W	Week of year as decimal number, with Monday as first day of week (00 - 51)
%x	Date representation for current locale
%X	Time representation for current locale
%y	Year without century, as decimal number (00 - 99)
%Y	Year with century, as decimal number
%Z, %Z	Time-zone name or abbreviation; no characters if time zone is unknown

%%	Percent sign
----	--------------

The # flag may prefix any formatting code. In that case, the meaning of the format code is changed as follows.

%#a, %#A, %#b, %#B, %#p, %#X, %#z, %#Z, %#%	# flag is ignored.
%#c	Long date and time representation, appropriate for current locale. For example: "Tuesday, March 14, 1995, 12:41:29".
%#x	Long date representation, appropriate to current locale. For example: "Tuesday, March 14, 1995".
%#d, %#H, %#l, %#j, %#m, %#M, %#S, %#U, %#w, %#W, %#y, %#Y	Remove leading zeros (if any).

locale (language) affects, how string is formatted. Locale string can be found in Control Panel Regional Settings.

Syntax3

Formats date/time string in Windows style. This style is selected at run time if format-control string does not contain % characters. Below listed character sequences in format-control strings are replaced with year, month, etc values. Other characters are copied unchanged. To prevent character replacement, enclose it in single quotation marks.

winfrmdate:

d	Day of month as digits.
dd	Day of month as digits with leading zero for single-digit days.
ddd	Day of week as a three-letter abbreviation.
dddd	Day of week as its full name.
M	Month as digits.
MM	Month as digits with leading zero for single-digit months.
MMM	Month as a three-letter abbreviation.
MMMM	Month as its full name.
y	Year as last two digits, but with no leading zero for years less than 10.
yy	Year as last two digits, but with leading zero for years less than 10.
yyyy	Year represented by full four digits.
gg	Period/era string.

winfrmtime:

h	Hours with no leading zero for single-digit hours; 12-hour clock.
hh	Hours with leading zero for single-digit hours; 12-hour clock.
H	Hours with no leading zero for single-digit hours; 24-hour clock.
HH	Hours with leading zero for single-digit hours; 24-hour clock.
m	Minutes with no leading zero for single-digit minutes.
mm	Minutes with leading zero for single-digit minutes.
s	Seconds with no leading zero for single-digit seconds.
ss	Seconds with leading zero for single-digit seconds.
t	One character time-marker string, such as A or P.
tt	Multicharacter time-marker string, such as AM or PM.

The function cannot format milliseconds.

Format date/time string

Syntax

```
s.timeformat([frm] [time] [locale] [dateFlags] [timeFlags])
```

Parameters

s - str variable.

frm - format-control string.

- If omitted or "", uses "{D} {T}" (short date/time, like "08/09/2009 2:25 PM").

time - variable of type [DATE](#), [SYSTEMTIME](#) or [FILETIME \(93\)](#), containing date and/or time.

- If omitted or 0, uses current local time.
- QM 2.3.4. Also can be variable of type [DateTime](#), or [long](#) in [FILETIME](#) format.

locale - locale identifier. See [table](#). If omitted, or 0, uses LOCALE_USER_DEFAULT. Read more later in this topic.

dateFlags - [GetDateFormat](#) flags.

timeFlags - [GetTimeFormat](#) flags.

Remarks

Added in QM 2.3.1. In older QM versions can be used [str.time \(235\)](#).

The function copies **frm** to **s**, replacing parts enclosed in {} to date or time. Certain characters in enclosed parts control the date/time format. The characters must match case.

In **frm** can be used parts of several types:

{D}	short date, like "08/09/2009".
{DD}	long date, like "August 9, 2009".
{T}	time without seconds, like "15:59" or "3:59 PM".
{TT}	time with seconds, like "15:59:30" or "3:59:30 PM".

For custom date format can be used the following character sequences. They must be in **frm** parts enclosed in {}. They are the same as with [GetDateFormat](#).

	Inserts	Example
d	Day of month as digits.	5
dd	Day of month as digits with leading zero for single-digit days.	05
ddd	Day of week as its abbreviated name.	Sun
dddd	Day of week as its full name.	Sunday
M	Month as digits.	4
MM	Month as digits with leading zero for single-digit months.	04
MMM	Month as its abbreviated name.	Apr
MMMM	Month as its full name.	April
y	Year as last two digits, but with no leading zero for years less than 10.	9
yy	Year as last two digits, but with leading zero for years less than 10.	09
yyyy	Full year.	2009
gg	Period/era string.	A.D.

For custom time format can be used the following character sequences. They must be in **frm** parts enclosed in {}. They are the same as with [GetTimeFormat](#).

	Inserts
h	Hours with no leading zero for single-digit hours; 12-hour clock.
hh	Hours with leading zero for single-digit hours; 12-hour clock.
H	Hours with no leading zero for single-digit hours; 24-hour clock.
HH	Hours with leading zero for single-digit hours; 24-hour clock.
m	Minutes with no leading zero for single-digit minutes.

mm	Minutes with leading zero for single-digit minutes.
s	Seconds with no leading zero for single-digit seconds.
ss	Seconds with leading zero for single-digit seconds.
t	One character time-marker string, such as A or P.
tt	Multicharacter time-marker string, such as AM or PM.

This function cannot format milliseconds. For this you can use class [DateTime](#) or Windows API functions, such as [GetTickCount](#), [timeGetTime](#), [GetLocalTime](#). See example.

If you have a [DateTime](#) variable, you can instead call its function [ToStr](#) or [ToStrFormat](#) (supports milliseconds etc).

By default, this function uses the date/time format and language from Control Panel -> Regional. Use **locale** only if you need some other language/format. A [locale identifier](#) is a number that includes primary language and sublanguage constants, like in this example:

```
out _s.timeformat("{DD} {TT}" 0 WINAPI.LANG_FRENCH|(WINAPI.SUBLANG_FRENCH_CANADIAN<<10))
```

This function does not generate errors. If some part or sequence in **frm** is incorrect, tries to find the nearest match (eg yyy - > yyyy), or leaves the part unchanged. If **locale** is unsupported, uses current locale (LOCALE_USER_DEFAULT). If replacing some enclosed part fails for other reasons, for example an incorrect argument value, the enclosed part will be empty, and the function sets [_hresult\(144\)](#) to 1. Sets it to 0 if all parts were successfully replaced.

To create code for this function, you can use the Text dialog from the floating toolbar.

With [F string \(138\)](#), use {{ }} for date/time parts, because { } are used for variables. See example.

Examples

```
str s
s.timeformat ;;same as s.timeformat("{D} {T}")
s.timeformat("{D} {TT}")
s.timeformat("Current date is {DD}")
s.timeformat("Current date is {MMM dd yyyy}, {HH 'hours and' mm} minutes")
s.timeformat("" 0 WINAPI.LANG_ENGLISH|WINAPI.SUBLANG_ENGLISH_US)

DATE d.getclock ;;get current time
d=d+1 ;;add 1 day
s.timeformat("{DD} {TT}" d)

SYSTEMTIME st
GetLocalTime &st
s.format("%02i:%02i:%02i.%03i" st.wHour st.wMinute st.wSecond st.wMilliseconds)

int var=5
s.timeformat(F"{var}. {{DD}}") ;;with F string use {{}} for date/time parts
```

Remove characters from the beginning or/and end

Syntax

```
s.ltrim([char|chars])  
s.rtrim([char|chars])  
s.trim([char|chars])
```

Parameters

s - str variable.

char - character to remove. Integer [character code \(239\)](#). Default: space and other white characters.

chars - set of characters to remove. String.

Remarks

ltrim removes the specified characters from the beginning of **s**.

rtrim removes the specified characters from the end of **s**.

trim removes the specified characters from the beginning and end of **s**.

In [Unicode \(267\)](#) mode, don't use non ASCII characters as **char** or in **chars**.

Example

```
str s = " String "  
s.trim  
now s is "string"  
s.trim('g')  
now s is "strin"  
s.trim("stn")  
now s is "Stri"
```

Convert string to/from Unicode UTF-16

Syntax

```
s.unicode([ss] [codepage] [length])
s.ansi([ss] [codepage] [length])
```

Parameters

s - str variable.

ss - subject string. Default: **s**. With [ansi](#), it can be string or word* or BSTR.

codepage (QM 2.3.0) - [code page identifier](#). If omitted or negative, uses the code page that QM uses everywhere, which depends on whether QM is running in Unicode mode. Read more in Remarks.

length - number of characters to get from **ss**. If omitted or negative, gets whole **ss**. With [unicode](#) it must be number of bytes (even if there are multibyte characters). With [ansi](#) it must be number of 2-byte characters (even if there are 4-byte characters).

Remarks

To store [Unicode \(267\)](#) text, often is used UTF-16 format, where characters consist of 2 bytes (sometimes 4). It is used with most COM functions, with Windows API functions whose names end with W, and with many other functions. However QM functions don't work with UTF-16 strings. They work with ANSI or UTF-8 strings. Therefore sometimes it is necessary to convert from/to UTF-16. Although normally str variables store ANSI or UTF-8 strings, they also can store UTF-16 strings.

[unicode](#) converts **ss** from ANSI or UTF-8 to UTF-16, and stores the result into **s**.

[ansi](#) converts **ss** from UTF-16 to ANSI or UTF-8, and stores the result into **s**.

QM 2.3.0. A BSTR variable can be simply passed to [ansi](#). Previously you would have to use its pstr member.

If **ss** is a variable containing binary data (null characters), the functions get only the part of it until the first null character, unless you explicitly specify **length**.

Variables of [BSTR \(171\)](#) type store text in UTF-16 format. To convert to/from BSTR, also can be used operator =. It uses default code page (CP_ACP in ANSI mode, CP_UTF8 in Unicode mode). Unlike [ansi/unicode](#), it gets binary data too.

To convert to UTF-16 when calling dll functions, it is more convenient to use [operator @ \(134\)](#). See example.

Note: For historical reasons these functions are incorrectly named, because UTF-8 actually is Unicode too. A better name for [unicode](#) would be something like [toutf16](#), and for [ansi](#) - [fromutf16](#). QM versions before QM 2.3.0 did not support UTF-8, so these names were good.

If QM is running in ANSI mode (Unicode unchecked in Options), default code page is CP_ACP (0). It is the current system Windows ANSI code page. To see the actual value you can use [GetACP](#). If QM is running in Unicode mode (Unicode checked in Options), default code page is CP_UTF8. It is Unicode encoded in UTF-8 format.

To convert string from one ANSI or UTF-8 encoding to another ANSI or UTF-8 encoding, you can use [str](#) function [ConvertEncoding](#) or [LoadUnicodeFile](#). See example.

See also: [_unicode variable \(144\)](#), [ANSI and ASCII characters \(239\)](#), [code page identifiers](#)

Examples

```
int h; str s1 s2 s3
s1="QM_Editor"
s2="QM_Editor"
```

[call a function that uses UTF-16 string as an input parameter](#)

```
h=FindWindowW(+s1.unicode 0)
```

[or you can use operator @](#)

```
h=FindWindowW(@s2 0)
```

[or you can use operator L, but only with string constants](#)

```
h=FindWindowW(L"QM_Editor" 0)
```

call a function that uses UTF-16 string as an output parameter

```
BSTR b.alloc(300)
```

```
GetWindowTextW h b 300
```

```
s3.ansi(b) ;;note: don't use s3=b because it gets whole buffer
```

```
out s3
```

convert from current QM encoding to iso-8859-1

```
s1.ConvertEncoding(_unicode 28591)
```

Character codes

Strings (text) consist of characters (letters, digits, spaces, etc). A character is stored as a numeric value - character code. Different values are displayed as different characters.

There are various character sets. Character codes in ANSI character sets consist of single byte. Character codes in the [Unicode \(267\)](#) character set consist of 1-4 bytes (depends on encoding). Characters in range 0 to 127, also called ASCII characters, are the same in all character sets. They are listed in the tables below. To see all characters in all character sets, use Windows program Character Map ([run "charmap"](#)).

Most of characters in range 0 to 31 are rarely used. Often used are these characters:

Code	Character
0	Terminating null character. Used at the end of a string.
9	Tab.
10	New line, also called linefeed.
13	Carriage return. On Windows, character sequence 13 10 is used for a new line.

Other ASCII characters:

Code	Char	Code	Char	Code	Char
32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	\$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(72	H	104	h
41)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[123	{
60	<	92	\	124	
61	=	93]	125	}
62	>	94	^	126	~
63	?	95	_	127	unused

QM uses the Unicode character set if the checkbox is checked in Options. Otherwise it uses the default ANSI character set.

Characters in the Unicode character set are the same on all computers. Note that not all fonts support all characters.

ANSI characters in range 128 to 255 are different (different character is displayed for the same character code) in different ANSI character sets (Western, Baltic, Cyrillic, Greek, etc). The default ANSI character set on a computer depends on the language that is set when installing Windows or changed in Control Panel -> Regional -> Advanced or Administrative -> Language for non-Unicode programs. Therefore these characters can be displayed differently on different computers. You can see characters in various character sets using a Windows program called "Character Map". You can use the first macro in the examples (below) to see how these characters look on your computer.

In a macro, to get character code of an ASCII character, enclose the character in single quotes. For example, `out 'A'` will display 65. To get character code of a character in a string variable, use operator `[]`. Example: `str s="ABC"; out s[0]; out s[1]`. To insert an ASCII character into a string using character code, use operator `[]`: `str s=" "; s[0]='A'`, or `format field (248) %c` with `str.format (213)` or other function that supports formatting: `str s.format ("%c" 'A')`.

Color

A color is stored as integer value in format 0xBBGGRR. The low-order byte contains a value for the relative intensity of red; the second byte contains a value for green; and the third byte contains a value for blue.

Examples:

0x000000 - black
 0xFFFFFFFF - white
 0x0000FF - red
 0x00FF00 - green
 0xFF0000 - blue
 0x00FFFF - yellow (red+green)
 0xC0C0C0 - gray
 0x808080 - dark gray
 0x008000 - dark green

To compose a color at run time from red, green and blue components in range 0 to 255, use function [ColorFromRGB](#).

Example: `yellow=ColorFromRGB(255 255 0)`.

To get a system color, use function [GetSysColor](#). Example: `ttcol=GetSysColor(WINAPI.COLOR_INFOTEXT)`.

To parse color to red, green and blue components, use function [ColorToRGB](#).

When capturing icon in the Icon Editor, you can see the color of the top-left pixel.

The following macro displays the color from the mouse pointer.

```
str s.format("0x%X" pixel(xm ym))
out s
```

Coordinates

Many functions ([lef](#), [rig](#), [mid](#), [dou](#), [mou](#), [win](#), [child](#), [pixel](#), [mov](#), [siz](#), [wait](#)) use coordinates **x** and **y** to specify a point in the screen or a window.

With mouse commands, [pixel](#) and [wait](#), if **window** is omitted or literal 0, coordinates are relative to the top-left corner of the screen or the work area. Otherwise, they are relative to the window or to the client area of the window. Window's client area includes all except border, caption bar, scroll bars and standard menu bar. The work area is the portion of the screen not obscured by the system taskbar or by application desktop toolbars.

If coordinates are integer values, they are interpreted as number of pixels. A pixel is the smallest unit that can be displayed on a monitor. It depends on screen resolution. For example, resolution of 1024x768 means 1024 pixels in horizontal and 768 pixels in vertical. When you move the mouse, in QM status bar you can see mouse pointer coordinates in pixels.

If coordinates have type of double (usually from 0.0 to 0.999) then they are interpreted as fraction of the screen, window, client area or child window. For example, `mou 0.5 0.5 child("OK" "Button" "Message")` moves the mouse to the middle of OK button. Double coordinates cannot be used with user-defined and dll functions.

Usually screen coordinates must be relative to the primary monitor. If you want to use coordinates relative to some other monitor, use function [XyMonitorToNormal](#) to convert them.

Declarations

Beside [QM intrinsic functions \(52\)](#), you can also use functions from Windows dlls, as well as from other dlls (e.g. downloaded). Dll functions, together with types and constants that are used with them, are called *Application Programming Interface*, or *API*. Windows API are documented in [MSDN library \(256\)](#). Before using, they must be declared. Otherwise QM would be unable to find functions, perform type checking, retrieve values of constants, etc.


















You can place declarations in the same macro/function where used, or in other functions that were executed or [compiled \(174\)](#) before. For example, you can place them in [init2](#) function (if does not exist - create), which is executed at startup, or in some function that has "QM file loaded" trigger. It is not error if identical declarations are in more than one place. Read [more \(257\)](#).


All declared identifiers have global scope (except local and thread variables), and can be used in any macro. They are listed in the [popup list \(46\)](#) that appears when you type dot (.). In code they are colored. You also can see them in QM status bar.

Some Windows API functions, types, constants and interfaces are declared by default. You don't need to declare them. Some of them are declared internally, others in the System folder. Many other declarations are in WINAPI and WINAPIV [reference files \(160\)](#), and you also don't have to declare these identifiers. In the QM forum you can find and download another file that contains more declarations.

Many other declarations are in other reference files and in [type libraries \(164\)](#). You don't have to explicitly declare those identifiers. Just type a library name, dot (.), and double click the identifier in the popup list. That is, use syntax `libname.identifier`. QM reference files are supported only by QM, but type libraries are supported by most programming languages. Type libraries are mostly used to describe COM components. Many type libraries are already installed. Installed type libraries are listed in the COM Libraries dialog. Many free and commercial components with/and type libraries are available on the Internet. Type libraries and reference files must be declared ([typelib \(164\)](#), [ref \(160\)](#)), unless they are declared by default. Declared reference files and type libraries are listed at the bottom of the global popup list.

Below are listed all kinds of identifiers supported by QM.

	Declaration	Name examples
 QM intrinsic functions (52)		<code>lef, if</code>
 User-defined functions	function (152)	<code>MyFunction</code>
 Dll functions	dll (153)	<code>GetCursorPos</code>
 QM intrinsic types (140) , OLE types (145)		<code>int, BSTR</code>
 User-defined types	type (154)	<code>LOGFONT</code>
 Classes	class (157)	<code>Ftp</code>
 COM (162) classes	Declared in type libraries (164) .	<code>WshNetwork</code>
 COM interfaces	Declared in type libraries. Also can be used interface (165) .	<code>IHTML element</code>
 Named constants	def (139)	<code>WM_USER</code>
 COM type libraries	typelib (164)	<code>Shell32</code>
 API reference files	ref (160)	<code>WINAPI</code>
 Categories	category (159)	<code>internet</code>
Compiler directives (52)		<code>#ifdef</code>
 Variables	Type name (141)	<code>i</code>
 Member functions of intrinsic types (52)		<code>getwintext</code>
 Member functions of classes (157)	Name is declared by using special syntax in QM item name. To declare type and parameters, use function (152) .	<code>SetValue</code>
 Methods (168) (member functions of COM interfaces)	Declared in type libraries. Also can be used interface (165) .	<code>GoBack</code>
 Properties (168) (member functions of COM)	Declared in type libraries. Also can be used interface (165) .	<code>Height</code>

interfaces)		
 Events (169) (user-defined functions called by COM object)	Declared in type libraries.	a_Activate

These identifiers can be declared in type libraries: COM classes, COM interfaces, user-defined types, dll functions, named constants, enumerations. Enumerations are not used in QM, but constants from them can be used.

These identifiers can be declared in reference files: dll functions, user-defined types, classes, categories, COM interfaces, named constants, type libraries.

DPI-scaled windows

On Windows Vista and later, in Control Panel -> Display you can make text and other items in all windows bigger (125%, 150%...). Let's call it *DPI scaling*. When DPI scaling is not 100% and not using XP-style scaling, some windows are stretched and look a little blurry. Let's call these windows *DPI-scaled windows*.

DPI-scaled windows is a problem for automation programs like QM. QM solves most of these problems in its internal functions. This topic is about problems that you may still have with DPI-scaled windows.

Windows Vista, 7 and 8.0

On these OS, with DPI-scaled windows Windows API functions use coordinates of non-scaled window, and they are smaller than the window view that you see. These coordinates are called *logical*. The scaled/stretched view is in *physical* coordinates.

QM 2.3.6 and later with DPI-scaled windows uses physical coordinates (what you see), and it works well. Older QM versions used logical coordinates in most cases, and it did not work well. If in older QM you have created macros that use logical coordinates (x, y, width, height) in DPI-scaled windows, in new QM these macros will not work. Need to replace logical coordinates with physical.

If in macros you use Windows API functions, and want to make the macros work well with DPI-scaled windows, replace the API functions with similar QM functions. Note that QM dialogs and other QM windows cannot be DPI-scaled, therefore you can safely use Windows API functions with them.

Windows API function	QM function
GetCursorPos	xm (55)
GetWindowRect , GetClientRect	DpiGetWindowRect
ScreenToClient	DpiScreenToClient
ClientToScreen	DpiClientToScreen
MapWindowPoints	DpiMapWindowPoints
MoveWindow , SetWindowPos	mov, siz (75)

Windows 8.1 and 10

On these OS, most Windows API functions use physical coordinates with all windows. Therefore the above QM functions ([DpiGetWindowRect](#) etc) on these OS don't convert physical/logical coordinates. You can still use them to make your code work well on all OS.

QM has problems with DPI-scaled windows on these OS:

- Most accessible objects lie about their coordinates.
- QM does not work well with windows that are on monitors that use different DPI scaling than the primary monitor. QM "child window from point" functions get wrong control. QM "window text" functions use wrong coordinates. And more.
- QM does not work well after DPI scaling is changed, until you logoff/logon. Some QM functions get/use wrong coordinates of UI objects etc, even in some non-DPI-scaled windows.
- And possibly more.

You can disable DPI scaling for a program in its file Properties dialog, Compatibility tab. To disable for all programs, in Control Panel -> Display check "...one scaling..." and choose 125%. On older OS - check XP-style.

QM DpiX functions

Added in QM 2.3.6. The functions work with all windows. They use physical coordinates (what you see) with all windows.

```
#DpiIsWindowScaled hwnd ;;if hwnd 1, gets is scaling enabled
```

Returns 1 if window is DPI-scaled, 0 if not.

hwnd - window handle. If **hwnd** is 1, returns 1 if DPI scaling is enabled, 0 if not.

Often this function is used like this: `if (DpiIsWindowScaled(w)) ... DpiScale(...)`. You may want to disable such

code on Windows 8.1 and later, if related Windows API functions on these OS use physical coordinates. Use code like this:

```
if (winver < 0x603 and DpiIsWindowScaled(w)) ... DpiScale(...). What is \_winver and 0x603 \(144\).
```

```
#DpiGetWindowRect hwnd RECT*r [flags] ;;flags: 4 client, 8 client in screen
```

Gets window rectangle in screen, or window client area rectangle, or window client area rectangle in screen.

hwnd - window handle.

r - address of variable that receives physical rectangle coordinates.

Returns 2 if the window is DPI-scaled, 1 if not, 0 if failed.

```
#DpiClientToScreen hwnd POINT*p [flags] ;;flags: 16 window, 0x100 RECT
```

```
#DpiScreenToClient hwnd POINT*p [flags] ;;flags: 16 window, 0x100 RECT
```

Converts point coordinates in window client area to coordinates in screen, and vice versa.

If flag 16, converts point coordinates in window to coordinates in screen, and vice versa.

hwnd - window handle.

p - address of variable that contains physical point coordinates.

- If flag 0x100, converts rectangle. Then **p** must be address of a RECT variable.

Returns 2 if the window is DPI-scaled, 1 if not, 0 if failed.

```
DpiMapWindowPoints hwnd1 hwnd2 POINT*p n [flags]
```

Converts coordinates of **n** points from client area of one window to client area of another window.

hwnd1 - handle of window where coordinates are before calling this function. Screen if 0.

hwnd2 - handle of window where coordinates are after calling this function. Screen if 0.

p - address of one or more **POINT** or **RECT** variables that contain physical coordinates.

n - number of points to convert. Must be 1 if **p** is address of single **POINT** variable. Must be 2 if **p** is address of single **RECT** variable, like `+&r`.

```
DpiScale POINT*p n ;;n: >0 scale, <0 unscale
```

Converts coordinates of one or more points from logical to physical or vice versa.

p - address of one or more **POINT** or **RECT** variables that contain coordinates.

n - number of points to convert. Must be 1 if **p** is address of single **POINT** variable. Must be 2 if **p** is address of single **RECT** variable, like `+&r`. If **n** is negative, converts from physical to logical.

This function simply multiplies coordinates. For example, if text size is 125% and p.x is 100, the function makes it 125 if **n**>0, or 80 if **n**<0. Works always, regardless of system DPI settings and version.

On Windows 8.1 and later, multiple monitors can have different DPI. This function uses DPI of the primary monitor.

```
#DpiGetDPI
```

Returns text size in DPI (dots-per-inch). Text size 100% is 96 DPI, 125% is 120 DPI, and so on. Works even if text size is 100% or using XP style scaling.

On Windows 8.1 and later, multiple monitors can have different DPI. This function gets DPI of the primary monitor. To get DPI of any monitor, use Windows API [GetDpiForMonitor](#).

With all functions that have **flags** parameter, can be used flag 1 if you know the window is not DPI-scaled, and 2 if DPI-scaled. It makes the function faster.

Expressions

An expression is something that has some value (variable, constant) or returns some value as result of function call (e.g., `Func(a b)`) or calculation with operators (e.g., `a + b`).

You can assign expressions to variables (variable = expression), return from functions (`ret expression`), use expressions as arguments of commands/functions, and as parts of other expressions. Expressions that contain operators and spaces must be enclosed in parentheses when passing to a function.

Examples:

```
a = 10
b = a
s = "string"
a = b + (c * 5) - 1
def C5 (c * 5)
a = b + C5 - 1
a = Func(b c)
a = Func2(b 1.5 (c + 1) Func(d 10) s)
a = Func2(b 1.5 c+1 Func(d 10) s)
ret a + 10
out Func(a b)
```

Here a, b, c, d and s are variables; `Func` and `Func2` are functions.

F1 help and the tips/output pane

F1 in the code editor

When you type or click an identifier (function, type, etc), and then press F1, QM shows help for it. The table shows where QM looks for help and where shows it.

Identifier	On F1
QM intrinsic function, type, etc	Opens its topic in QM Help.
User-defined function	Displays function's help section in the Tips pane.
API (dll function, type, constant, interface)	<ul style="list-style-type: none"> • If declared in a type library, opens its help file, if available. • Else looks for " /?" in the macro where declared, at the beginning of a line, except if declared in a reference file. If there is " /?", opens that macro. If there is " /? macro name", shows that macro in Tips. • Else shows help search links in Tips.
Class name	Looks for macro named "classname help" and displays it in Tips. If does not find, processes " /?" like with API.
Unknown (not currently declared)	Shows help search links in Tips.

Function's help section

User-defined functions can contain comments at the beginning or below the `function` statement. It is function's *help section*.

```
function x y ;;this line is displayed in QM status bar
```

These comments, including the 'function...' line, is function's help section.
 It ends with an empty line followed by code.
 It is displayed in Tips when you click the function in code editor and press F1.
 The first line of comments is displayed in status bar and annotations (menu -> Help -> Annotations).

To create help section, you can use: floating toolbar -> More Tools -> Function help editor.

In Tips can be displayed functions, macros and [QM items \(19\)](#) of other types.

- For functions, shows only help section.
- For macros, shows all text as help section. Code below `EXAMPLES` line should not be commented.
- For other items, shows all text as code.

To show help, you can use F1 or function `QmHelp`.

The help section can contain tags and special lines. Use them to format the text when it is displayed in Tips.

Special lines in help section

Certain text lines in the help section can be used for text formatting. To display colored code, bold/italic headings, etc.

Special lines	Formatting
<code>EXAMPLE</code> <code>EXAMPLES</code>	Below this line, text is displayed as code (colored). <ul style="list-style-type: none"> • To insert code in other text, use <code><code>...</code></code>.
<code>See also:</code>	In this line, function names enclosed in <code><></code> (like <code><Function></code>) make help links. <ul style="list-style-type: none"> • To make help links in other text, use <code><help>Function</help></code>.
<code>param -</code> <code>param (...</code> <code>param:</code> <code>param, ...</code>	Parameters are highlighted.
<code>Errors:</code> <code>ERRORS</code>	In or below this line, error constants (such as <code>ERR_FAILED</code>) make links to the error string. <ul style="list-style-type: none"> • If this line is missing, QM extracts errors from code. Extracts values used with end (131) statement

in this function. Only if not encrypted.

- To add errors of other functions, include function names enclosed in < >.
- Use empty <> to add errors of this function (extract from code).
- QM 2.3.5. Use <.> to add errors of all called functions. Use <..> to add errors of this and called functions.

Example: **Errors:** <>, <Func1>, <Func2>

UPPERCASE

Bold text. For example, **REMARKS** is displayed **Remarks**.

Returns:

If a line begins with this text, this text is italic.

Errors:

See also:

Version:

Added in:

Author:

All except **EXAMPLE(S)** and **See also:** added in QM 2.3.3.

Example.

```
/
function# $s flags [str&so]
```

What it does ...

Returns: ...

s - ...

flags:

1 - ...

2 - ...

so (v2.5) - ...

REMARKS

...

...

EXAMPLE

str s

FunctionName "abc" 1 s

out s

Tags in help section and output text

Tags can be used in help sections of user-defined functions and macros. To insert tags, you can use: floating toolbar -> More Tools -> Function help editor. Tags are replaced with links, colors etc when the function help is displayed in the Tips pane, for example when you press F1 in a macro when the text cursor is on that function name.

QM 2.3.0. Tags also can be used with [out \(57\)](#), if text begins with "<>". Also can be used with [end \(131\)](#), without "<>".

QM 2.3.3. Tags also can be used with [OutStatusBar](#).

Tags also can be used in API help search links. To edit, right click in the Tips pane.

Syntax for most tags: `<tag>text</tag>`, or `<tag "attribute">text</tag>`, or `<tag "attribute /more info">text</tag>`.

Attributes can contain [QM escape sequences \(137\)](#). For example, use two ' for ". If single word, can be without quotes (QM 2.4.1).

Links

```

<link "http://www.quickmacros.com">open web page</link>
<link "mailto:abc@def.gh?subject=test%20qm%20links&body=test%0Aqm%0Alinks">create email</link>
<link "notepad.exe">run file notepad.exe</link>
<link "notepad.exe /$desktop$\test.txt">run notepad with command line parameters</link>
<macro "MyMacro">run macro MyMacro</macro>
<macro "MyMacro /abc">run macro MyMacro; the _command variable will contain "abc"</macro>
<open "MyMacro">open MyMacro</open>
<open "MyMacro /10">open MyMacro and go to position 10</open>
<open "MyMacro /L10">open MyMacro and go to line 10 (1-based).</open> QM 2.3.2.
<open "MyMacro:Sub">open MyMacro and go to sub-function Sub</open> QM 2.4.1.
<help "::-/qm_help/IDH_QUICK.html">open a topic in QM Help</help>
<help "#IDH_QUICK">open a topic in QM Help</help> QM 2.4.1
<help "#IDP_QMDLL#PerfOut">with anchor</help> QM 2.4.1
<help "qm2help.chm">open help file qm2help.chm</help>
<help "qm2help.chm::-/qm_help/IDH_QUICK.html>main">open a topic in a help file, "main" window type</help>
<help>MyFunction</help> QM 2.3.3. Shows function help. Read more below.
<tip "E_IF">display a tip from $qm$\tips.txt</tip>
<tip "#MyMacro">display a macro or function (only help section) in tips</tip>
<google "quick macros">Google search for 'quick macros'</google> QM 2.3.3.
<google "quick macros /&start=10&lr=lang_de">Google search with parameters</google> QM 2.3.3.
<mes "text">show message box</mes> QM 2.3.4
<out "text">show text in QM output</out> QM 2.3.4

```

Instead of `<tag "attribute">text</tag>` can be `<tag>text</tag>`. Then text is used as attribute. Example:

```
<link>http://www.quickmacros.com</link>
```

QM 2.3.2. Instead of `<tip "#MyMacro">MyMacro</tip>` can be `<tip>MyMacro</tip>`.

QM 2.3.3. With `<help>` can be used any function or other identifier. Shows help as if you would press F1 in code editor.

Examples: `<help>MyFunction</help>` `<help>act</help>` `<help>SetTimer</help>` `<help>WM_TIMER</help>`
`<help>str.replacex</help>` `<help>My Macro</help>`.

Styles

```

<b>bold</b> <i>italic</i> <u>underline</u>
<c "0x00ff00">green color</c>
<b><i><c "0xff">nested tags</c></i></b>
<c 0x8000>nested<c 0xff0000>tags</c>again</c>
<z "0xff">red background</z> QM 2.3.3
<Z "0xff">red background full line</Z> QM 2.3.3
<hidden>cannot see this, but can copy or get with scintilla API</hidden> QM 2.4.1

```

Code

Macro or function:

```

<code>
int i
for(i 0 5) out 1
</code>

```

Menu:

```

<code "1">
abc :run "abc.exe" * icon.ico
abc :run "abc.exe" * icon.ico
</code>

```

Autotext list:

```

<code "2">
abc :key "abc"
abc :key "abc"
</code>

```

Images (QM 2.4.1)

Images are displayed below the text line containing image tags.

```
<image "resource:<My Macro>image:Name"></image> image from macro resources
<image "resource:<>image:Name"></image> image from file resources
<image "image:Name"></image> the same as above
<image "c:\planets\venus.bmp"></image> image file. Can be bmp, png, jpg, gif.
<image "$my qm$\small.ico"></image> 16x16 icon
<image "&big.ico"></image> 32x32 icon in $my qm$ folder
<image "$system$\shell32.dll,5"></image> icon in a dll, exe or icl file
<image "c:\MyArrow.cur"></image> 32x32 cursor. Can be cur, ani.
<image "c:\file.html"></image> shell icon of any file
<image "resource:<My Macro>Name.png"></image> Can be bmp, png, jpg, gif, ico, cur, ani.
```

Example

```
out
Dir d
foreach(d "$program files\*.png" FE_Dir 4)
  _str path=d.FullPath
  _out F"<><image '{path}'>{path}</image>"
  ifk(C) break ;;end when Ctrl pressed
```

You can add image files to macro resources with `_qmfile.ResourceAdd`. Adds to file resources if the first argument is 0.

Ordinary text (ignore tags)

```
<_> Text with <b>tags</b> that are <u>ignored</u>. </_>
```

Replacements

The word for which pressed F1. Only in API help search links:

```
<keyword>
<macro "MyMacro /<keyword>">run macro MyMacro; the _command variable will contain the F1 word</macro>
```

File paths

Special folders

With all QM file functions ([run](#), [cop](#), [ren](#), [del](#), [FileExists](#), `str.searchpath`, `str.getfile` and other, also in user-defined functions) and in menus/toolbars, a file path can begin with a special folder string. There are several ways to specify a special folder.

1. QM-defined special folder string enclosed in \$.

- Example: "\$desktop\$\file.txt".
- To see available special folders, click SF button in a dialog, for example in the Run Program dialog.
- QM-specific special folders:
 - \$qm\$ - Quick Macros program folder, eg "C:\Program Files (x86)\Quick Macros 2" or "G:\PortableApps\QuickMacrosPortable\App\QuickMacros".
 - \$my qm\$ - Quick Macros data folder, eg "C:\Users\Me\Documents\My QM" or "G:\PortableApps\QuickMacrosPortable\Data\My QM". Can be changed in Options -> Files.
 - \$temp qm\$ - QM subfolder in user's temporary folder (special folder \$temp\$). Normally it is same as \$temp\$\QM. In portable QM it is same as \$temp\$\QM-portable, and QM deletes it when exits. The folder may not exist, but most QM file functions create it when creating a file or folder in it.
 - \$drive\$ - drive of Quick Macros program folder, like "C:", "G:" or "\\server\share". This can be useful in [portable \(259\)](#) QM, where drive letter is not constant.
- [Special folder CSIDLs](#).

```
{CSIDL_ADMINTOOLS, "Administrative Tools"},
{CSIDL_APPDATA, "AppData"},
{CSIDL_INTERNET_CACHE, "Cache"},
{CSIDL_CDBURN_AREA, "CD Burning"},
{CSIDL_PROGRAM_FILES_COMMON, "Common Files"},
{CSIDL_COOKIES, "Cookies"},
{CSIDL_DESKTOPDIRECTORY, "Desktop"},
{CSIDL_PERSONAL, "Documents"},
{CSIDL_PERSONAL, "My Documents"}, //alias
{CSIDL_PERSONAL, "Personal"}, //alias
{CSIDL_FAVORITES, "Favorites"},
{CSIDL_FONTS, "Fonts"},
{CSIDL_HISTORY, "History"},
{CSIDL_LOCAL_APPDATA, "Local AppData"},
//{?, "Local Settings"}, //no csidl. Dropped in QM 2.3.0.
{CSIDL_MYMUSIC, "My Music"},
{CSIDL_MYPICTURES, "My Pictures"},
{CSIDL_MYVIDEO, "My Video"},
{CSIDL_NETHOOD, "NetHood"},
//{CSIDL_PHOTOALBUMS, "Photoalbums"}, //Vista
//{CSIDL_PLAYLISTS, "Playlists"}, //Vista
{CSIDL_PRINTHOOD, "PrintHood"},
{CSIDL_PROGRAM_FILES, "Program Files"},
{CSIDL_PROGRAM_FILES, "PF"}, //alias
{CSIDL_PROGRAMS, "Programs"},
{CSIDL_RECENT, "Recent"},
//{CSIDL_RESOURCES, "Resources"}, //Vista
//{CSIDL_SAMPLE_MUSIC, "Sample Music"}, //Vista
//{CSIDL_SAMPLE_PLAYLISTS, "Sample Playlists"}, //Vista
//{CSIDL_SAMPLE_PICTURES, "Sample Pictures"}, //Vista
//{CSIDL_SAMPLE_VIDEOS, "Sample Videos"}, //Vista
{CSIDL_SENDTO, "SendTo"},
{CSIDL_STARTMENU, "Start Menu"},
{CSIDL_STARTUP, "Startup"},
//{CSIDL_ALTSTARTUP, "AltStartup"}, //?
{CSIDL_SYSTEM, "System"},
{CSIDL_QM_TEMP, "Temp"}, //no scidl. Normally CSIDL_LOCAL_APPDATA\Temp.
{CSIDL_TEMPLATES, "Templates"},
{CSIDL_PROFILE, "User Profile"}, //QM 2.3.0
{CSIDL_WINDOWS, "Windows"},
{CSIDL_COMMON_ADMINTOOLS, "Common Administrative Tools"},
{CSIDL_COMMON_APPDATA, "Common AppData"},
{CSIDL_COMMON_DESKTOPDIRECTORY, "Common Desktop"},
```

246. File paths

```
{CSIDL_COMMON_DOCUMENTS, "Common Documents"},
{CSIDL_COMMON_FAVORITES, "Common Favorites"},
{CSIDL_COMMON_MUSIC, "Common Music"},
{CSIDL_COMMON_MUSIC, "CommonMusic"}, //alias, fbc
{CSIDL_COMMON_PICTURES, "Common Pictures"},
{CSIDL_COMMON_PICTURES, "CommonPictures"}, //alias, fbc
{CSIDL_COMMON_PROGRAMS, "Common Programs"},
{CSIDL_COMMON_STARTMENU, "Common Start Menu"},
{CSIDL_COMMON_STARTUP, "Common Startup"},
//{{CSIDL_COMMON_ALTSTARTUP, "Common AltStartup"}, //?
{CSIDL_COMMON_TEMPLATES, "Common Templates"},
{CSIDL_COMMON_VIDEO, "Common Video"},
{CSIDL_COMMON_VIDEO, "CommonVideo"}, //alias, fbc
{CSIDL_QM_QM, "QM"},
{CSIDL_QM_MYQM, "My QM"},
{CSIDL_QM_MYQM, "MyQM"}, //alias
{CSIDL_QM_TEMPQM, "Temp QM"}, //QM 2.3.5
{CSIDL_QM_DRIVE, "Drive"}, //QM 2.3.5
```

2. [Environment variable \(143\)](#) name enclosed in %.

- Example: "%temp%\file.txt".
- Environment variables are expanded recursively (QM 2.2.0). For example, if value of environment variable v1 is "\$desktop\$", then "%v1%" is expanded to full path of desktop folder.
- Tip: You can create environment variables and then use them everywhere. Example: `SetEnvVar "backup" "E:\ Backup"`. Use: `run "%backup%"`. Such variables exist only in current process (QM or exe).

3. QM 2.2.0. CSIDL numeric value enclosed in \$. For example, "\$3\$" expands to ITEMIDLIST string (see below) of Control Panel. To see available values, search for CSIDL in [MSDN Library \(256\)](#).

See also: [str.expandpath \(231\)](#).

ITEMIDLIST strings

Some objects in Window Explorer are not file system objects. Examples are Control Panel folder and most objects in it. They cannot be accessed using a file system path. They can be accessed using binary structures of type **ITEMIDLIST**. To simplify it, some QM functions ([run](#), [menu](#) and [toolbars](#), and some other) support ITEMIDLIST strings. These strings begin with "::<". They are inserted when you drag and drop a non-file-system object. They also can be inserted using Run Program, Open File and Open Folder dialogs.

Examples:

```
run "::< 14001F50E04FD020EA3A6910A2D808002B30309D" ;;My Computer
run "$0x11$" ;;My Computer
run "::< 14001F50E04FD020EA3A6910A2D808002B30309D 14002E1E2020EC21EA3A6910A2DD08002B30309D
A90000009CFFFFFFF1D002500433A5C57494E444F57535C73797374656D33325C6465736B2E63706C00446973706
C6179004368616E67652074686520617070656172616E6365206F6620796F7572206465736B746F702C20737563
6820617320746865206261636B67726F756E642C2073637265656E2073617665722C20636F6C6F72732C20666F6
E742073697A65732C20616E642073637265656E207265736F6C7574696F6E2E00" ;;Display
run "$3$"
A90000009CFFFFFFF1D002500433A5C57494E444F57535C73797374656D33325C6465736B2E63706C00446973706
C6179004368616E67652074686520617070656172616E6365206F6620796F7572206465736B746F702C20737563
6820617320746865206261636B67726F756E642C2073637265656E2073617665722C20636F6C6F72732C20666F6
E742073697A65732C20616E642073637265656E207265736F6C7574696F6E2E00" ;;Display
run "::< " ;;Desktop
```

The root object is Desktop. Its string is "::<". It is not the same as "\$desktop\$", which is a file system folder.

File system objects also can be accessed using ITEMIDLIST. To insert ITEMIDLIST string, Shift + drag and drop.

See also: [PidlToStr \(114\)](#), [PidlFromStr \(114\)](#).

File search paths, relative paths

Some functions ([run](#), [str.searchpath](#), [GetFileIcon](#), etc) don't require full path. Can be used only filename or relative path. Then they search in:

1. The QM directory. Before QM 2.3.3, [run](#) did not search here.
2. The current directory ([GetCurDir/SetCurDir](#)).
3. The 32-bit Windows system directory.
4. The 16-bit Windows system directory.
5. The Windows directory.
6. The directories that are listed in the PATH environment variable.
7. [run](#) and `str.searchpath` also search in the registry "App Paths" key.

Some functions, including `str.searchpath` and [GetFileIcon](#), at first search in \$My QM\$ special folder.

With some functions, `.\` at the beginning means current directory; `..\` at the beginning means one level up from current directory. For example, if current directory is "c:\program files\quick macros 2", then `..\` will be "c:\program files\", and `..\otherapp` will be "c:\program files\otherapp". Also, whole path can be or contain "." and "..".

Read more: [File names and paths](#)

Shortcuts

When you drag and drop a shortcut to the QM code editor, it inserts [run](#) with path of shortcut target. To insert shortcut path, Ctrl + drag and drop.

Notes

File functions may fail with mapped network drive paths (like "Z:\file"). Always use path like "\\server\share\file".

64-bit Windows has two System32 and Program Files folders. [Read more \(277\)](#).

Flags

Many functions have parameters, often called "flags", that can consists of several values (bits). To create flags from several values, use operator | (bitwise or). Examples:

```
int hwnd=win("Window" "" "" 1|8)
SetWindowPos(hwnd HWND_TOPMOST 0 0 0 0 SWP_NOSIZE|SWP_NOMOVE|SWP_SHOWWINDOW)
```

Usually flag values are divisible by two (1, 2, 4, 8, 16, and so on). If so, the | operator is the same as the + operator. For example, to specify flags 1, 4 and 16, you can use 1|4|16, or you can use 21. Flags usually are easier to read when they are in hexadecimal format, so instead of 21 you can use 0x15. Another example: 5|0x10.

A flag takes 1 bit, therefore an int value can have 32 flags: 1, 2, 4, 8, 0x10, 0x20, 0x40, 0x80, 0x100, ... 0x80000000.

If flags is an optional parameter of a function, its default value is 0.

How to compare flags

Usually you need to know whether one of flags is set. Use operator &. Example: `if(flags&8)`.

Don't use operator = (for example, `if(flags=3)`), usually it has no sense.

How to read flags from a hexadecimal number, and vice versa

QM dialogs often create code that includes flags, usually in hexadecimal format, like 0x3011. How to know what flags are encoded in the number?

At first, need to understand hexadecimal number format:

<https://en.wikipedia.org/wiki/Hexadecimal>

https://en.wikipedia.org/wiki/Flag_field

Each hexadecimal digit (after 0x) is a sum of max four flags at that digit position: 1, 2, 4 and 8.

Examples:

0x5 contains flags 0x4 (4) and 0x1 (1).

0x500 contains flags 0x400 (1024) and 0x100 (256).

0x52A contains flags 0x400, 0x100, 0x20, 0x8 and 0x2.

0x3011 contains flags 0x2000, 0x1000, 0x10 and 0x1.

Table:

Digit - flags

0 - none

1 - 1

2 - 2

3 - 1, 2

4 - 4

5 - 4, 1

6 - 4, 2

7 - 4, 2, 1

8 - 8

9 - 8, 1

A - 8, 2

B - 8, 2, 1

C - 8, 4

D - 8, 4, 1

E - 8, 4, 2

F - 8, 4, 2, 1

To add a flag to a hexadecimal number containing flags, simply add the flag value at that position if it does not exist. In code we use operator |, not +. This is to avoid adding when already exists etc.

Example: add flag 0x100 to 0x402: 0x402|0x100 = 0x502.

To remove a flag from a hexadecimal number containing flags, simply subtract the flag value at that position if it exists. In code we use operator &~ (or just ~ in QM), not -.

Example: remove flag 0x100 from 0x5C0: 0x5C0&~0x100 = 0x4C0.

247. Flags

Don't translate a hexadecimal flags number like 0x3011 to decimal, it will not have sense. But can translate a single flag, for example to write shorter, eg can write 4 instead of 0x4, or 16 instead of 0x10.

Format fields

Format fields are used in strings with functions [str.format \(213\)](#), [out](#), [paste](#) and [operator F \(138\)](#). A format field is a special character, or several characters, preceded with %. The whole string is followed by variables or other values that will replace the format fields. The format fields tell how to format the values.

Syntax

`%[flags][width][.precision][h|l|I64|L]type`

type - character that determines whether the argument is interpreted as a character, a string, or a number.

flags - character or characters that control justification of output and adding of signs, blanks, decimal points, and octal and hexadecimal prefixes.

width - number that specifies the minimum number of characters in output.

precision - number that specifies the maximum number of characters used for all or part of the output field, or the minimum number of digits used for integer values.

h, l, I64, L - character that specifies argument size. For arguments of type long, use I64. Note: here l is uppercase i, not lowercase L.

Remarks

%% in format strings is replaced with %.

Almost everything is the same as with C/C++ printf, sprintf and similar functions. Differences:

- Does not add "(null)" for null strings.
- QM 2.3.3. Supports binary data (%m).

Example

```
int i=50
str s="stringvar"
double d=3.1415926535897932384626433832795
str f
f.format("variables: i=%i, s='%s', d=%.10G" i s d)
out f ;;variables: i=50, s="stringvar", d=3.141592654
```

Type

type is a character that determines whether the argument is interpreted as a number, string, character, etc.

Char.	Type	Output format
i, d	integer	Signed decimal integer. Example: <code>int i=5; out "i=%i" i</code>
u	integer	Unsigned decimal integer.
x	integer	Unsigned hexadecimal integer, using "abcdef."
X	integer	Unsigned hexadecimal integer, using "ABCDEF."
e	double	Signed value having the form [-]d.dddd e [sign]ddd where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or -.
E	double	Identical to the e format except that E rather than e introduces the exponent.
f	double	Signed value having the form [-]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
g	double	Signed value in f or e format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than -4 or greater than or equal to the precision field. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	double	Identical to the g format, except that E, rather than e, introduces the exponent (where appropriate).
s	string	Characters are added up to the first null character or until the precision value is reached.
S	word*	QM 2.3.3. Unicode UTF-16 string. Example: <code>BSTR b="utf-16 string"; out "%S" b.pstr</code>
c	integer	Character code (239) .

		Note: In Unicode mode, don't use non ASCII characters.
C	integer	QM 2.3.3. Unicode UTF-16 character code. Must be <0x10000.
m	pointer	QM 2.3.3. Binary data. Length must be specified using precision. Example: <code>s.format("%.*m" 10 ptr) ;;10 bytes from ptr</code>
m	byte	QM 2.3.3. Fill with an ASCII character. Length must be specified using precision. Example: <code>out "%.*m" 4 '*' ;;****</code> . For character code 0 use argument value 0x100.

Flags

flags is a character that justifies output and adds signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag may appear in a format field.

Flag	Meaning	Default
-	Left align the result within the given field width .	Right align.
+	Prefix the output value with a sign (+ or -) if the output value is of a signed type.	Sign appears only for negative signed values (-).
0	If width is prefixed with 0, zeros are added until the minimum width is reached. If 0 and - appear, the 0 is ignored. If 0 is specified with an integer format (i, u, x, X, o, d) the 0 is ignored.	No padding.
space	Prefix the output value with a space if the output value is signed and positive; the blank is ignored if both the space and + flags appear.	No blank appears.
#	When used with the o, x, or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No blank appears.
#	When used with the e, E, or f format, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
#	When used with the g or G format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it. Trailing zeros are truncated.
#	QM 2.3.3. When used with the s format, # specifies that precision is the number of characters, not bytes. In Unicode mode, non-ASCII characters have several bytes. Example: <code>out "%#.5s" "ačĕëišūž"</code>	

Width

width is a decimal integer controlling the minimum number of characters added. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values (depending on whether the - flag (for left alignment) is specified) until the minimum width is reached. If width is prefixed with 0, zeros are added until the minimum width is reached (not useful for left-aligned numbers).

The **width** specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified **width**, or if **width** is not given, all characters of the value are added.

If the width specification is an asterisk (*), an **int** argument from the argument list supplies the value. The **width** argument must precede the value being formatted in the argument list. Example: `out "%*s" 20 "string"`

Precision

precision is a nonnegative decimal integer, preceded by dot (.), which specifies the number of characters to be added, the number of decimal places, or the number of significant digits. Unlike the **width** specification, the **precision** specification can cause either truncation of the output value or rounding of a floating-point value.

The type determines the interpretation of **precision** and the default when **precision** is omitted, as shown in table:

Type	Meaning	Default
c, C	precision has no effect.	
d, i, u, o, x, X	precision specifies the minimum number of digits to be added. If the number of digits in the argument is less than precision , the output value is padded on the left with zeros. The value is not truncated when the number of digits	Default precision is 1.

	exceeds precision .	
e, E	precision specifies the number of digits to be added after the decimal point. The last added digit is rounded.	Default precision is 6; if precision is 0 or if . appears without a number following it, no decimal point is added.
f	precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 6; if precision is 0 or if . appears without a number following it, no decimal point is added.
g, G	precision specifies the maximum number of significant digits added.	Six significant digits are added, with any trailing zeros truncated.
s, S	precision specifies the maximum number of characters to be added. Characters in excess of precision are not added. Note: With s, precision is the number of bytes. In Unicode mode, non-ASCII characters have several bytes. In QM 2.3.3 and later you can use flag # to specify that precision must be number of characters. Example: <code>out "%#.5s" "ačĉėįšųūž"</code>	Characters are added until a null character is encountered.
m	precision specifies the number of bytes to be added.	0

Example:

```
str s=format("%.4s %.3f" "123456789" 1.12345)
out s ;;1234 1.123
```

If the precision specification is an asterisk (*), an **int** argument from the argument list supplies the value. The precision argument must precede the value being formatted in the argument list. Example:

```
str s=format("%.*s %.*f" 4 "123456789" 3 1.12345)
out s ;;1234 1.123
```

Argument size

For arguments of type long need to specify size l64. Example:

```
long k=123456789012345
out "incorrect: %i" k
out "correct: %I64i" k
```

For arguments of other types usually don't need to specify size. But anyway, here is the table:

To specify	Use prefix	With type
int	l	d, i, o, x, or X
unsigned int	l	u
signed word	h	d, i, o, x, or X
word	h	u
long	l64	d, i, o, u, x, or X
Single-byte character	h	c or C
Wide character	l	c or C
Single-byte character string	h	s or S
Wide-character string	l	s or S

GUID - globally unique identifier

Globally unique identifiers (or *universally unique identifiers*) are used to identify COM interfaces, coclasses, type libraries, etc. It is like a name that is globally unique. A GUID can be stored as a variable of type **GUID**, or as string like {XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}, where X are hexadecimal digits 0 to F. In programming, GUID sometimes is called differently: IID for interfaces, CLSID for classes, REFIID, etc.

Use [uuidof \(110\)](#) to get pointer to GUID of a COM interface or coclass, or to convert GUID string to variable. To convert variable to string, use function [str.FromGUID](#). To create new GUID, use [str.Guid](#) or [CoCreateGuid](#).

In QM 2.4.0 and later, [QM items \(19\)](#) also have GUID. QM can find items by name or GUID string. It is supported in [QM command line \(18\)](#), [qmitem \(107\)](#) and other functions. The GUID string can be followed by any text; the text is ignored.

Example: `int iid=qmitem("{a138af76-23b1-4ef8-a4e1-7f459f948dbd}Macro1")`. QM item GUID cannot be changed.

Why to use QM item GUID instead of name, for example in a command line? The main reason - the command line with GUID will still work after renaming the QM item. If using just name, after renaming also would need to edit the command line.

QM items that are in different [QM files \(17\)](#) but have the same name usually have different GUID. They can have the same GUID only in copied files.

High-order word, low-order word

Sometimes some value can be stored in high-order word and low-order word of an integer value. I'll try to explain what it is.

At first you should know [variable types \(140\)](#) int, word and byte.

int has 4 bytes. word has 2 bytes. Therefore int has 2 words.



Green - low-order word.

Blue - high-order word.

The picture shows bytes in memory, in little-endian format, where the byte order is reverse than in hexadecimal representation (0xHHHHLLLL) and bit-shift operator names.

The low-order word adds its exact value to the total value of the int. The high-order word adds its value multiplied by 0x10000 (left-shifted by 16 bits).

To store two values L and H into low-order and high-order word of integer value R, use this:

```
R = H<<16 | L
or
R = MakeInt(L H)
```

To get low-order word and high-order word:

```
L = R&0xFFFF
H = R>>16
```

High-order byte, low-order byte

Sometimes some value can be stored in high-order byte and low-order byte of a word. Everything is similar as above.

To store two values L and H into low-order and high-order byte of word value R, use this:

```
R = H<<8 | L
```

To get low-order byte and high-order byte:

```
L = R&0xFF
H = R>>8&0xFF
```

QM key codes

These key codes can be used with [key \(56\)](#) and some other functions, and in keyboard triggers (when entering directly in the toolbar).

A	Alt		O	Num Lock
B	Back		P	Page Up
C	Ctrl		Q	Page Down
D	Down arrow		R	Right arrow
E	End		S	Shift
F1-F24	F1-F24		T	Tab
G	Pause		U	Up arrow
H	Home		V	Space
I	Insert		W	Win
J	Scroll lock		X	Delete
K	Caps Lock		Y	Enter
L	Left arrow		Z	Esc
M	Context Menu			
N0-N9	Numpad 0 to 9	(44)		Print Screen
N/-N.	Numpad / * - + .			

For other keys use lowercase letters and other characters, except { } () ; ".
You can also use [virtual-key code \(270\)](#) enclosed in parentheses.

Quick Macros license agreement

You may use this software only as described in this license agreement. If you do not agree to the terms below, do not install or use the software. Here "you" means the person or organization who evaluates the software, or had purchased a license to use the software, or otherwise uses the software.

1. **EVALUATION.** You may evaluate the software for 30 days. To continue using the software, need to register it (purchase a license).
2. **LICENSE.** Once you registered the software, you are granted a license to use the software, as well as all future versions of the software.

A registered copy of the software may either be used by a single person (or family) on up to 5 computers simultaneously, or used on a single computer by multiple people.

You must not share the registration code with others (except your family members). Illegally shared registration codes will be disabled.

3. **COPYRIGHT AND OWNERSHIP.** The software is owned and copyrighted by Gintaras Didzgalvis (author). The software is protected by copyright laws. You acknowledge that no title to the intellectual property in the software is transferred to you.

4. **DISTRIBUTION.** The software may be freely distributed, provided the distribution package is not modified. No person or company may charge a fee for the software (except a distribution fee) without written permission from the author.

5. **EXE FILES.** With this software you can create .exe files. Exe files created with a registered copy of the software may be distributed without limitations.

6. **MALWARE.** You agree that you will not use the software to create and distribute macros or .exe files that may be considered as virus, spyware or other malware.

7. **REVERSE ENGINEERING.** You agree that you will not attempt to modify or disassemble the software. However you can modify the open-source parts (extensions and samples).

8. **NO OTHER WARRANTIES.** Author does not warrant that the software is error-free. Author does not warrant that future versions of the software will be fully compatible with this version. Author disclaims all other warranties with respect to the software.

9. **NO LIABILITY FOR CONSEQUENTIAL DAMAGES.** In no event shall author or its suppliers be liable to you for any damages of any kind arising out of the use of the software, even if author has been advised of the possibility of such damages.

Line breaks

On Windows, standard line break consists of two [characters \(239\)](#) - carriage return (13) followed by line feed (10). Other OS use different characters.

QM functions that deal with lines recognize Windows (13+10) and Unix/Linux (10) line breaks. Line breaks 13, Unicode line breaks and other are not recognized.

Links in mes, inp and other dialogs

QM 2.4.1. These standard dialog functions support links in static text: [mes](#), [inp](#), [inpp](#), [InputDialog](#), [ListDialog](#), [MsgBoxAsync](#), [ErrMsg](#). You can use links to show more info/help, run a file, open a web page, show a menu in [inp](#), add more choices in [mes](#), etc.

To enable links, the static text must begin with "<>". To insert links, you can use tags of 2 types:

- `link text`.
- `link text` or `link text`.

The id links are like simple buttons. On click the dialog is closed and the dialog function returns **retVal** (a nonzero number). With [mes](#), values 1-15 are mapped to button characters that match Windows API standard button constants, eg IDOK to 'O', IDCANCEL to 'C'.

The href links are used to execute any code. On click is called your callback function whose address is **callbackAddress**.

The callback function must begin with `function# hwnd $params`.

hwnd - dialog window handle.

params - tag text that follows **callbackAddress** and space.

If the callback function returns a nonzero value, the dialog is closed and the dialog function ([mes](#) etc) returns the value.

By default, text in these dialogs is displayed in Static control. Text with links is displayed in SysLink control. SysLink is unavailable in exe without manifest.

Examples

```
add more choices in message box
sel mes("<>Question.[] []Other choices:[]<a id='1000'>Maybe</a>" "" "YN?")
case 'Y' out "Yes"
case 'N' out "No"
case 1000 out "Maybe"

add popup menu in input box
str s
if(!inp(s F"<><a href='{&sub.inp_link}'>select</a>")) ret
out s

#sub inp_link
function# hwnd $params
str items=
one
two
three
int i=ShowMenu(items hwnd 0 2)
if(!i) ret
str si.getl(items i-1)
EditReplaceSel hwnd 4 si 3
```

Make macro smaller

Tips how to make macro smaller and easier to read and edit.

Reuse variables

For example, you have code like this:

```
int w=win("Calculator" "CalcFrame")
lef 31 304 w 1 ;;push button '1'
int w1=win("Calculator" "CalcFrame")
lef 68 307 w1 1 ;;push button '2'
int w2=win("Calculator" "CalcFrame")
lef 104 305 w2 1 ;;push button '3'
```

Line 1 creates variable w (`int w`), finds a window (`win(...)`), and stores window handle in w. Line 2 uses the handle (w).

If window in lines 1-2, 3-4 and 4-5 is the same, use single variable and `win`:

```
int w=win("Calculator" "CalcFrame")
lef 31 304 w 1 ;;push button '1'
lef 68 307 w 1 ;;push button '2'
lef 104 305 w 1 ;;push button '3'
```

If window is different, and will not need previous window handle later, use multiple `win` and single variable:

```
int w=win("Calculator" "CalcFrame")
lef 31 304 w 1 ;;push button '1'
w=win("Calculator" "CalcFrame")
lef 68 307 w 1 ;;push button '2'
w=win("Calculator" "CalcFrame")
lef 104 305 w 1 ;;push button '3'
```

Also you can reuse `Acc` and other variables added by QM floating toolbar dialogs. For example, replace this:

```
int w=win("Calculator" "CalcFrame")
Acc a.Find(w "PUSHBUTTON" "5" "" 0x1005)
a.DoDefaultAction
...
int w1=win("Calculator" "CalcFrame")
Acc a1.Find(w1 "PUSHBUTTON" "5" "" 0x1005)
a1.DoDefaultAction
```

to this (if need the same object):

```
int w=win("Calculator" "CalcFrame")
Acc a.Find(w "PUSHBUTTON" "5" "" 0x1005)
a.DoDefaultAction
...
a.DoDefaultAction
```

or this (if need other object and will not need previous object):

```
int w=win("Calculator" "CalcFrame")
Acc a.Find(w "PUSHBUTTON" "5" "" 0x1005)
a.DoDefaultAction
...
a.Find(w "PUSHBUTTON" "5" "" 0x1005)
a.DoDefaultAction
```

See also: [about variables \(140\)](#), [creating variables \(141\)](#)

Repeat

For example, you have code like this:

```
int w1=win("Window1")
act w1
key Cc D      ;; Ctrl+C Down
int w2=win("Window2")
act w2
key Cv D      ;; Ctrl+V Down
act w1
key Cc D      ;; Ctrl+C Down
act w2
key Cv D      ;; Ctrl+V Down
act w1
key Cc D      ;; Ctrl+C Down
act w2
key Cv D      ;; Ctrl+V Down
```

There are 3 identical 4-line code blocks. Instead use single block that is executed 3 times with [rep \(126\)](#):

```
int w1=win("Window1")
int w2=win("Window2")
rep 3
  act w1
  key Cc D      ;; Ctrl+C Down
  act w2
  key Cv D      ;; Ctrl+V Down
```

Often it's possible to repeat even when there are multiple similar but not identical code blocks. For example, replace this:

```
int w=win("Calculator" "CalcFrame")
lef 19 17 id(131 w)
0.5
lef 19 17 id(135 w)
0.5
lef 19 17 id(139 w)
0.5
```

to this:

```
int w=win("Calculator" "CalcFrame")
str ss="131[]135[]139"
str s
foreach s ss
  lef 19 17 id(val(s) w)
  0.5
```

There are 3 similar 2-line code blocks. With 10 blocks * 10 lines it would be 100 lines. With repeating (foreach) - 13 lines.

See also: [for \(127\)](#), [foreach \(128\)](#)

Use functions

When there are multiple similar but not identical code blocks, often it's better to move part(s) of code to functions. For example, replace this:

```
int w=win("Calculator" "CalcFrame")
lef 1 2 id(131 w)
paste "text1"
lef 3 4 id(132 w)
paste "text2"
lef 5 6 id(133 w)
paste "text3"
```

to this:

```
int w=win("Calculator" "CalcFrame")
Function225 1 2 id(131 w) "text1"
Function225 3 4 id(132 w) "text2"
Function225 5 6 id(133 w) "text3"
```

and create function Function225 (menu File -> New):

```
/
function x y hwnd str's
spe -1
lef x y hwnd
paste s
```

There are 3 similar 2-line code blocks. With 10 blocks * 10 lines it would be 100 lines. With function - 15 lines.

See also: [function tips \(150\)](#), [about functions \(149\)](#), [declaration \(paramters etc\) \(152\)](#)

Use arrays

When you need several related variables of same type, you can use single array variable.

For example, you can replace this:

```
str s0 s1 s2 s3
str s4 s5
s0="zero"
s1="one"
...
s5="five"
```

to this:

```
ARRAY(str) a.create(6)
a[0]="zero"
a[1]="one"
...
a[5]="five"
```

Arrays also allow to make something easier and smaller. Example:

```
ARRAY(str) a="zero[]one[]...[]five"
int i
for i 0 a.len
_out a[i]
```

See also: [ARRAY \(146\)](#)

Use user-defined types

When you need several related variables of different types, you can use single variable of a user-defined type.

Replace this:

```
int i1 i2
str s
i1=1
i2=2
s="text"
```

to this:

```
type MACRO_X_VAR i1 i2 str's
MACRO_X_VAR v
v.i1=1
v.i2=2
```

```
v.s="text"
```

Here MACRO_X_VAR is some unique name.

See also: how to [declare \(154\)](#) and [use \(155\)](#) user-defined types.

Remove declarations of unused variables

To find unused variables, check menu Run -> Compile Options -> [Show unused variables \(8\)](#) and compile the macro.

MSDN Library (Windows API function reference)

In QM can be used Windows API functions, interfaces, types and constants. They are documented in [MSDN Library](#) on the Internet.

See also: [F1 help and the tips pane \(245\)](#), [declarations \(242\)](#)

QM identifier names (naming conventions)

Names of identifiers (QM keywords, variables, functions, types, etc) are case sensitive and can contain only alphanumeric characters (a to z, A to Z, 0 to 9) and underscore (_). First character cannot be digit.

Identifier name must be unique in its scope:

Name of a local variable must be unique in that function or macro. If a local variable has same name as user-defined global identifier, is used local variable. Different functions or macros can have local variables with same name.

Global identifier names must be unique in that [QM file \(17\)](#). All identifiers, except local and thread variables, are global.

It is not error to declare a global identifier more than once, if declarations are identical.

When QM searches for an identifier, it does it in the following sequence: at first searches for QM-defined identifiers (QM intrinsic functions, predefined types, constants and variables), then for local and thread variables, members (in a member function only), application variables, user-defined global identifiers, and finally in type libraries and reference files.

Network setup

Quick Macros must be installed on each computer where it will be used, unless you use [portable QM \(259\)](#).

Sharing and deploying QM files

If you want to use same [QM file \(17\)](#) (thereafter *file F*) on multiple computers or user accounts simultaneously:

- Note that in QM 2.4.0 has been changed file format and the way QM opens files.
- While a file is open in QM, other QM instances (computers/accounts) cannot open it, unless all open it as read-only.
- Each computer/user must have its own main file. Add file F to the main file as a shared file (menu File -> Import). It will create a virtual folder with S letter on folder icon, where items from file F will be loaded.
- Normally only single computer (thereafter *computer A*) should open file F as editable. To prevent other computers opening it as editable (and making unavailable for computer A), on that computers in QM right click the S folder and check Folder Properties -> Shared file -> Load local read-only copy.
- When you make changes in file F on computer A, on other computers QM will load file F (or its copy) when starting or reloading main file. Note that QM does not write changes to .qml files immediately; read more in the [QM files \(17\)](#) topic.

When QM setup program runs, if it finds file Main.qml in its folder, it installs the file as default main file for that computer user. You can use this feature to automate installing same modified main QM file on multiple computers. For example, will not need to add the shared file F on multiple computers; you can do it on single computer, if the path will be valid on all computers.

The "Load local read-only copy" feature also is useful in cases when the network file F is unavailable when QM starts. Then QM loads previously used local copy of the file.

Deploying QM registry settings

QM saves most its settings in registry for each user. To export to file qmreg5.reg, click Options -> General -> Export settings.

When QM setup program runs, if finds file qmreg5.reg in its folder, installs the file in QM folder. Then, when QM runs first time on a user account on that computer, it imports QM settings from qmreg5.reg to the registry of that user. It then places "reg loaded"=1 value in the "HKCU\Software\GinDi\QM2\settings" registry key to prevent importing again. Note: This requires administrator privileges; On Windows Vista/7/8/10, a consent dialog may be shown.

QM 2.4.0. Custom toolbar position/style settings now are saved in main file, not in registry.

Note: The exported reg file includes the path of the current main QM file. You may have to open the reg file in Notepad and edit QM file path ("file"), as well as other file paths, if they are different on other computers. In file paths, you can use special folder strings, for example \$qm\$ (QM program folder) and \$my qm\$ (My Documents\My QM). Usually don't need editing because QM saves paths with special folder strings.

Deploying a macro

To send a macro to other computer, use function [NetSendMacro](#). To execute - function [net](#).

Read more in [net \(101\)](#) topic.

Registration

Enter QM registration code (when available) on each computer. You must have a [license \(252\)](#) for as many computers as actually use QM (unless only you use QM on several your computers). On multi-user computers, if you are logged on as an administrator, this automatically registers QM for all user accounts. You don't have to purchase multiple licenses for a multi-user computer.

Upgrading

To upgrade QM, install new QM version on each computer. It is free, don't need a new license.

There is a sample [macro \(101\)](#) that automates upgrading QM. To use the macro, QM on each computer must be running and configured to run macros from other computers.

See also: [setup command line parameters \(262\)](#), [license \(252\)](#).

Portable QM

You can install portable QM in a USB drive and run it on other computers without installing there. Portable QM will not require administrator rights, and will not leave its files or settings there. Supports PortableApps.com, but it is not required. You can read more about portable apps in Wikipedia.

Installing

At first QM must be installed normally on your computer. To install or upgrade portable QM in a removable drive, in QM window select menu Tools -> Portable QM. It shows 'Portable QM Setup' dialog. Options:

Install in this folder in USB drive - portable QM root folder. QM will create the folder and copy files and settings there. If using PortableApps.com, it should be X:\PortableApps\QuickMacrosPortable, where X is USB drive letter.

Replace portable file - replace portable qml file (macros) with currently loaded non-portable file.

Replace portable settings - replace portable QM settings with current non-portable QM settings. It exports QM registry key and saves in file Data\settings.xml.

Replace everything - delete the specified folder before reinstalling portable QM there. If unchecked, replaces only QM program files, other QM data, optionally macros and settings, and does not touch other files in portable QM folders.

When installing portable QM, the non-portable QM should be registered on the computer. Then portable QM will use the same registration info. If unregistered, portable QM will run with most features disabled, like the evaluation period is expired.

Your portable \$my qm\$ folder will be in the USB drive. Your portable [QM file \(17\)](#) will be \$my qm\$\Portable.qml by default.

Portable QM will create backup files in its drive, which may be undesirable. You can change it in Options -> Files, either before installing or when running portable QM.

Normally portable QM uses 12-15 MB in the USB drive.

Using

To launch portable QM:

- If using PortableApps.com, you can run QM from its menu, or set to start automatically.
- Or run QuickMacrosPortable.exe. It is in the folder specified when installing portable QM.
- Don't run qm.exe directly.

In macros:

- Avoid making changes in registry keys other than the [default key \(106\)](#). When portable QM exits, it saves the default key in the USB drive (settings.xml) and deletes the key. Some functions that make changes in registry: [rset](#), [FileTypeRegister](#), [RegisterComComponent](#).
- Avoid creating and leaving files on host computer. Create temporary files in [special folder \(246\)](#) \$temp qm\$. It is on host computer, but portable QM deletes it when exits. Create other files in the USB drive, for example in [special folder \(246\)](#) \$my qm\$ or \$drive\$\Documents.
- You can use [predefined variable \(144\)](#) _portable. It is 1 in portable QM, 0 in non-portable.

Before ejecting USB drive, you should close all portable apps. QM exits automatically if its drive ejected. If portable QM does not exit properly (eg crashes), it leaves its settings and temporary files on host computer. You can run/exit it again to clean everything.

Portable QM also can run on computers where non-portable QM is installed (but not running). It uses the same registry key, and backups/restores it.

Unavailable and limited features

Portable QM cannot use features that require administrator rights to be installed or used:

1. [Vista/7/8/10 \(277\)](#) integrity levels (IL) and other UAC-related features:
 - You cannot specify IL in Options (UAC: run as). By default QM runs as User. To run as Administrator, right click QuickMacrosPortable.exe or QM icon in PortableApps.com menu. Cannot run as uiAccess.
 - QM cannot correctly run programs with different IL than QM. When need higher IL, shows UAC consent. When need lower IL, runs as QM instead. It applies to functions [run](#), [web](#), [StartProcess](#), and to macros that are set to

run in separate process with different IL.

- QM running as User cannot automate admin windows, change admin settings, etc. Read in [Vista \(277\)](#) topic.

2. Process triggers.
3. Shell menu triggers.
4. Unlock computer.

In portable QM you should not make changes in registry and file system, unless it is restored later. These features are disabled:

1. Run at Windows startup. Instead, in PortableApps.com you can set QM to run at its startup.
2. Create scheduled tasks and shortcuts.
3. And some other.

Some other features are not installed:

1. QM shortcuts in Start menu.
2. Registration of QM program paths and .qml file type.

Synchronizing files

Sometimes you may want to synchronize your [QM file \(17\)](#), \$my qm\$ folder or other files between your PC and portable drive. Currently Quick Macros does not have this feature. You can either manually copy file(s) to/from USB drive, or use a file synchronization software. Several such programs are in PortableApps.com collection. When copying .qml files, also copy .qml-wal files with the same name, if they exist in the same folder.

Uninstalling

To uninstall portable QM, delete the folder where it is installed. Together will be deleted portable \$my qm\$ folder and portable QM file in it, unless their location is changed.

QM file management functions, macro settings and resources

In macros you can call functions of special variable `_qmfile` to manage [macro resources \(261\)](#) and other data in current [QM file\(s\) \(17\)](#). Unavailable in exe.

Most functions added in QM 2.4.0, some in QM 2.4.1.

All functions throw error if failed, for example if macro or resource not found. With 'Get' functions, error if the resource or setting does not exist; you can use [err \(129\)](#). With 'Delete' and 'Enum' functions, not error when there are no matching resources or settings. The 'Add' functions add or replace.

[Sub-functions \(182\)](#) cannot have resources and settings, but can use resources and settings of parent QM item.

Functions, parameters, examples:

Functions to manage macro resources

```
ResourceAdd($macro $resName !*data nBytes)
ResourceGet($macro $resName str&data [flags])
ResourceDelete($macro $resNameWildcard)
ResourceRename($macro $oldName $newName)
ResourceEnum($macro $resNameWildcard ARRAY(str) &aName [ARRAY(str) &aData])
```

macro - QM item whose [resources \(261\)](#) are managed. Can be:

- [QM item \(19\)](#) name (full, case insensitive) or [+id \(107\)](#) (integer). Also can be QM item path or [GUID \(249\)](#) string.
- 0 - file resources (not attached to a macro).
- +-1 - QM item that calls the file management function.
- +-2 - caller of the QM item (function) that calls the file management function.
- +-3 or "" - QM item currently open in the code editor.

resName - resource name, eg "image:bird" or "cut.ico".

- With [ResourceGet](#) also can be like "resource:<macro>name" (then **macro** not used) or "resource:name".
- Cannot be empty or contain characters " ' < > and newline. Must match case.

data, nBytes ([ResourceAdd](#)) - pointer to resource data, and resource size. See example.

data ([ResourceGet](#)) - str variable that receives resource data.

flags ([ResourceGet](#)): 1 - if **macro** does not have the resource, try to get it from a caller in the function call stack. 2 - if **macro** does not have the resource, but another QM item in the same QM file has a resource with this name, show a note in QM output.

resNameWildcard ([ResourceDelete](#), [ResourceEnum](#)) - resource name. To match multiple resources, use wildcard characters as with [SQLite GLOB \(271\)](#). If "" or "*", [ResourceEnum](#) gets all.

oldName, newName ([ResourceRename](#)) - current and new resource name.

aName ([ResourceEnum](#)) - array variable that receives names of all matching resources.

aData ([ResourceEnum](#)) - optional array variable that receives data of all matching resources.

Examples

```
str s1 s2
import image resource from file
s1.getfile("$my qm$\test.bmp")
s1.encrypt(32) ;;LZO-compress
_qmfile.ResourceAdd("Macro2193" "image:test" s1 s1.len)

export image resource to file
_qmfile.ResourceGet("Macro2193" "image:test" s2)
err out "resource not found"; ret
s2.decrypt(32) ;;LZO-decompress
s2.setfile("$my qm$\test2.bmp")
```

Functions to manage macro settings

In macros you sometimes use various settings that must be saved somewhere. Usually programs and macros use registry ([rset, rget \(106\)](#)) or files for it. Alternatively you can store macro settings in current QM file. Registry settings are specific to

the computer/user. Settings stored in QM file are specific to that QM file. QM also stores some of its settings in QM file, for example toolbar positions. Note that this feature is unavailable in exe.

Macro settings are similar to macro resources. You can use it to store any settings and other data used in your macros. Unlike resources, settings of macros of all currently loaded QM files (main and shared) are stored in the main file. Macro settings are not used with temporary QM items (items in the Temp folder).

You can manage macro settings with these functions. There are no dialogs for this.

```
SettingAddB($macro $setName !*data nBytes)
SettingAddS($macro $setName $data)
SettingAddI($macro $setName data)
SettingGetB($macro $setName !*data nBytes)
$SettingGetS($macro $setName [str&data])
#SettingGetI($macro $setName)
SettingDelete($macro $setName)
```

macro - QM item whose settings are managed. Same as with the 'Resource' functions, see above.

setName - setting name. Must match case. With [SettingDelete](#) can contain wildcard characters to delete multiple.

The 'SettingAdd' functions add or replace setting **setName** of QM item **macro**. With [SettingAddB](#), **data** is pointer to data, and **nBytes** is data size. With [SettingAddS](#), **data** is a string value. With [SettingAddI](#) - an integer value.

The 'SettingGet' functions get data of setting **setName** of QM item **macro**. Function [SettingGetB](#) copies max **nBytes** of data to memory buffer **data**; error if data size is < **nBytes**. Function [SettingGetS](#) stores data in a str variable **data** and returns **data** as string. Function [SettingGetI](#) returns data as integer value.

Examples

```
str s1 s2; int i1 i2; POINT p1 p2
add or update settings
s1="string value1"
_qmfile.SettingAddS("Macro2201" "name1" s1)

i1=100
_qmfile.SettingAddI("Macro2201" "name2" i1)

p1.x=10; p1.y=20
_qmfile.SettingAddB("Macro2201" "name3" &p1 sizeof(p1))

get settings
_qmfile.SettingGetS("Macro2201" "name1" s2)
err out "setting not found"; ret
out s2
or
out _qmfile.SettingGetS("Macro2201" "name1" _s)

i2=_qmfile.SettingGetI("Macro2201" "name2")
out i2

_qmfile.SettingGetB("Macro2201" "name3" &p2 sizeof(p2))
out F"{p2.x} {p2.y}"
```

Functions to manage QM file directly with SQLite class

QM files are SQLite databases and can be managed with [SQLite](#) class. For example, you can create and use your own tables. You also can get data from QM tables, but should not modify them, especially items and texts. QM currently uses these tables: items, texts, resources, xFind, xSett, xTags. Use these functions to access currently loaded QM files (main or shared). For other files instead use [SQLite.Open](#).

```
!*SQLiteBegin([iid])
SQLiteEnd()
#SQLiteItemProp($macro [int&rowid] [GUID&guid])
```

[SQLiteBegin](#) gives you access to the SQLite database of the currently loaded main QM file. Note that there are separate

databases for each currently loaded file (the main file and shared files). You can use **iid** (QM item id) to access the file in which is the QM item.

SqliteEnd must be called when the macro finishes working with the database. Between calls to **SqliteBegin** and **SqliteEnd** the database is locked, and QM (or another macro) waits when it wants to access the database (read or write). If **SqliteEnd** not called, the file is locked until the thread ends.

SqliteItemProp gets database-specific QM item properties that you cannot get with **qitem** (107) and **str.getmacro** (219). In databases, QM items can be identified by row id or by **GUID** (249). Row id is unique in that file; it's the id column used in some tables; it's not the same as QM item id. GUID is globally unique; it's the guid column used in some tables. The **macro** parameter can be QM item name, id, etc, like with other functions. The function returns QM item id.

Examples

```
get reference to database in which is macro Macro2202
Sqlite& x=_qmfile.SqliteBegin(qmitem("Macro2202")) ;;note: Sqlite& x, not Sqlite x

now can use Sqlite functions, for example:
str sql=
CREATE TABLE IF NOT EXISTS myTable(a INT, b TEXT);
INSERT INTO myTable VALUES(1, 'one'), (2, 'two');
x.Exec(sql)
SqliteStatement p.Prepare(x "SELECT b FROM myTable")
rep
if(!p.FetchRow) break
out p.GetText(0)

SqliteItemProp example
int iid rowid; GUID guid
iid=_qmfile.SqliteItemProp("Macro2202" rowid guid)
out "%i %i %s" iid rowid _s.FromGUID(guid)

unlock database
_qmfile.SqliteEnd
```

Other functions

```
FullSave()
```

This function does the same as the Save button in QM window - applies and saves all changes. Also saves other QM settings - registry settings, toolbar positions etc. Also checkpoints currently loaded **QM files** (17), ie transfers changes collected in temporary .qml-wal files to .qml files. You can call this function for example before a custom backup. QM internally does the same when closing files, hiding QM window and in some other cases.

```
GetLoadedFiles(ARRAY(str) &aFiles)
```

Gets paths of currently loaded **QM files** (17).

aFiles - variable that receives path strings. The first element is the main file. Other elements - shared files. Memory databases (Temp, Deleted) are not included.

See also: **FullSave** (above).

Example

```
ARRAY(str) a
_qmfile.GetLoadedFiles(a)
out a[0] ;;main file path
```

Resources (macro resources)

In macros you sometimes use images or other binary (non-text) data. There are several ways to store such data:

- External files. For example, .ico files for icons.
- Convert to text (hex or base64) and store in macro.
- Macro resources. Added in QM 2.4.1, partially in QM 2.4.0.

Each macro (or other [QM item \(19\)](#)) can have any number of resources. They are stored in the same [QM file \(17\)](#). They are deleted when you delete the macro; copied when you clone or export the macro. You can think about macro resources like about email message attachments.

Resources have names. Case-sensitive. Some names are recognized by QM as images:

- Names that begin with "image:" are LZO-compressed bitmap (.bmp file) images.
- Names that end with ".bmp", ".png", ".jpg", ".gif", ".ico", ".cur" and ".ani" are images, icons and cursors. The data format is the same as in files.

In macros use resource name with "resource:" prefix. Example: `"resource:email.ico"`.

- The prefix is optional if name begins with "image:". Example: `"image:button1"`.
- If the resource is in another macro, use prefix "resource:<macro name>". Examples: `"resource:<macro1>image:button1"`, `"resource:<icons>email.ico"`. In some cases need this even if in same macro. For macro name also can be used [GUID \(249\)](#).
- If macro name not specified, in most cases QM also can find the resource in a caller function in the function call stack.
- Resources that are not attached to a macro are called *file resources*. To manage file resources, select <file> in the Resources dialog. In macros use prefix "resource:<>". Example: `"resource:<>email.ico"`. Don't use file resources in [shared files \(17\)](#), because QM looks for the resource only in the current main file.

Functions that support resources:

- [scan](#) - "image:", ".bmp", ".png" and ".ico" resources.
- [GetFileIcon](#) - ".ico", ".cur", ".ani".
- [LoadPictureFile](#) - "image:", ".bmp", ".png", ".jpg", ".gif".
- [__ImageListLoad](#) - "image:", ".bmp".
- [ShowDialog](#) (images, background, icon) - all image resources.
- [str.getfile \(217\)](#) - all resources. Also [IXml.FromFile](#), [ICsv.FromFile](#), [RichEditLoad](#) and some other.
- [bee](#) - ".wav".
- Functions that use the above functions. For example [AddTrayIcon](#), [__ImageList.Load](#).
- [#exe addfile \(179\)](#).
- Icons in menus and toolbars - ".ico".
- QM item icon, exe icon - ".ico".

In the code editor QM displays images below lines with image resource name and file path strings. Even if the strings are in comments, if enclosed in ". To show/hide images, use toolbar button 'Images in code editor'. You also can display images in [output and function help \(245\)](#).

In Options you can set to record screenshots when recording mouse clicks or using the Mouse dialog. QM saves the recorded images in macro resources with names that begin with "~:", and inserts the names in macro text as comments. Auto-deletes unused screenshot resources, for example when you delete macro text containing "~:..." and then close the macro.

When creating [exe \(51\)](#), QM looks for strings that begin with "resource:" or "image:" and adds the macro resources to exe resources (it's a different thing, although similar). Then in exe QM functions get data from these exe resources. Supports multiline strings. Use [#exe addfile \(179\)](#) if QM does not auto-add a resource, eg if its full name string is not used in macro.

Exe resource type and name (or id) of auto-added macro resources depends on macro resource name. If it contains colon, eg "A:B", in exe A will be resource type, and B name. Else exe resource name will be the same as macro resource name, and type depends on macro resource name: if it ends with ".bmp", ".ico", ".cur" or ".ani", type will be RT_BITMAP, RT_GROUP_ICON, RT_GROUP_CURSOR or RT_ANICURSOR, else RT_RCDATA. If type or name string begins with an integer number 1-65535, in exe resources it will be numeric (id), else string. For example, if macro resource name is "10 info.ico", in exe it will be RT_GROUP_ICON resource with id 10. Note that if exe has icon resources with string names, Windows Explorer displays wrong icon; to avoid it, all icon resources added to exe should have numeric names.

[Sub-functions \(182\)](#) cannot have resources, but can use resources of parent QM item.

To manage macro resources, you can use:

- Dialog 'Resources'.
- Dialog 'Find Image' (capture image).
- [QM file management functions \(260\)](#).

The 'Resources' dialog

You can find the dialog in the Tools menu or toolbar.

At the left are listed all QM items (macros etc) that have resources. At the right are displayed resources of the selected item.

Special items at the left:

- <this> - the macro that is currently open in the code editor. When you open another macro, the dialog is updated to show its resources. If the macro does not have resources, there will be only an Add link.
- <all> - resources of all macros in single view.
- <duplicate> - resources that have identical data but are in different macros or have different names.
- <unused> - resources whose names are not found in text (or as icon) of any macro.
- <size> - resources of all macros sorted by size.
- <file> - resources that are not attached to any QM item (file resources).

At the right side you can manage resources of the selected item:

- Click the Add link to add a resource from a file or another macro. Or drag and drop files from Windows Explorer.
- Click a resource name link to delete, rename, export, etc.
- When exporting an "image:..." resource, QM uncompresses it and saves as .bmp file. When importing a .bmp file, if the 'Compress' checkbox is checked, imports as "image:..." resource. Other files/resources are imported/exported unchanged.
- When importing a .ico file, the suggested resource name begins with a number to be used for exe resource id.
- To move or copy one or more resources from macro A to macro B:
 - Open resources of macro A (or <all> etc). Optionally select text with the resource names (links).
 - Drag the line or selected text with the right mouse button, and drop to macro B in the list. To copy, use Ctrl.
 - Or copy selected text to the clipboard, open resources of macro B, click the Add link.
 - Or click a resource name and select 'Move...', then click Add in B resources.
 - If need several selections, at first make one as usually, then make more selections with Ctrl.
- To open a macro from the dialog, click its name (link) in the right. Or double click at the left, or Enter.
- If you want to see only resources of some types or containing certain string in name, use the Filter field. Will be displayed only resources with names matching the filter. Use wildcard characters as with [SQLite GLOB \(271\)](#). Supports operator NOT at the beginning.
- To find a resource by name in all resources, select <all> and enter resource name in the Filter field. Use wildcard characters to find partial, eg *partOfName*. Must match case.

Setup command line parameters

The Setup program accepts optional command line parameters. These can be useful to system administrators, and to other programs calling the Setup program.

`/SP-`

Disables the This will install... Do you wish to continue? prompt at the beginning of Setup.

`/SILENT, /VERYSILENT`

Instructs Setup to be silent or very silent. When Setup is silent the wizard and the background window are not displayed but the installation progress window is. When a setup is very silent this installation progress window is not displayed. Everything else is normal so for example error messages during installation are displayed and the startup prompt is (if you haven't disabled it with the `'/SP-'` command line option explained above).

If a restart is necessary and the `'/NORESTART'` command isn't used (see below) and Setup is silent, it will display a Reboot now? message box. If it's very silent it will reboot without asking.

`/LOG`

Causes Setup to create a log file in the user's TEMP directory detailing file installation actions taken during the installation process. This can be a helpful debugging aid. For example, if you suspect a file isn't being replaced when you believe it should be (or vice versa), the log file will tell you if the file was really skipped, and why.

`/NOCANCEL`

Prevents the user from cancelling during the installation process, by disabling the Cancel button and ignoring clicks on the close button. Useful along with `'/SILENT'` or `'/VERYSILENT'`.

`/NORESTART`

Instructs Setup not to reboot even if it's necessary. Rebooting may be necessary only when reinstalling.

`/RESTARTEXITCODE=exit code`

Specifies the custom exit code that Setup is to return when a restart is needed. Useful along with `'/NORESTART'`.

`/LOADINF="filename"`

Instructs Setup to load the settings from the specified file after having checked the command line. This file can be prepared using the `'/SAVEINF='` command as explained below. Don't forget to use quotes if the filename contains spaces.

`/SAVEINF="filename"`

Instructs Setup to save installation settings to the specified file. Don't forget to use quotes if the filename contains spaces.

`/DIR="x:\dirname"`

Overrides the default directory name displayed on the Select Destination Location wizard page. A fully qualified pathname must be specified.

`/GROUP="folder name"`

Overrides the default folder name displayed on the Select Start Menu Folder wizard page.

`/NOICONS`

Instructs Setup to initially check the Don't create any icons check box on the Select Start Menu Folder wizard page.

Delimiters

Functions [findw \(189\)](#), [findt \(190\)](#), [tok \(192\)](#), [str.gett \(223\)](#) and [str.findreplace \(211\)](#) have a **delim** parameter.

delim can be:

- A string consisting of delimiter characters.
 - For example, if **delim** is " ,", delimiters will be space and comma.
 - Note that **delim** is not a delimiter string. It is a set of possible delimiter characters.
 - QM 2.3.3. All these functions also have a flag to use blanks as additional delimiter characters.
- If **delim** is omitted or "" or 0, all non-word characters will be delimiters.
- If **delim** is 1, delimiters will be blanks, double quotes and parentheses.
- QM 2.3.3. If **delim** is 2, delimiters will be blanks.

With these functions, blanks are: space, new line, tab, all characters with codes 0-31.

With these functions, word characters are:

- Alphanumeric [ASCII \(239\)](#) characters (A-Z a-z 0-9).
- Underscore (_).
- In Unicode mode - all non-ASCII characters.
- In non-Unicode mode - other ANSI alpha characters.

In [Unicode \(267\)](#) mode, can be used only ASCII characters. Non ASCII characters are not delimiters, even if explicitly specified.

Table of delimiters

delim also can be a table of delimiters. Use flag 0x100. The table must be a 256-byte array (usually str variable). Each byte is for a character whose [code \(239\)](#) is equal to the array indice. For delimiter characters, set the bytes to 1, for others 0. For example, if `delim[32]` is 1, then space (code 32) is delimiter. Always set `delim[0]` to 1. In Unicode mode, don't use non ASCII characters as delimiters.

In the example, we create table of delimiters and use it with [tok](#) function. Delimiters will be all blanks (characters with values < 33), ", < and >.

```
str d.all(256 2 0)
d.set(1 0 33)
d[34]=1; d['<']=1; d['>']=1

str s="one.one 'two two' <three + three>"
lpstr s1 s2 s3
int n=tok(s &s1 3 d 1|4|64|256)
Now s1 is "one.one",
s2 is "two two",
s3 is "three + three"
```

Trigger coding

To assign a trigger, you can either use the Properties dialog, or write encoded trigger string directly in the "Trigger" field on the toolbar. Below is described the format of encoded trigger strings for various types of triggers.

Hotkey (30)

Syntax:

```
[mod]key[nextkey] [flags]
```

To specify trigger keys, are used [QM key codes \(251\)](#), as with the [key \(56\)](#) command. The trigger string begins with optional modifier keys (C, S, A, W). Then follow one or two keys. Here you can also use [virtual-key code \(270\)](#) in parentheses (must be number, not named constant). May be followed by space and flags (undocumented). Examples:

F9	F9
CSc	Ctrl+Shift+C
Awe	Alt+W+E
q 0x5	Q, don't eat, wait until released

Mouse (31)

Syntax:

```
#[#] [mod]trigger[monitor|hittest] [flags]
```

Begins with # or ## (if double). Then follows optional modifier key codes (C, S, A). Then follows mouse trigger type: button (L, R, M, X1 or X2), or wheel direction (U or D), or screen edge number (1 to 12, clockwise, starting from top-left), or mouse movement direction (<, >, v, ^) and area number (1 - top or left, 2 - middle, 3 - bottom or right). Then optionally can follow monitor index (m and number 1 to 30, or only m for all monitors) or [hit test code \(40\)](#) (h and number 1 to 30). May be followed by space and flags (undocumented). Examples:

#SR	Shift+mouse right click
#CSD	Ctrl+Shift+mouse wheel backward
##12	mouse movement to the top of the screen left edge, twice
#A>3	Alt+mouse movements right-left in the screen bottom area
#Lh2 0x1	Left click on window caption bar, don't eat

Window (32)

Syntax:

```
!triggertype"name" ["class"] ["childname"] ["childclass"] [style] [flags]
```

Begins with !. Then follows one or two letters: ca - created & active, cv - created & visible, a - activated, i - deactivated, v - visible, n - name changed, d - destroyed. Then follow one or more properties, and optionally flags (undocumented). All strings may contain [escape sequences \(137\)](#).

QM events (33)

Begins with @. Undocumented.

Other

Begins with \$. Undocumented.

User-defined triggers (39)

Begins with ^. Then follows trigger manager function name and optionally more data that is defined by the trigger.

Scope (only for key, mouse, window and text triggers)

If trigger is active only with single or several programs, after trigger code goes space, / and comma-delimited program names. If trigger is active with all programs except single or several programs, use \. Examples:

F12 /NOTEPAD	trigger (hotkey F12) is valid only when active window belongs to NOTEPAD program
!ca"Open"	trigger (created window "Open") valid only if the window does not belong to WINWORD or

\WINWORD,EXCEL

EXCEL program.

If the macro has a [filter function \(40\)](#), after trigger and programs goes / and filter function name. If trigger is not program-specific, filter function name goes after trigger code, space and //.

Variable types: conversions, etc

	Type	Assignment: type checking, conversions, memory management, etc		When goes out of scope	In expression with operators
		At right can be	Comments		
intrinsic types	int	any, except structures (154)	from str and lpstr receives string pointer. From BSTR and VARIANT gets numeric value (converts from string if need).		int
	byte	any, except structures	the same		int
	word	any, except structures	the same		int
	long	any, except structures	the same		long
	double	any, except structures	the same		double
	lpstr	strings (lpstr, str, BSTR, VARIANT(BSTR)), byte*, 0	from str and lpstr receives pointer value. BSTR is converted to a hidden local str variable.		lpstr or unsigned int
	str	any, except structures	copies strings, converts numeric types to string	freed string	lpstr or unsigned int
OLE types	FLOAT	intrinsic and OLE types (140)	converts strings to numeric		double
	DATE	intrinsic and OLE types	the same		double
	CY	intrinsic and OLE types	the same		double
	DECIMAL	intrinsic and OLE types	the same		double
	BSTR	intrinsic and OLE types	copies strings, converts numeric types to string	freed string	double
	VARIANT	any, except structures	copies all data, internally calls AddRef, Release, QueryInterface, etc.	freed/Releases all data	double
	ARRAY	ARRAY of the same type, VARIANT, strings	copies all elements and associated data, calls required functions, etc	the same	cannot be used
	structure	structure of the same type	copies all data, calls required functions, etc	the same	cannot be used
	interface pointer	interface pointer of the same type, 0, IUnknown, IDispatch, VARIANT	Calls AddRef. Calls Release on previous interface pointer. If type differs, calls QueryInterface.	calls Release	cannot be used
pointers	pointer	pointer of the same type, 0, byte*	function parameter of reference type also is pointer		pointer or unsigned int
	byte*	pointer of any type, 0, interface pointer, str, lpstr	similar as void* in C, or Any in VB		pointer or unsigned int
	reference	the same as variable	the same as variable	does nothing	the same as variable

Columns:

1. Groups of variable types.
2. Variable type or types.
3. Types that are allowed at right side of assignment operation when at left side is variable of type from column 2. Assignment operation occurs when using operator =, passing argument to function, returning from function ([ret](#)).
4. Comments.
5. This column shows how are released associated memory when variable is destroyed (goes out of scope).
6. This column shows how variable is interpreted in expression with operators.

Notes:

1. Type checking is more strict with function arguments, and less strict with = and [ret](#).
2. When a str, BSTR, VARIANT, ARRAY or interface pointer variable is a parameter or return value of a function (except user-defined functions), QM does not copy the associated data and does not call AddRef (the function does it itself, if necessary).
3. A structure is a user-defined type other than listed in this table.

Type declaration for functions, parameters, etc

In declarations, must be specified types of function parameters, type members, etc. Usually, type name is placed before parameter/member name, without spaces between them. To separate them, use ' '. Example:

```
str'a
```

If it is pointer or reference, ' ' is not used. Examples:

```
str*a
str&a
str**a
str***a
str*&a
```

If it is ARRAY, ' ' is optional. Examples:

```
ARRAY(str) a
ARRAY(str) 'a
```

For some types, also can be used type declaration character instead of type name. Examples:

```
#a
~a
```

Type declaration characters:

Type	byte	word	int	long	double	lpstr	str	VARIANT
Character	!	@	# or none	%	^	\$	~	` (QM 2.3.3)

If type is **int**, type name can be omitted. Example:

```
a
```

The return type of a user-defined function must follow the **function** keyword. Use ' ' to separate them, unless you use a type declaration character. Examples:

```
function'int
function#
```

For functions that don't return a value, don't specify a type.

Examples (full declarations)

Type of function SetCursorPos is int; types of parameters - int:

```
dll user32 #SetCursorPos x y
```

Type of function is word pointer; types of parameters - int, lpstr, str reference, POINT pointer:

```
function'word* int'i lpstr'a str&sr POINT*pp
or
```

```
function@* i $a ~&sr POINT*pp
```

Type of rcPaint is RECT; type of r is byte array; type of other members is int:

```
type PAINTSTRUCT hdc fErase RECT'rcPaint fRestore fIncUpd !r[32]
```

Unicode, UTF-8

Unicode is used to display text in any language, including Chinese, Japanese, Korean, Arabic, Russian. It is a character set consisting of more than 100000 characters. ANSI character sets have only [256 different characters \(239\)](#).

QM supports Unicode, starting from version 2.3.0. You can use any characters in macros, triggers, window names, file names, etc.

QM can run in ANSI mode (like in previous versions) or Unicode mode. To enable Unicode mode, check the checkbox in Options. If the checkbox in Options is unchecked, QM runs in ANSI mode, like in previous versions. To enable Unicode for a [dialog \(63\)](#), also check the checkbox in dialog editor Options.

QM does not use wide character strings (UTF-16). Instead, when running in Unicode mode, it uses UTF-8. It is a Unicode encoding. Characters in range 0 to 127 ([ASCII \(239\)](#) characters) are the same in both modes. They consist of 1 byte. Other characters consist of 2-4 bytes. In ANSI mode, all characters consist of 1 byte.

It is recommended to use Unicode mode. Especially if your system character set is multibyte, for example chinese simplified, because then in ANSI mode some characters also consist of more than 1 byte, and QM does not work well with it.

If you enable Unicode mode, you also have to retype all non ASCII characters in macros, triggers and everywhere in QM interface. Or run the converter, read below. It is because in Unicode mode text is interpreted as UTF-8, and these characters must be encoded using UTF-8. Since the text was entered in ANSI mode, these characters in Unicode mode are invalid. Invalid characters are displayed as hexadecimal character codes in black rectangles. You also will have to retype non ASCII character [escape sequences \(137\)](#) in strings. If something containing non ASCII characters is encrypted - you'll have to re-encrypt it. If you switch back to ANSI mode, non ASCII characters entered in Unicode mode also will become invalid because they are encoded as several bytes.

QM does not do any conversions of data when you change Unicode mode. However, there is an ANSI to Unicode converter. After switching to Unicode mode, in Options click the small arrow button beside the Unicode checkbox. It displays all items containing non-ASCII characters. These items must be converted to Unicode, unless you already did it. Click an item name in the output to open the item. Click again to convert. Note that QM cannot know whether the item is already converted. If QM thinks it might be already converted, QM displays a warning message box. Note that QM may fail to convert macros created on other computers because of different ANSI character set (Control Panel -> Regional -> Advanced or Administrative -> Language for non-Unicode programs). An Unicode to ANSI converter is not available.

After switching Unicode mode, you also may have to change something else in macros created in other mode. It is documented in help topics of the functions that behave differently in ANSI and Unicode modes. Search for "UTF-8" or "Unicode".

If you did not use non ASCII characters, after switching to/from Unicode mode everything should work well without any modifications.

Not all QM functions and other features support Unicode. Where it is not supported, or partially supported, or needs to review/edit code, it is documented in the Help.

All string functions where you can specify string length (or character offset, number of characters, etc), always interpret it as number of bytes. In Unicode mode, it is not always equal to the number of characters because non ASCII characters consist of several bytes. Some functions may have an option to specify number of UTF-8 characters instead of bytes. Be careful when working with strings that contain non ASCII characters. For example, if you split a string in a middle of a character that consists of several bytes, the string will be invalid.

You can use [this macro](#) to see what characters require more than 1 byte.

Macro

```
displays Unicode character codes, characters, and number of UTF-8 bytes when QM is running in Unicode mode
out
int i
for i 1 0x10000
  str s=UnicodeCharToString(i)
  out "%i %s [%i]" i s s.len
```

Function UnicodeCharToString

```
/
function~ ch
```

Converts Unicode character to string.

In Unicode mode the string will be UTF-8.

Otherwise it will be ANSI, and Unicode characters that cannot be translated to ANSI will be converted to ?.

```
if(ch&0xffff0000) end ES_BADARG

str s
lpstr ss+=&ch
s.ansi(ss)

ret s
```

Unicode characters with character codes >0xFFFF are rarely used. They require 4 bytes when encoded in UTF-8. In UTF-16 they require 2 words (4 bytes). Other characters (0 to 0xFFFF) in UTF-8 require 1 to 3 bytes.

When working in Unicode mode, sometimes you have ANSI text that you get from an external source, for example from a file. If the text contains non ASCII characters, you may have to convert it to UTF-8 to make compatible with QM functions. And vice versa. Use `str` function [ConvertEncoding](#) or [unicode/ansi \(238\)](#) or [LoadUnicodeFile](#). Explicit conversions are rarely needed, because QM functions that know external text format convert the text automatically. For example window/control/object names, file names, etc.

To store Unicode text, often is used UTF-16 format, where all characters consist of 2 bytes (rarely 4). For example, variables of BSTR type use UTF-16. In some cases working with such text is easier than with UTF-8, because of fixed length characters. However it is difficult to use in some programs (including QM), and in most cases requires more space. To store text of various languages without using UTF-16, are used various character sets and encodings. The UTF-8 encodes Unicode characters in a string so that the string in most cases looks like ANSI for programs that don't use UTF-16, but UTF-8-aware programs (including QM, when it runs in Unicode mode) can display all Unicode characters in the string.

Examples of characters, their values and size in ANSI, UTF-16 and UTF-8:

	ANSI	UTF-16	Number of bytes used in UTF-8
A	65	65 (2 bytes: 65 0)	1, like all characters in range 0 to 127
©	169	169 (2 bytes: 169 0)	2, like all characters in range 128 to 255
β		0x03B2 (2 bytes)	2, like all characters in range 0x100 (256) to 0x7FF
岸		0x5CB8 (2 bytes)	3, like all characters in range 0x800 to 0xFFFF

You can find more information about Unicode and UTF-8 on the Internet.

See also: [str.unicode/ansi \(238\)](#), [_unicode variable \(144\)](#), [Unicode dialogs \(63\)](#), [Unicode dll functions \(153\)](#).

Unlock computer

Some triggers (scheduler, file, event log, process) may start the macro while QM is not running in the currently active user session/desktop. For example, the computer may be locked (directly or by a secure screen saver), the user is not logged on, the user is switched off, a custom desktop is active. Then the macro runs in the background, and cannot use keyboard, mouse, and some other functions.

To ensure that the macro will run only in normal conditions, check Properties -> Macro properties -> "Don't run in background". This also will close nonsecure screensaver.

If you also check "If computer locked, unlock", QM unlocks computer if it is locked when the macro starts. Also closes screensaver. When the macro ends, QM can lock computer again. This feature is unavailable in [portable QM \(259\)](#).

Before using this feature first time, configuration is necessary. To show the configuration dialog, click the 'How to unlock computer...' button. Enter the Windows user account password. If need, change keys and other options. Whenever in the future you change Windows user account password, need to change it here too.

To unlock, QM uses keys specified in the dialog. The format is the same as with [key \(56\)](#). For example, T is Tab, Y is Enter, Au is Alt+U, (1.0) is 1 second delay, "Text" is Text typed as simple text, etc. Some [key](#) features not supported, for example VK_ and (#n). Use (USERNAME) and (PASSWORD) to type username/password specified in the "Unlock Computer - Options" dialog. Examples:

1. Windows XP, when shows dialog with user name and password fields:

Au (USERNAME) Ap (PASSWORD) Y

2. Windows XP, welcome screen, when initially shows password field:

(PASSWORD) Y

3. Windows XP, welcome screen, when initially does not show password field. Examples for user 1, 2 and 3:

TT (PASSWORD) Y

TTD (PASSWORD) Y

TTDD (PASSWORD) Y

4. Windows Vista/7. Examples for user 1, 2 and 3:

V (1.0) B (PASSWORD) Y

R V (1.0) B (PASSWORD) Y

RR V (1.0) B (PASSWORD) Y

5. Windows Vista/7, when initially shows password field:

(PASSWORD) Y

6. Windows 8/8.1:

Z (0.5) Z (1.5) (PASSWORD) Y

7. Windows 10:

(PASSWORD) Y

QM 2.3.5. Alternatively you can use a program that unlocks computer. You can create it from a macro. It can do anything - evaluate conditions, send keys, click, wait, use window and accessible object functions, etc. It should return a nonzero value if failed. [Example](#).

```
Acc a.Find(win "TEXT" "" "class=Edit" 0x1005) ;;find password field
err
key Y
2
a.Find(win "TEXT" "" "class=Edit" 0x1005)
err
LogFile "no password field" 0 "qmtul.log"
ret 1 ;;error
act child(a) ;;set focus
key "password" Y
```

note: out will not work. Use LogFile.

Notes

1. The options in the configuration dialog are common to all macros.
2. This feature has minimal impact on security. The password is saved encrypted. While temporarily unlocked, keyboard and mouse are disabled, and there is no way to access the computer. Pressing Ctrl+Alt+Delete will immediately lock the computer.
3. QM attempts to unlock computer only if it is locked (manually or by a secure screen saver). The macro will not run if not logged on, or another user is active, or another desktop is active.
4. While computer is temporarily unlocked, macros that have "don't run in background" checked are not allowed to run. They wait until computer is locked and unlocked again (if "if locked, temporarily unlock" is checked) or don't run.
5. The log file can be used to find problems. It does not have a size limit.
6. You also can use function [EnsureLoggedOn \(114\)](#).
7. Does not support [Unicode \(267\)](#). User name, password, keys and file names must be ANSI.
8. On Windows XP fails to unlock in these conditions: a) After switching to other user and then logging off. b) When locked by Remote Desktop.
9. On Windows Vista and later fails to send Ctrl+Alt+Delete if security settings modified. To fix it, run gpedit.msc, open Computer Config/Admin Templ/Win Comp/Win Logon Opt/Disable Enable SAS, and enable Ease Access Apps.
10. Currently tested on Windows XP, Vista, 7, 8, 8.1 and 10. May not work on newer Windows versions.

Use this macro to test unlocking:

```
shutdown 6 ;;lock
wait 10
int unlocked=EnsureLoggedOn(1)
out "unlocked=%i" unlocked ;;should be 2
if(!unlocked and FileExists("$qm$\qmtul.log")) run "notepad.exe" "$qm$\qmtul.log"
```

VARIANT members table

C type	QM type	Name	vt	vt constant	Comments
			0	VT_EMPTY	[V] [P] nothing
			1	VT_NULL	[V] [P] SQL style Null
short	word	(p)iVal	2	VT_I2	[V][T][P][S] 2 byte signed integer
int, long	int	(p)lVal	3	VT_I4	[V][T][P][S] 4 byte signed integer
float	FLOAT	(p)fltVal	4	VT_R4	[V][T][P][S] 4 byte real
double	double	(p)dblVal	5	VT_R8	[V][T][P][S] 8 byte real
CY	CURRENCY	(p)cyVal	6	VT_CY	[V][T][P][S] currency
DATE	DATE	(p)date	7	VT_DATE	[V][T][P][S] date/time
BSTR	BSTR	(p)bstrVal	8	VT_BSTR	[V][T][P][S] OLE Automation string
IDispatch*	IDispatch	(p)pdispVal	9	VT_DISPATCH	[V][T][P][S] interface that supports IDispatch
SCODE	int	(p)scode	10	VT_ERROR	[V][T][P][S] error code
VARIANT_BOOL	word	(p)boolVal	11	VT_BOOL	[V][T][P][S] boolean. True=-1 (0xffff in word), False=0.
VARIANT*	VARIANT*	pvarVal	12	VT_VARIANT	[V][T][P][S] In VARIANT it is pointer to other VARIANT, with VT_BYREF flag. In array not pointer.
IUnknown*	IUnknown	(p)punkVal	13	VT_UNKNOWN	[V][T] [S] interface that does not support IDispatch
DECIMAL	DECIMAL	(p)decVal	14	VT_DECIMAL	[V][T] [S] uses whole VARIANT
char	byte	(p)cVal	16	VT_I1	[V][T][P][S] 1 byte signed integer
BYTE (unsigned char)	byte	(p)bVal	17	VT_UI1	[V][T][P][S] 1 byte unsigned integer
WORD (unsigned short)	word	(p)uiVal	18	VT_UI2	[V][T][P][S] 2 byte unsigned integer
DWORD (unsigned long)	int	(p)ulVal	19	VT_UI4	[V][T][P][S] 4 byte unsigned integer
__int64	long	(p)llVal	20	VT_I8	[V][T][P][S] 8 byte signed integer
unsigned __int64	long	(p)ullVal	21	VT_UI8	[V][T][P][S] 8 byte unsigned integer
int	int	(p)intVal	22	VT_INT	[V][T][P][S] use VT_I4 instead
UINT (unsigned int)	int	(p)uintVal	23	VT_UINT	[V][T] [S] use VT_UI4 instead
void*	byte*	byref	24	VT_VOID	[T] C style void
			25	VT_HRESULT	[T] Standard return type
			26	VT_PTR	[T] pointer type
			27	VT_SAFEARRAY	[T] safe array
			28	VT_CARRAY	[T] C style array
			29	VT_USERDEFINED	[T] user-defined type
			30	VT_LPSTR	[T][P] null terminated string
			31	VT_LPWSTR	[T][P] wide null terminated string
		pvRecord	36	VT_RECORD	[V] [P][S] user-defined type. In QM used only in arrays.
			64	VT_FILETIME	[P] FILETIME
			65	VT_BLOB	[P] Length prefixed bytes
			66	VT_STREAM	[P] Name of the stream follows
			67	VT_STORAGE	[P] Name of the storage follows
			68	VT_STREAMED_OBJECT	[P] Stream contains an object
			69	VT_STORED_OBJECT	[P] Storage contains an object
			70	VT_BLOB_OBJECT	[P] Blob contains an object

269. VARIANT members table

		71	VT_CF	[P] Clipboard format
		72	VT_CLSID	[P] Class ID
		0x1000	VT_VECTOR	[P] simple counted array (flag)
	(p)parray	0x2000	VT_ARRAY	[V] SAFEARRAY* (flag)
		0x4000	VT_BYREF	[V] pointer (flag)

If SAFEARRAY* (ARRAY in QM) is stored in VARIANT, vt is element type with VT_ARRAY flag.

(p) - exists member with p prefix. It is pointer, and vt is with VT_BYREF flag.

[V] - may appear in a VARIANT.

[S] - may appear in a SAFEARRAY.

[P] - may appear in an OLE property set.

[T] - may appear in a TYPEDESC (type library).

Virtual-key codes

Constant name	Value	QM	Key or mouse button
VK_LBUTTON	0x1		Left mouse button
VK_RBUTTON	0x2		Right mouse button
VK_CANCEL	0x3		Control-break (Ctrl+Pause)
VK_MBUTTON	0x4		Middle mouse button
VK_XBUTTON1	0x5		X1 mouse button
VK_XBUTTON2	0x6		X2 mouse button
-	0x7		Undefined
VK_BACK	0x8	B	BACKSPACE key
VK_TAB	0x9	T	TAB key
-	0xA-0xB		Reserved
VK_CLEAR	0xC		CLEAR key (Shift+Num5)
VK_RETURN	0xD	Y	ENTER key
-	0xE-0xF		Undefined
VK_SHIFT	0x10	S	SHIFT key
VK_CONTROL	0x11	C	CTRL key
VK_MENU	0x12	A	ALT key
VK_PAUSE	0x13	G	PAUSE key
VK_CAPITAL	0x14	K	CAPS LOCK key
VK_KANA	0x15		IME Kana mode
VK_HANGUEL	0x15		IME Hanguel mode
VK_HANGUL	0x15		IME Hangul mode
-	0x16		Undefined
VK_JUNJA	0x17		IME Junja mode
VK_FINAL	0x18		IME final mode
VK_HANJA	0x19		IME Hanja mode
VK_KANJI	0x19		IME Kanji mode
-	0x1A		Undefined
VK_ESCAPE	0x1B	Z	ESC key
VK_CONVERT	0x1C		IME convert
VK_NONCONVERT	0x1D		IME nonconvert
VK_ACCEPT	0x1E		IME accept
VK_MODECHANGE	0x1F		IME mode change request
VK_SPACE	0x20	V	SPACEBAR
VK_PRIOR	0x21	P	PAGE UP key
VK_NEXT	0x22	Q	PAGE DOWN key
VK_END	0x23	E	END key
VK_HOME	0x24	H	HOME key
VK_LEFT	0x25	L	LEFT ARROW key
VK_UP	0x26	U	UP ARROW key
VK_RIGHT	0x27	R	RIGHT ARROW key
VK_DOWN	0x28	D	DOWN ARROW key
VK_SELECT	0x29		SELECT key
VK_PRINT	0x2A		PRINT key
VK_EXECUTE	0x2B		EXECUTE key
VK_SNAPSHOT	0x2C		PRINT SCREEN key
VK_INSERT	0x2D	I	INS key
VK_DELETE	0x2E	X	DEL key
VK_HELP	0x2F		HELP key
	0x30	0	0 key
	0x31	1	1 key
	0x32	2	2 key
	0x33	3	3 key
	0x34	4	4 key

270. Virtual-key codes

	0x35	5	5 key
	0x36	6	6 key
	0x37	7	7 key
	0x38	8	8 key
	0x39	9	9 key
-	0x3A- 0x40		Undefined
	0x41	a	A key
	0x42	b	B key
	0x43	c	C key
	0x44	d	D key
	0x45	e	E key
	0x46	f	F key
	0x47	g	G key
	0x48	h	H key
	0x49	i	I key
	0x4A	j	J key
	0x4B	k	K key
	0x4C	l	L key
	0x4D	m	M key
	0x4E	n	N key
	0x4F	o	O key
	0x50	p	P key
	0x51	q	Q key
	0x52	r	R key
	0x53	s	S key
	0x54	t	T key
	0x55	u	U key
	0x56	v	V key
	0x57	w	W key
	0x58	x	X key
	0x59	y	Y key
	0x5A	z	Z key
VK_LWIN	0x5B	W	Left Windows key
VK_RWIN	0x5C		Right Windows key
VK_APPS	0x5D	M	Applications key
-	0x5E		Reserved
VK_SLEEP	0x5F		Computer Sleep key
VK_NUMPAD0	0x60	N0	Numeric keypad 0 key
VK_NUMPAD1	0x61	N1	Numeric keypad 1 key
VK_NUMPAD2	0x62	N2	Numeric keypad 2 key
VK_NUMPAD3	0x63	N3	Numeric keypad 3 key
VK_NUMPAD4	0x64	N4	Numeric keypad 4 key
VK_NUMPAD5	0x65	N5	Numeric keypad 5 key
VK_NUMPAD6	0x66	N6	Numeric keypad 6 key
VK_NUMPAD7	0x67	N7	Numeric keypad 7 key
VK_NUMPAD8	0x68	N8	Numeric keypad 8 key
VK_NUMPAD9	0x69	N9	Numeric keypad 9 key
VK_MULTIPLY	0x6A	N*	Multiply key
VK_ADD	0x6B	N+	Add key
VK_SEPARATOR	0x6C		Separator key
VK_SUBTRACT	0x6D	N-	Subtract key
VK_DECIMAL	0x6E	N.	Decimal key
VK_DIVIDE	0x6F	N/	Divide key
VK_F1	0x70	F1	F1 key
VK_F2	0x71	F2	F2 key
VK_F3	0x72	F3	F3 key
VK_F4	0x73	F4	F4 key
VK_F5	0x74	F5	F5 key

270. Virtual-key codes

VK_F6	0x75	F6	F6 key
VK_F7	0x76	F7	F7 key
VK_F8	0x77	F8	F8 key
VK_F9	0x78	F9	F9 key
VK_F10	0x79	F10	F10 key
VK_F11	0x7A	F11	F11 key
VK_F12	0x7B	F12	F12 key
VK_F13	0x7C	F13	F13 key
VK_F14	0x7D	F14	F14 key
VK_F15	0x7E	F15	F15 key
VK_F16	0x7F	F16	F16 key
VK_F17	0x80	F17	F17 key
VK_F18	0x81	F18	F18 key
VK_F19	0x82	F19	F19 key
VK_F20	0x83	F20	F20 key
VK_F21	0x84	F21	F21 key
VK_F22	0x85	F22	F22 key
VK_F23	0x86	F23	F23 key
VK_F24	0x87	F24	F24 key
-	0x88-0x8F		Unassigned
VK_NUMLOCK	0x90	O	NUM LOCK key
VK_SCROLL	0x91	J	SCROLL LOCK key
	0x92-0x96		OEM specific
-	0x97-0x9F		Unassigned
VK_LSHIFT	0xA0		Left SHIFT key
VK_RSHIFT	0xA1		Right SHIFT key
VK_LCONTROL	0xA2		Left CONTROL key
VK_RCONTROL	0xA3		Right CONTROL key
VK_LMENU	0xA4		Left MENU key
VK_RMENU	0xA5		Right MENU key
VK_BROWSER_BACK	0xA6		Browser Back key
VK_BROWSER_FORWARD	0xA7		Browser Forward key
VK_BROWSER_REFRESH	0xA8		Browser Refresh key
VK_BROWSER_STOP	0xA9		Browser Stop key
VK_BROWSER_SEARCH	0xAA		Browser Search key
VK_BROWSER_FAVORITES	0xAB		Browser Favorites key
VK_BROWSER_HOME	0xAC		Browser Start and Home key
VK_VOLUME_MUTE	0xAD		Volume Mute key
VK_VOLUME_DOWN	0xAE		Volume Down key
VK_VOLUME_UP	0xAF		Volume Up key
VK_MEDIA_NEXT_TRACK	0xB0		Next Track key
VK_MEDIA_PREV_TRACK	0xB1		Previous Track key
VK_MEDIA_STOP	0xB2		Stop Media key
VK_MEDIA_PLAY_PAUSE	0xB3		Play/Pause Media key
VK_LAUNCH_MAIL	0xB4		Start Mail key
VK_LAUNCH_MEDIA_SELECT	0xB5		Select Media key
VK_LAUNCH_APP1	0xB6		Start Application 1 key
VK_LAUNCH_APP2	0xB7		Start Application 2 key
-	0xB8-0xB9		Reserved
VK_OEM_1	0xBA	:	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the ';' key
VK_OEM_PLUS	0xBB	+ or =	For any country/region, the '+' key
VK_OEM_COMMA	0xBC	, or <	For any country/region, the ',' key
VK_OEM_MINUS	0xBD	- or	For any country/region, the '-' key

270. Virtual-key codes

VK_OEM_PERIOD	0xBE	– . >	For any country/region, the '.' key
VK_OEM_2	0xBF	/ or ?	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '/' key
VK_OEM_3	0xC0	` or ~	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '`~' key
-	0xC1-0xD7		Reserved
-	0xD8-0xDA		Unassigned
VK_OEM_4	0xDB	[Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '[' key
VK_OEM_5	0xDC	\ or	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the '\ ' key
VK_OEM_6	0xDD]	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the ']' key
VK_OEM_7	0xDE	'	Used for miscellaneous characters; it can vary by keyboard. For the US standard keyboard, the 'single-quote/double-quote' key
VK_OEM_8	0xDF		Used for miscellaneous characters; it can vary by keyboard.
-	0xE0		Reserved
	0xE1		OEM specific
VK_OEM_102	0xE2		Either the angle bracket key or the backslash key on the RT 102-key keyboard
	0xE3-0xE4		OEM specific
VK_PROCESSKEY	0xE5		IME PROCESS key
	0xE6		OEM specific
VK_PACKET	0xE7		Used to pass Unicode characters as if they were keystrokes.
-	0xE8		Unassigned
	0xE9-0xF5		OEM specific
VK_ATTN	0xF6		Attn key
VK_CRSEL	0xF7		CrSel key
VK_EXSEL	0xF8		ExSel key
VK_EREOF	0xF9		Erase EOF key
VK_PLAY	0xFA		Play key
VK_ZOOM	0xFB		Zoom key
VK_NONAME	0xFC		Reserved for future use
VK_PA1	0xFD		PA1 key
VK_OEM_CLEAR	0xFE		Clear key

Columns:

1. Constant name. Can be used with [key](#), like `key (VK_TAB)`.
2. Value, in hexadecimal format. Can be used with [key](#), like `key (0x9)`.
3. QM key code. Can be used with [key](#), like `key T`.

The constants and values also can be used with Windows API functions.

Virtual-key codes of alphanumeric keys match character codes of corresponding uppercase characters. For example, `'A'` is virtual-key code of key A.

See also: [key \(56\)](#), [QmKeyCodeFromVK \(114\)](#), [QmKeyCodeToVK \(114\)](#)

Wildcard characters

Wildcard characters can be used in some functions that compare strings, find files, windows, etc.

*	matches zero or more characters. For example, "c:\folder*.txt" matches all text files in "c:\folder" folder.
?	matches any single character.

With most file functions that support it, wildcard characters can be used only in filename part. In Include/Exclude fields in menu properties, wildcards can be used in any place of file path.

See also: [matchw \(196\)](#)

SQLite GLOB

With [Sqlite](#) functions, in SQL you can use GLOB operator. It supports special characters *, ? and more.

*	Matches any sequence of zero or more characters.
?	Matches exactly one character.
[...]	Matches one character from the enclosed list of characters.
[^...]	Matches one character not in the enclosed list.

With the [...] and [^...] matching, a range of characters can be specified using -. Example: "[a-z]" matches any single lower-case letter. To match character -, make it the last character in the list. To match character], make it the first after [or ^. To match * or ?, put them in [].

What's new in versions 2.1.0 - 2.3.6

QM 2.3.6 (August 26, 2013)

Exact version: 2.3.6.5.

Main new features

1. Supports DPI-scaled windows.
2. Bug fixes.

All new features

1. Supports [DPI-scaled windows \(243\)](#).
2. New functions:
 - [Htm.Hwnd](#).
3. New in:
 - [mov \(75\)](#): option + (move and resize).
 - [end \(131\)](#): flag 32 (temporary).
4. Bug fixes:
 - QM 2.3.5 bug: sometimes QM crashes when collapsing a folder.
 - QM 2.3.5 bug in Dialog Editor: when adding control, sometimes moves other control.
 - Sometimes incorrectly records mouse movements.
 - When ending debug mode, sometimes executes some more code.
 - Autotext triggers: sometimes does not work if not using low level hooks.
 - Sometimes QM or some running thread stops working after creating exe.
 - In QM 2.3.6.3-5 fixed bugs in Sqlite.Open. Added more declarations in ref `__sqlite`. Fixed bugs in demo macros.
 - And more.
5. **What can be incompatible with previous versions:**
 - In [DPI-scaled windows \(243\)](#) old QM would record mouse clicks with incorrect coordinates. In some cases the macro would work anyway, but now it will not work, need to record again. Same with all mouse and window functions that use coordinates (x, y, width, height) in DPI-scaled windows.

QM 2.3.5 (July 21, 2013)

Exact version: 2.3.5.12.

Main new features

1. Can run as portable app.
2. Can compile and execute C# and VB.NET code.
3. Several new functions. Improvements in some functions and other features.
4. Bug fixes.

All new features

1. Can be installed in a USB drive and run as [portable \(259\)](#) app.
2. Class [CsScript](#), functions [CsExec](#), [CsFunc](#), [VbExec](#), [VbFunc](#). Compiles and executes C# or VB.NET code.
3. Other new functions and classes:
 - [PerfFirst, PerfNet, PerfOut \(114\)](#). Shows time spent executing code.
 - [CompareFilesInFolders](#), [InputBox](#).
 - [More](#).

- [DT_SetAccelerators](#) (dialog hotkeys), [TriggerInfoAutotext](#), [MessageLoopOptions](#), [GetNetRuntimeFolder](#).
- [__ComActivator](#) and [__ComActivator_CreateManifest](#). Use COM and .NET COM components without registry.
- [__SharedMemory](#). Memory that can be used by multiple processes.
- [GetCallStack \(114\)](#), [IsValidCallback \(114\)](#).

4. New in:

- [GetFilesInFolder](#) can get file size or time; easy to sort.
- [opt \(97\)](#): nowarningshere, noerrorshere. In 2.3.5.9 fixed noerrorshere bug: [err](#) does not work in the function.
- [end \(131\)](#): flag 16 (append last dll error string).
- [tok \(192\)](#): Can trim blanks, etc.
- [str.ucase](#), [str.lcase \(226\)](#): can convert part of string.
- New [predefined variable \(144\)](#) [_portable](#).
- New [special folders \(246\)](#) [\\$drive\\$](#) and [\\$temp qm\\$](#).
- [str.setfile \(217\)](#) and most other QM file-write functions: auto-creates parent folder.
- [IXml, IXmlNode \(117\)](#): More features in [Path](#) and some other functions.
- Make exe: Adds text of macros from string constants like "macro:MacroName". [Read more \(179\)](#).
- And more. To find all, search for string 2.3.5 in this Help file and in QM.

5. [Autotext \(29\)](#) (TS menu): Supports items that begin with a delimiter character.6. [Shell menu triggers \(38\)](#): Improvements and optimizations.7. [File saving and backup \(15\)](#): Improvements and optimizations.

8. Recording: Now also records mouse wheel and X1/X2 buttons.

9. [Unlock computer \(268\)](#): can run a program.10. Some [changes](#) in VARIANT and ARRAY.

- Fixed bugs and removed some limitations in assignment VARIANT=ARRAY.
- Converts array type in assignments ARRAY(BSTR)=ARRAY(str), VARIANT=ARRAY(str) and vice versa.
- VARIANT better supports long (64-bit integer) type. Added member lVal etc. In assignment VARIANT=long, stores the value as VT_I8 (previously VT_DECIMAL). And more.
- In assignment VARIANT=&Function, stores the value as VT_I4 (previously VT_I4|VT_BYREF).

11. [Multiline strings \(138\)](#) in the code editor: String color. F-string variables also colored correctly.

12. Some more tooltips and polishing in Options, Find and other dialogs.

13. Bug fixes:

- Memory leaks.
- Mouse double movement triggers don't work.
- Keyboard triggers: with some macros, after Alt+key or similar trigger, for several seconds behaves like if Alt is pressed.
- Often fails to find accessible object in Firefox first time if without waiting.
- And more.

14. **What can be incompatible with previous versions:**

- If you have functions with same names as the new functions, will need to rename or delete them.

QM 2.3.4 (February 2, 2013)

Exact version: 2.3.4.8.

Main new features

1. Window text capturing.
2. Improvements in many functions and other features.

3. Bug fixes.

All new features

1. **WindowText** class. Captures window text, clicks an item, etc. Look in floating toolbar -> Windows, controls.
2. New functions:
 - [ChangeFileSecurity](#), [CaptureWindowAndRect](#), [WshExec](#), [RtOptions \(114\)](#), [QmSetWindowClassFlags \(114\)](#).
3. New in:
 - Dialogs: easy tooltips. Look in Dialog Editor.
 - [F strings \(138\)](#): Coloring etc, like in other code. Supports strings in variable fields, enclosed in `.
 - [win \(77\)](#), [child \(83\)](#): New syntax with CSV string instead of x y. Can find window containing specified child window. [child](#) supports Windows Forms (.NET) control name, accessible name and more.
 - [but](#): Supports Windows Forms (.NET) controls better.
 - [wait \(89\)](#) WT: wait for specified window name.
 - [run \(64\)](#): added flag to run 64-bit system programs.
 - [bee](#): Supports "resld file.wav" syntax that allows to easily add the wav file to [exe \(51\)](#).
 - [#exe addfile \(179\)](#): supports any resource type.
 - Make exe: can add version info.
 - [IStringMap](#), [ICsv](#), [IXml](#): Supports [_create](#). Added member functions to set flags.
 - [ICsv \(116\)](#): New functions Find, AddRow2, FromArray, ToArray.
 - [Database](#): [CsAccess](#) and [CsExcel](#) support file formats used by Office 2007+.
 - Supports png images in dialogs etc.
 - File triggers: Can watch only files, only folders, or all.
 - Window class name in all QM functions and triggers can be with wildcard characters. Previously would need a flag.
 - New QM output [tags \(245\)](#): mes, out.
 - More options in some other functions.
4. It is possible to [use COM components without registration \(163\)](#).
 - Exe: don't need to register QM COM components on other computers. These are mailbee.dll (used by QM email functions) and arservicesmgr.dll (used with Services functions). Need just these dlls in exe folder.
5. Optimizations, changes, etc:
 - Better Windows 8 compatibility.
 - Removed qm64 process on 64-bit Windows.
 - Simplified floating toolbar dialogs. Added tooltips.
 - Recording: Removed some options. Added 'Slow' option. Generates more comments. And more.
 - Removed "QM always on top" view. Instead you can use floating output and status bar windows.
 - Most global run-time options moved from Options dialog to function [RtOptions \(114\)](#).
 - Make exe: Removed 'Compress' option, because of possible problems with antivirus software.
 - Macros that [run in separate process \(51\)](#) now by default use a small .qmm file instead of .exe file.
 - QM enables [Chrome accessible objects \(84\)](#).
6. Bug fixes:
 - F strings, when used as a function argument: error or incorrect result if [digits] used in a variable. Example: `act F"{a[0]}"`.
 - Operator [or \(133\)](#): If first operand nonzero, returns its value (must be 1). It's not a problem in most cases, because the value itself is rarely used. Another bug - does not work if first operand is double. Added warning for yet another bug that cannot be fixed.
 - [DaysInMonth](#): no leap year correction at 100/400 years. Also in [DateTime](#) functions [AddMonths](#) and [AddYears](#).
 - [TimeSpanFromStr](#): may incorrectly parse string if without days.
 - Process triggers: if 'Already running' selected, the macro runs before initializing QM extensions and user startup functions.
 - Keyboard triggers: don't work repeated triggers with modifier keys.
 - And more.
7. What can be incompatible with previous versions:

- If you have functions with same names as the new functions, will need to rename or delete them.
- Removed function [share \(111\)](#).
- If you use the 'Favorite dialogs' feature (in floating toolbar), may need to rebuild your favorites, because there are changes in dialogs.

Below are some changes since QM 2.3.4.3. Also included above.

QM 2.3.4.4-8.

1. [run \(64\)](#): added flag to run 64-bit system programs.
2. Macros that [run in separate process \(51\)](#) now by default use a small .qmm file instead of .exe file.
3. Supports png images in dialogs etc.
4. QM enables [Chrome accessible objects \(84\)](#).
5. QM 2.3.4.7 fixed QM 2.3.4.6 bug: QM always runs in Unicode mode, even if you uncheck it.
6. QM 2.3.4.8 fixed some small bugs.

QM 2.3.3 (March 10, 2012)

Exact version: 2.3.3.7

Main new features

1. New UI object ([Acc](#) etc) functions. Other new or improved functions.
2. New or improved information features: annotations, errors, function info/help, coloring, [key](#) comments.
3. New [autotext list \(29\)](#) (TS menu) features.
4. Bug fixes.

All new features

1. New in UI objects:
 - [acc \(85\)](#) flags: 0x2000 - search only in web page (faster). 0x4000 - use more properties, including HTML attributes.
 - New [Acc](#) functions: [Find](#) (replaces [acc](#)), [FindFF](#) (faster, Firefox/Chrome), [WebScrollTo](#), [WebAttribute](#), [CbSelect](#) and more.
 - New class [FFNode](#). Firefox/Chrome HTML nodes.
 - New [Htm](#) functions: [FromAcc](#), [FromXY](#), [DocProp](#).
2. New functions and classes:
 - Class [__Settings](#). Manages settings that you use in a macro.
 - New [str](#) functions: [UniqueFileName](#), [LimitLen](#), [GetClipboardHTML](#).
 - New [HtmlDoc](#) functions: [SetOptions](#), [Delete](#).
 - New [ExcelSheet](#) functions/options.
 - [ExeFullPath](#), [ExeParseCommandLine](#), [ExeQmGridDll](#), [MakeExeCloseRunning](#), [DT_SetTextColor](#), [DT_SetBackgroundColor](#), [DT_SetBackgroundImage](#), [OutStatusBar](#), [TriggerInfoTsMenu](#), [EndThread](#).
 - [paste \(58\)](#). Alias for [outp](#).
3. New options and improvements in:
 - [key \(56\)](#): can repeat a key. Text ([key](#) "text") does not depend on state of Caps Lock, Shift, Ctrl, Alt, Win.
 - [ifk \(59\)](#), [iff \(70\)](#), [ifa, ifi \(78\)](#): can be used as functions.
 - [scan \(87\)](#), [wait S](#): can find all.
 - [opt \(97\)](#), [getopt](#): keychar.
 - [ShowDialog](#): flag 128 - hidden.
 - [tim \(92\)](#): flag 2 - one-shot timer.
 - [IStringMap \(115\)](#): [AddList](#) and [GetList](#) support CSV.
 - [StrCompareEx](#), [ICsv.Sort](#), [DlgGrid.Sort](#) and [ARRAY.sort](#): flag to sort as date.
 - [run \(64\)](#), [getopt \(98\)](#), [newitem \(108\)](#), [AutoPassword](#), [QmHelp](#), [CaptureImageOrColor](#), [__Font.Create](#), [RunAs](#), [Sqlite.ToQmGrid](#), [str.fix \(212\)](#), [str.flags \(234\)](#), [Htm.Attribute](#).
 - More options in string functions that use [delimiter characters \(263\)](#).

- String formatting functions: [%S, %C, %m, %#s \(248\)](#).
4. New in run-time errors:
 - Defined [constants \(144\)](#) for errors.
 - Added menu Run -> Show RT Errors. Read more in [err \(129\)](#) help.
 - For user-defined functions, shows errors in [function help \(245\)](#).
 - [end \(131\)](#): flag 8 (warning).
 - [opt \(97\)](#), [getopt](#): nowarnings.
 5. New in [function help and tags \(245\)](#):
 - Some new tags and other features.
 - Better formatting of function help.
 - Function help editor.
 6. New in status bar:
 - Shows more function info. Colored.
 - Shows function info when caret is in its arguments.
 - Shows function info when you click a function in popup lists.
 - You can set status bar text. Use [OutStatusBar](#). Supports [tags \(245\)](#), like with [out](#).
 - Can show > 2 lines. Vertical scrollbar. Wraps lines.
 - Added menu. You can turn off mouse info, etc.
 7. New in code editor:
 - Can be set style for variables.
 - Member functions now are colored.
 - Shows modified lines in selection bar.
 - 'Find' dialog: Highlights all if you click 'Find Text' button with Ctrl. Added 'Dialog 2' button (use two Find dialogs).
 - And more.
 8. New in [autotext lists \(29\)](#) (TS menus):
 - Text trigger. Don't need a key trigger before typing text.
 - Options for postfix.
 - Shows popup list if several items match.
 - And more.
 9. New in recording:
 - [Records \(43\)](#) text keys as characters, not as key codes.
 - Adds comments with decoded keys.
 10. New in tool dialogs:
 - The Keys dialog easier to use.
 - New dialog: Edit application command keys (floating toolbar -> more tools). You can change or disable actions assigned to keys such as Back, Mute, Calculator.
 11. Updated documentation of many functions.
 12. Can show annotations with short function descriptions. Menu -> Help -> Annotations.
 13. Added [type declaration character \(266\)](#) ` for [VARIANT](#).
 14. New triggers: QM events -> QM show, QM hide, Recording ended.
 15. [Hybrid paste \(58\)](#).
 16. More templates in menu -> File -> New -> Templates.
 17. New in [toolbar right-click menu \(25\)](#): Hide if Fullscreen Window. Super On-Top (replaces the "Toolbars: on-top" checkbox in Options that previously would be applied to all free toolbars).
 18. By default hides private Running items, eg QM extensions (System). To unhide, right click a folder or empty space.
 19. Optimizations.
 20. Bug fixes:
 - If base class is declared as private (`class X :--Y ...`), functions of the inherited class cannot access members of the base class.
 - QM hangs when using some user-defined classes, like this: `class CFont :--GdiObject'g`.

272. What's new in versions 2.1.0 - 2.3.2

- `pointer._new(variable)` returns incorrect value. Same with `_resize`.
- `wait ... S "macro:..."` does not automatically add the macro to exe.
- Fixed problems with processes that are started using short path (eg c:\xxxxx~1\yyyyyy~2.exe). Then in some cases triggers and process functions did not work as expected, because QM used short path. Now QM always uses long path. Also fixed problems with long process filenames on Windows 2000.
- `getopt(speed 4)` always returns 0.
- `Htm.Attribute` flags ignored.
- Gets incorrect location of some html elements in IE8/9: when zoom is not 100%; in some frames/iframes; some AREA elements.
- `acc` does not find CLIENT of frames/iframes in Internet Explorer.
- `acc` fails if **window** is accessible object and used > 8 arguments.
- `acc` in some windows is too slow if flag 128 (reverse).
- `key "text"` may type incorrect text if the active window uses a non-default keyboard layout.
- Array elements cannot be used as some arguments of some QM intrinsic functions.
- Cannot return some types (e.g. Acc) from user-defined functions. Exception or some other error.
- On some XP computers does not show "Temporarily Unlock Computer" configuration dialog.
- Sometimes does not add tray icon at Windows startup.
- QM tray icon problems on Windows XP if taskbar is auto-hide.
- `ProcessNameTold` does not support special folders. Also third parameter of `win`.
- Cannot be used [0] in F strings, like `F"{s[0]}"`.
- Mouse double click triggers don't always work if not using low level hook.
- Temporarily Unlock Computer: sending Ctrl+Alt+Delete does not work on Windows Vista/7.
- Accessible object triggers often don't work on 64-bit Windows.
- `str.escape` bugs.
- `_getactive` may return object of incorrect type when wildcard used.
- Passing COM interface variables to user-defined functions: if type different, later does not release the object.
- QM crashes if `newitem` called while editing a macro name or called repeatedly with flag 8.
- `tim`: if there are multiple timers, cannot stop some of them.
- `findl`: does not work if `n` omitted.
- And more.

21. What can be incompatible with previous versions:

- If you have functions with same names as the new functions, will need to rename or delete them. Same with classes and other declarations.

Below are some changes since QM 2.3.3.4. Also included above.

QM 2.3.3.5 - 7

1. New `ExcelSheet` functions/options.
2. New functions: `str.LimitLen`, `str.GetClipboardHTML`, `Acc.GetChildObjects`, `MakeExeCloseRunning`, `EndThread`.
3. Autotext list (TS menu) option /m (confirm).
4. `Hybrid paste (58)`.
5. Added keyword `paste`, as alias for `outp`.
6. `IStringMap (115)`: `AddList` and `GetList` support CSV.
7. `StrCompareEx`, `ICsv.Sort`, `DlgGrid.Sort` and `ARRAY.sort`: flag to sort as date.
8. `HtmlDoc`: with `SetOptions` flags 1 or 2, may work slightly differently than in earlier QM 2.3.3 versions. Please test your code.
9. More templates in menu -> File -> New -> Templates.
10. By default hides private Running items, eg QM extensions (System). To unhide, right click a folder or empty space.
11. More options in string functions that use `delimiter characters (263)`.
12. New in `toolbar right-click menu (25)`: Hide if Fullscreen Window. Super On-Top (replaces the "Toolbars: on-top" checkbox in Options that previously would be applied to all free toolbars).

13. Some bugs fixed. QM 2.3.3 bugs: 1. Incorrect selection bar color. 2. str.gett works only with ASCII string. 3. Bad autotext synchronization in some cases. 4. HtmlDoc.SetOptions(2) does not work in exe. 5. Triggers don't work in some windows after restarting QM. 6. And more.

QM 2.3.2 (August 4, 2010)

Exact version: 2.3.2.8

1. [Variables in strings \(138\)](#).
2. New functions and classes. Many previously available in the forum.
 - Class [MenuPopup](#) and function [ShowMenu](#).
 - Classes [Sqlite](#) and [SqliteStatement](#). Sqlite database functions.
 - Class [DateTime](#) and functions [TimeSpanFromParts](#), [TimeSpanFromStr](#), [TimeSpanGetParts](#), [TimeSpanGetPartsTotal](#), [TimeSpanToStr](#), [DaysInMonth](#).
 - Class [__Tcc](#). Compile C code. Call functions from QM. Make exe, dll.
 - [str](#) functions [LoadUnicodeFile](#), [SaveUnicodeFile](#), [DateInFilename](#), [SqlEscape](#), [FromGUID](#).
 - [ExcelSheet](#) functions [SelectCell](#), [SelectRange](#), [GetSelectedRange](#), [GetUsedRange](#), [RunMacro](#).
 - [HtmlDoc](#) function [FindHtmlElement](#).
 - Functions [FileTypeRegister](#), [FileTypeUnregister](#), [FileTypeAddVerb](#), [FileTypeRemoveVerb](#), [FileTypeSetDefaultVerb](#).
 - Functions [GetIdleTime](#), [WaitIdle](#).
 - Functions [SelStr](#), [SelInt](#).
 - Functions [HtmlToWebBrowserControl](#), [RemapKeyboardKeys](#), [outw](#), [MakeInt](#), [IntCheckConnection](#) (alias for [IntPing](#)), [Http.Get](#).
 - New dll functions [q_printf \(114\)](#), [q_sort \(114\)](#), [iscsym \(114\)](#), [DetectStringEncoding \(114\)](#), [UnloadDll \(114\)](#), [ConvertSignedUnsigned \(114\)](#), [StrCompareEx \(114\)](#), [QmCodeToHtml \(114\)](#).
 - Categories [keytext](#) (replaces [io](#)), [_debug](#), [_operator](#), [_other](#).
 - Directive [#ret \(181\)](#).
 - GDI reference file. GDI+ flat API declarations.
 - Several new [ICsv \(116\)](#) functions. Some functions have more options. All functions can be used in exe.
 - [QM_Grid \(119\)](#) control enhancements. Class [DlgGrid](#). **Possible incompatibility**: changed some [QM_Grid](#) control features. If you used the control in your macros, please test them.
 - [Ftp](#) functions [FileGetStr](#), [FilePutStr](#), [SetProgressDialog](#), [SetProgressCallback](#).
 - [Http](#) functions [SetProgressDialog](#), [SetProgressCallback](#), [PostAdd](#), [Get](#) (replaces [FileGet](#)).
3. More options in:
 - [category \(159\)](#). Can be added class members and other strings.
 - [scan \(87\)](#). Find not exact match, search in background window, in bitmap, and more.
 - [wait](#). Supports custom cursors. Look in the Wait dialog.
 - [str.encrypt](#), [str.decrypt \(209\)](#). LZO compression. MD5 of file.
 - [Crc32 \(114\)](#). CRC of file.
 - [act \(72\)](#), [str.escape \(210\)](#), [str.setfile \(217\)](#), [OnScreenDisplay](#).
 - [#opt hidedecl \(176\)](#).
 - [type](#). Easier nonstandard [alignment \(156\)](#).
 - [ARRAY \(146\)](#). One new function (insert). More options in some functions.
 - [str.getstruct \(222\)](#). Supports members of any type.
 - [shutdown \(104\)](#). Flags 4 and 8 (synchronous).
 - [__Font](#). Create. Angle, template.
 - [Ftp.Connect](#). Passive.
 - [Ftp.FileGet](#), [Ftp.FilePut](#), [Http.Post](#), [Http.PostFormData](#), [IntGetFile](#). All these functions now can: 1. Show progress dialog. 2. Use callback function. 3. Run in other thread. 4. Download to file.
 - [Http.PostFormData](#). Can be used [Http.PostAdd](#) instead of array.
4. Enhancements in information features:
 - [Lists of functions \(46\)](#) work better. Better sorting, autocompletion, changed icons, and more.

- Can erase category name when typed `categoryname.` and selected something from the list. Look in Options.
- F2 works with class member variables and locally declared variables.
- And more.

5. Various enhancements:

- Functions in `WINAPIV` and `WINAPI7` now are delay-loaded (dll-). Added some missing declarations.
- In `exe (51)`, if a dll not found in exe folder, also searches in QM folder, if installed.
- Possible to execute functions while compiling for exe. For example, `#compile* Function` previously was error.
- Shift+drag selection of multiple QM items for cut/paste.
- `act (72)` works better with multiple Excel and PowerPoint document windows.
- Accessible object triggers work in QM window.
- "Test" button in "Find Image" and "Enumerate Files" dialogs.
- List of QM shortcuts in "Create shortcut" dialog.
- `ARRAY`, `BSTR` and `VARIANT` destructors don't reset `GetLastError`.
- While QM is ending a thread (on "end macro" command, file reload, etc), the thread now can use `mac`, `dis` and other functions that previously would cause incorrect thread termination, especially if they were in destructors or `atend` functions.

6. New tools:

- Remap Keys. Look in floating toolbar, "More Tools" menu.

7. Bug fixes:

- Some functions unavailable in exe (`__ProcessMemory`, `GetListViewItemText`).
- Incorrect mouse trigger hit test in some windows on Windows 7.
- Sometimes missing tray icon at startup on Windows 7.
- QM occasionally hangs for several seconds when pressed hotkey or other trigger.
- `ret` returns invalid COM object in some cases.
- `Dir.FileSize` returns incorrect file size of very big files.
- `HtmlDoc` InitX functions may change some characters.
- `DATE` from/to string assignment uses system default locale. If date format of user and system default locales are different, possible error or unexpected date string format. Now changed to user's locale. **Possible incompatibility:** now in such case will be different date string format than before.
- Fixed several bugs related to declarations and reference files.
- Destructors and `atend` functions not always called when user ends thread without terminating.
- On user accounts other than where QM was installed, does not work menu Edit -> Active Items -> Open Multiple -> Open Automatically.
- `ARRAY.sort` does not work with arrays of interface pointers.
- `ARRAY` functions sort and shuffle don't work properly if `lbound` is not 0.
- Cannot load imagelist bmp file in exe from resource.
- In some cases allows to call private functions of base class from function of inherited class.
- And more.

8. What can be incompatible with previous versions:

- If you have functions with same names as the new system functions, you will have to rename or delete them. Same with classes and other declarations.
- See the red "Possible incompatibility" above.

QM 2.3.1 (September 5, 2009)

Exact version: 2.3.1.9

1. Many internal code changes and optimizations.
2. Some new `debug features (99)` - variables, call stack, and more.
3. New functions and classes: `str.timeformat (236)`, `OutWinMsg (114)`, `Statement (114)`, `PsCmd`, `PsFile`, `str.ConvertEncoding`, `__ProcessMemory`.
4. More options in: `deb (99)`, `tok (192)`, `str.dospath (208)`, `Htm.CbSelect`.

5. Works better: [acc](#) (with WPF windows), [GetListViewItemText](#).
6. Fast [encrypting/decrypting \(14\)](#) multiple QM items.
7. F1 and F2 in the [help search field \(4\)](#) work in a similar way as in the code editor.
8. [Output history \(8\)](#).
9. Added [reference file \(160\)](#) WINAPI7 with Windows 7 declarations. Also updated WINAPI2 in the forum.
10. Bug fixes.

QM 2.3.0 (April 29, 2009)

Exact version: 2.3.0.13

Summary: Supports Unicode. Does not run on Windows 98/Me. Improved code editor. Changes in QM user interface. A faster way to find help and tools. Updated WINAPI, added Vista declarations. New functions. XML and CSV functions. Can open external text files. Custom icons of QM items. Shell menu triggers. Bookmarks and saved searches. Printing. QM Pro not needed for exe.

Warning: In this version there are changes that may break some of your existing macros. Read about possible compatibility problems [here](#).

1. Supports [Unicode \(267\)](#).
2. Does not run on Windows 98/Me. The last QM version that supports these OS - QM 2.2.1 - is [here](#).
3. Improved code editor. Some related info is [here \(7\)](#) and [here \(114\)](#). You can change colors and fonts in Options -> Editor.
4. Open Items and Running Items lists. To show, check View Active Items in Edit or Run menu.
5. [Find help, functions, tools \(5\)](#). Previously it was available in the forum. If you have downloaded it, delete "QM quick help search" folder from the list of macros.
6. Custom [icons \(16\)](#) of QM items.
7. More options in [Icons dialog \(16\)](#).
8. Imagelist editor.
9. Shell menu triggers. Previously available in the forum. If you have it, delete 'Shell menu triggers' folder.
10. [Bookmarks and saved searches \(4\)](#).
11. Other changes in QM user interface: docked Find dialog, changes in menus and toolbars, changed some control ids, changed some colors, icons, added Back and Forward buttons.
12. Updated [WINAPI \(160\)](#) reference file. Added W function versions. Added wininet functions, and removed inet typelib. Removed some rarely used functions: cryptography, XML, some other. Added WINAPIV reference file containing Vista declarations.
13. New functions and interfaces: [ICsv \(116\)](#), [IXml, IXmlNode \(117\)](#), [empty \(185\)](#), [SetEnvVar](#), [GetEnvVar](#), [SetCurDir](#) (replaces [ChDir](#)), [GetCurDir](#) (replaces [CurDir](#)), [CB_Add](#), [LB_Add](#), [str.Swap](#), [WaitForThreads](#), [GetIpAddress](#), [ScreenColors](#), [WinA](#) (find window containing acc. object), [Drag](#), [outb](#) (display binary data), [RunConsole2](#), [FileCopy](#), [FileMove](#), [FileRename](#), [FileDelete](#), [CloseWindowsOf](#), [MsgBoxAsync](#), [__ImageList](#), [__Hicon](#), [StrCompare \(114\)](#), [StrCompareN \(114\)](#), [MemCompare \(114\)](#), [GetUserInfo \(114\)](#) (user/computer name), [GetFullPath \(114\)](#) (from relative), [GetQmCodeEditor \(114\)](#), [InsertStatement \(114\)](#), and more.
14. New predefined variables: [_unicode \(144\)](#).
15. New operators: [@ \(134\)](#) (converts string to Unicode UTF-16).
16. More options in: [str.unicode](#), [str.ansi \(238\)](#), [findrx \(197\)](#) (flag 32), [str.replacerx \(230\)](#) (flag32), [rget_rset \(106\)](#), [str.escape \(210\)](#), [interface \(165\)](#) (optional parameters; comments; multiple [function calls \(168\)](#) in single statement), [type \(154\)](#) (comments), [out \(57\)](#) (colors, links), [len \(184\)](#) (supports BSTR, VARIANT, word*), [newitem \(108\)](#) (temporary), [lef/rig/mid/dou \(53\)](#) (return), [opt \(97\)/getopt](#) (hungwindow), [zip \(71\)](#) (no compression), [GetFileIcon \(114\)](#) (cursor, custom size), [RegWinClass](#), [MainWindow](#), [MessageLoop](#), [OpenSaveDialog](#) (hwndowner), [OnScreenDisplay](#) (picture).
17. Added QM item type - [file link \(19\)](#). Added related options to [qmitem \(107\)](#), [str.getmacro \(219\)](#), [newitem \(108\)](#) and in other places.
18. Added QM item property - temporary.
19. New triggers: [QM events \(33\)](#) -> File link run, End thread.
20. [Special folders \(246\)](#) more reliable. Several new special folder names and aliases.
21. [Exe \(51\)](#): Any files can be added to resources using syntax ":resourceid filepath". For example, you can use jpg and gif images in exe dialogs. Previously it worked only with bmp and ico files and only with some functions. Now it works

with most functions that load a file, including [str.getfile \(217\)](#). Also now can add cursor resources.

22. Toolbars: If you drag and drop a file onto a program icon, you can choose to open the file in that program.
23. Name completion. Partially type an identifier (function, etc) and press Ctrl+Space or use menu Edit -> Members.
24. Indented [multiline strings \(137\)](#).
25. Works better: [Loading files and HTML into a web browser control \(63\)](#). Some optimizations in [win](#), [child](#), [acc](#) and [htm](#). Ftp and Http functions don't show error descriptions in QM output; added Ftp/Http member variable lasterror, which will contain error description if a function failed (returned 0). Toolbars show nondistorted icons of sizes other than 16/32, if icons of that size are available.
26. Printing. Menu File -> More -> Print.
27. You can change backup folder path in Options -> Files.
28. New [menu option \(28\)](#): /multicolumn.
29. Changed behavior of "record window command" hotkey. Now shows popup menu with several choices. Removed "record mouse command" hotkey.
30. [Exe \(51\)](#): QM Pro license now is not needed. Currently there are no other features that would require Pro license.
31. Tested and runs well on Windows 7 beta. Works well with Internet Explorer 8.
32. Can check for new QM version (Options -> General).
33. Bug fixes.

Compatibility with previous Quick Macros versions

1. Unicode. If QM is running in ANSI mode, you will not have compatibility problems. It is running in ANSI mode if you upgraded QM from an older version. It is running in Unicode mode if you did not use QM before, or uninstalled it before installing new version. You can switch mode in Options. It is recommended to use Unicode. After switching to Unicode mode, you should convert your old macros that contain international characters. The Convert button helps you to do it. [Read more \(267\)](#).
2. This QM version does not run on Windows 98/Me. Exe files created with it too.
3. QM user interface. Changed some control types, ids, positions, etc. If you have macros that interact with them, they may stop working. Changed code editor control type. Read more [here \(114\)](#).
4. Added some QM extensions that previously were available in the forum. You should delete downloaded extensions. While not deleted, QM shows warnings.
5. QM item icons. Everything will work but some icons in menus, toolbars and dialogs will be different than before. Read [more \(16\)](#).
6. Updated WINAPI reference file. Removed some rarely used declarations.
7. New functions. If you have functions with same names, rename them. On name conflicts QM shows warnings or errors in output.
8. Changed behavior of some functions in some specific cases. With [rget, rset \(106\)](#), slightly changed behavior when a data type is explicitly specified. With [str.escape \(210\)](#), flag 2 now is ignored. With [out \(57\)](#), if text begins with <>, it is interpreted as containing formatting tags (colors, links, etc), and found tags are removed.
9. [win](#), [child](#), [acc](#), [htm](#). Changed search order. Now at first retrieves visible windows/controls, then hidden. [win](#) and [child](#) may find another control if matchindex is specified. Retrieved arrays of handles also may be ordered differently. This may happen only if some of matching windows or controls are hidden. Use window/control finder dialogs to fix the code.
10. Now uses different way to get paths of special folders. Normally this should not change anything, but if previously some paths were incorrect or could not be retrieved, now they will be correct.

QM 2.2.1 (January 1, 2008)

Exact version: 2.2.1.5

Summary: Many bug fixes. New functions, new options in existing functions. Better reliability of keyboard and clipboard commands.

1. Fixed 2.2.0 version bug: QM does not run on Windows XP without SP. QM-created exe files too.
2. Fixed 2.2.0 version bug: If using a locale where , is decimal separator, QM may start to improperly interpret numbers containing . . In exe too.
3. Fixed 2.2.0 version bug: On some Vista computers, scheduled tasks are not updated after editing.
4. Fixed 2.2.0 version bug: [VbsExec](#) and other VBScript functions don't work.

5. Fixed bug: On computers where hardware Data Execution Prevention is enabled for all programs, don't work dialogs and other code that uses callback functions.
6. Fixed [mac \(100\)](#) and [DynamicMenu](#) bugs with synchronous menus: Returns 0 if first line item selected. Does not wait if a menu is already shown.
7. Fixed bug: dll functions with double parameters give exception if passed certain expressions.
8. Fixed bugs in [wait K/M](#) in exe: On Windows 98 may not work. On Vista may not work first time in thread.
9. Fixed bugs in: [def](#) (strings cannot have ; characters), [clo \(73\)](#) (some windows crash), toolbars (sometimes are activated wrong windows), [OnScreenRect](#) (Vista may hang if the rectangle is large, eg 10000x10000), mouse triggers (incorrect hit test codes), [ifk](#) (on Vista sometimes does not work), [wait K/M](#) (the same as with [ifk](#)).
10. Improved synchronization of [key \(56\)](#) and clipboard functions. Better reliability, especially in stress conditions (low memory, busy CPU, etc).
11. New options used with [opt \(97\)](#) (keysync) and [getopt \(98\)](#) (slowkeys, slowmouse, keymark, keysync).
12. Better [wait \(88\)](#) precision (2 ms).
13. [Make exe \(51\)](#): Can compress.
14. [Dll functions \(153\)](#): 1. Supports optional parameters. 2. Supports . . . for variable number of parameters.
15. Added easier way to add [ARRAY \(146\)](#) elements.
16. New functions: [str.stem](#), [str.getstruct](#), [str.setstruct \(222\)](#), [ARRAY.sort_shuffle \(146\)](#).
17. New user-defined functions and classes: [OsdHide](#) (hide on-screen display), [GetVirtualScreen](#) (rectangle that contains all monitors), [ChooselconDialog](#), [Htm.ClickAsync](#), [str.RandomString](#), [str.Guid](#), [str.ReplaceInvalidFilenameCharacters](#), [str.GetFilenameExt](#), [str.FilenameWithDate](#), [GetFilesInFolder](#), [GetDrive](#), [GetDrives](#), [TagWindow](#), [FindTaggedWindow](#), [IsFunctionRunning](#), [MouseButtonX](#), [CenterWindow](#), [EnsureWindowInScreen](#), [XyMonitorToNormal](#), [XyNormalToMonitor](#), [MoveWindowToMonitor](#), [FirstWindowInMonitor](#), [SetWindowState](#), [HtmlDoc](#) class (HTML parsing), several new functions and classes in System\Functions\Programming. Some of these functions previously were available in the forum. You may have to delete duplicates.
18. New [dll functions exported by QM \(114\)](#): [GetProcessExename](#), [RealGetNextWindow](#), [MonitorFromIndex](#), [MonitorIndex](#), [AdjustWindowPos](#), [RealGetKeyState](#), [Round](#), [RandomNumber](#), [RandomInt](#).
19. Added predefined variable [_monitor \(144\)](#) that controls where various dialogs are displayed. Some functions modified to support it.
20. More options in: [StartProcess \(114\)](#) (window state), [GetWorkArea](#) (flags, monitor), [OnScreenDisplay](#) (flag 8 - hide when macro ends, flag 16 - raw x y, flag 32 - place by the mouse, supports [_monitor](#)), [ShowDialog](#) (flag 64 - raw x y, supports [_monitor](#)), [SetWinStyle](#) (flag 16 - update client), [wait \(89\)](#) S (can be used handle, can wait until something changes in the window or rectangle), toolbars ([3D buttons \(25\)](#)), [key \(56\)](#) (can return virtual-key codes), [IntGetFile/Http.GetUrl/Http.FileGet](#) (reponseheaders), [Http.PostFormData/Http.Post](#) (inettags, reponseheaders), [RegisterComComponent \(114\)](#) (can register 64-bit dlls, flag 8), [DT_GetControls](#), [DT_SetControls](#), [win \(77\)](#) (can get all matching windows), [child \(83\)](#) (the same), [list](#) (owner window), [inp](#) (owner window), [inpp](#) (owner window).
21. New categories: sysinfo, multimedia.
22. Email functions: 1. Support secure connection (SSL). To use secure connection, check the checkbox in Outlook Express Accounts or QM Email Settings. 2. Account settings can be specified in code.
23. Changed [ShowDialog \(63\)](#) position algorithm (differently depends on monitor, styles, hwndowner, etc).
24. Slightly changed behavior of: [win \(77\)](#) (prefers visible windows), [child \(83\)](#) (does not prefer visible windows if matchindex is nonzero), [create \(167\)](#).
25. Works better: [ifk \(59\)](#) on Vista, [scan \(87\)](#) with icons.
26. [Log loaded dlls and type libraries \(8\)](#).
27. The Find dialog supports AND and OR.
28. Changed double to string conversion (str operator = and other operators). Now the string never has decimal point at the end. The string can have more digits (better precision). The exponent sign is uppercase. The exponent does not include 0 at the beginning. If previously the string could look like 1.e+016, now it looks like 1E+16.
29. Toolbars: [/bsiz \(26\)](#) works with all styles, and added minwidth.
30. Other bug fixes and optimizations.

QM 2.2.0 (June 25, 2007)

Exact version 2.2.0.11

Summary: Runs well on Windows Vista. New winapiqm.txt. Macros can run in separate process. New functions, enhancements and new options in existing functions and other features.

1. QM now is **compatible with Windows Vista**, 32-bit and 64-bit versions. Read about it [here \(277\)](#). Also you may want

to read about [creating exe files for Vista \(51\)](#).

2. **Updated winapiqm.txt reference file (160)** (WINAPI). Also, in the QM forum you can find and download another reference file that contains more declarations.
3. Enhancements related to [reference files \(160\)](#): 1. Not error to explicitly declare something that is implicitly declared using a reference file. 2. Removed "optional" settings from Options (now everything from WINAPI is always available). 3. Reduced memory usage. 4. Can include other files. 5. Can log extracted declarations.
4. Enhancements related to [user-defined types \(154\)](#): 1. [Anonymous types within types \(156\)](#). 2. [Base type without member name \(157\)](#).
5. Enhancements related to [keyboard \(30\)](#) and [mouse \(31\)](#) triggers: 1. Can be used **low level hooks** (optionally), which make triggers more reliable and allow to use system hot keys. 2. A trigger key can be pressed repeatedly without releasing modifier keys.
6. New functions: interface [IStringMap \(115\)](#), [RunConsole](#), [lock \(112\)](#), [str.dospath \(208\)](#), [str.addline \(204\)](#), [ReceiveMail](#), [#exe addfile \(179\)](#), [ExeExtractFile](#), [ExeGetResourceData \(114\)](#), variable [_win64 \(144\)](#).
7. More options in: [Dir.GetSize](#) (can get folder size), [str.findreplace \(211\)](#) (replace null char), [mes \(62\)](#) (Vista shield icon), [opt \(97\)](#) keymark, [rset \(106\)](#) (delete key if empty), [web \(94\)](#), [wait \(89\)](#) (multiple handles), [shutdown \(104\)](#) (thread handle/id), [str.expandpath \(231\)](#) (unexpand), [EnumQmThreads \(114\)](#) (thread name), [OnScreenDisplay](#) (wrap), [newitem \(108\)](#) (encrypted), [run \(64\)](#) (run as admin), [_getactive \(167\)](#) (ROT moniker), [toolbar hook \(27\)](#) (WM_INITDIALOG), [Dir.FileName](#) (DOS format), [ShutdownProcess](#), [EnumProcessesEx \(114\)](#), [win \(77\)](#) (matchindex, exename), [child \(83\)](#) (matchindex), [acc \(85\)](#) (matchindex).
8. Small changes in: [rget \(106\)](#) (ini files), [wait \(89\)](#) x H (returns -1 on error), [Speak](#) (changed implementation; some changes in behavior with flag 2), [wintest \(77\)](#) (class is not required for hidden windows).
9. Several new [dll functions exported by qm.exe \(114\)](#).
10. Macro and function properties: **"Run in separate process", "Run As"**.
11. [Popup menus \(22\)](#): **Expandable file folders**, bitmaps and more.
12. [Special folders \(246\)](#): 1. Can be used CSIDL constants. 2. Environment variables are expanded recursively.
13. **Better supports ActiveX controls (63)**. In the [Type Libraries dialog \(163\)](#) added Controls view.
14. [Make exe \(51\)](#): 1. Can be specified custom manifest file. 2. The macro can be in a folder or not. 3. Adding files to exe (see [#exe addfile \(179\)](#), [ExeExtractFile](#), [ExeGetResourceData \(114\)](#)).
15. For compatibility with Windows Vista and some programs, [key](#) and clipboard commands now use different synchronization methods than in previous versions. In most cases, these commands will work as well as previously, but some macros may require some corrections ([wait](#), [err](#)). Also, after copy/paste commands, the clipboard contents now is restored immediately.
16. Pasting ([outp](#) and [str.setsel](#)) now works in console windows too.
17. These features are no longer supported in [exe \(51\)](#): [wait](#) x KF (wait for key-down and eat), [share](#) (shared memory). They can be used only in QM. In exe they would cost too much, especially on Windows Vista and 64-bit.
18. These features are now supported in exe: [shutdown -1](#) (exit exe), [Exit](#) (exit exe). Fixed [shutdown -7](#) bug.
19. In exe, all text and function names now are encrypted.
20. Now maximal text length is 4 MB for menus, toolbars and TS menus. For other items it remains 32 KB.
21. A global hotkey to show Threads dialog can be specified in Options.
22. Menu Edit -> Wrap Lines.
23. Options -> Files -> Always Add These Shared Files.
24. Toolbars: 1. Some changes in [hook functions \(27\)](#) (WM_INITDIALOG, etc). 2. New [toolbar options \(26\)](#) /glass, /font, 1-pixel border. 3. Setting text color does not disable visual styles. 4. If you use [GetWindow](#) or other Windows API function to get toolbar owner window, you should change it to [GetToolbarOwner](#), because [GetWindow](#) will return incorrect result.
25. File triggers: DOS filenames are no longer returned for Added and Modified events. You should modify Include/Exclude fields that contain DOS filenames.
26. Other enhancements and bug fixes.

QM 2.1.9 beta (September 29, 2006)

Exact version 2.1.9.1

Summary: Can create exe files from macros. Several new functions and new options in existing functions. Different dialog procedure.

1. **Make exe**. Can create standalone programs from macros and functions.

2. New directives: [#out](#), [#warning](#), [#error \(178\)](#), [#exe \(179\)](#). More options in [#opt \(176\)](#) (nowarnings).
3. New functions: [Http.PostFormData](#).
4. New predefined [variables and constants \(144\)](#).
5. More options in: [ShowDialog \(63\)](#) (icon, menu, hotkeys, resources), [dll \(153\)](#) (delay-loading), [out \(57\)](#) (clear), [scan \(87\)](#) (handle, resource), [LogFile](#), [IntGetFile](#) (callback function), [Http.FileGet](#) (callback function), [Http.GetUrl](#) (callback function), [Curtain](#) (returns window handle), [dir](#) and [Dir.dir](#) (finds drives).
6. Several new [dll functions exported by qm.exe \(114\)](#).
7. In [dialog \(63\)](#) procedure now it is not necessary to call [DT_Init](#) and other DT_x functions.
8. [File triggers \(34\)](#) more reliable. Fixed 2.1.9.0 bug with network folders.
9. Changed [End-macro hotkey \(13\)](#) behavior.
10. Other enhancements and bug fixes.

QM 2.1.8 (June 15, 2006)

Exact version 2.1.8.8

1. New trigger type - [QM events \(33\)](#). Special items now are included as QM events triggers, and removed from Options.
2. New trigger type - [file \(34\)](#). For Windows 2000/XP and later.
3. New trigger type - [event log \(35\)](#). For Windows 2000/XP and later.
4. New trigger type - [process \(36\)](#). For Windows 2000/XP and later.
5. New trigger type - [accessible object \(37\)](#).
6. [User-defined triggers \(39\)](#) now can be added to the Properties dialog.
7. Better integration with the Task Scheduler.
8. Added option to [unlock computer \(268\)](#) to run the macro in normal conditions.
9. All trigger settings now are included in the [trigger string \(264\)](#).
10. All [toolbar options \(26\)](#) now can be set in Properties. Added new toolbar options to set background picture and transparency.
11. New option for functions: Allow single instance.
12. New functions: [Speak](#), [SpeakStop](#), [OnScreenDisplay](#), [RunTextAsFunction2](#), [Curtain](#), [SetFileTimes](#), [GetClipboardFiles](#), [VbsExec2](#), [AutoPassword](#), [RunAs](#), [Acc.ObjectFromEvent](#), [ExcelSheet.Save](#), [Database](#) class, several new [dll functions exported by qm.exe \(114\)](#).
13. More options in: [IntPost](#), [Http.Post](#), [MouseWheel](#), [ExcelSheet.Init](#), [FE_ExcelRow](#), [BrowseForFolder](#), [RegWinClass](#), [ShowDialog](#) (x, y), [ShutDownProcess](#), [str.encrypt](#), [str.decrypt \(209\)](#).
14. Works better: wait for web page ([web \(94\)](#), [wait \(89\)](#)). Previously, with some multiframe pages, did not wait for all frames.
15. Works better: [_error \(129\)](#). Now it is properly populated in functions registered by [atend](#).
16. Works better: calling nondeclared COM functions. This also makes working with WMI easier.
17. Works better: now dialogs with DS_SETFOREGROUND style are really activated, regardless of system settings and other conditions.
18. Works better: [_setevents \(169\)](#).
19. New dialogs: Get file info, Enter password.
20. Dialogs, toolbars, [str.setclip](#) and [str.setsel](#) now support image files of bmp, jpg and gif type.
21. In [menus \(24\)](#) can be used icon handles.
22. Timer functions ([tim \(92\)](#)) now run in new thread each time.
23. Redesigned Properties and Options dialogs.
24. The command line parameter E now always causes to exit QM, even if an error occurs.
25. Now passwords that are passed to functions can be [encrypted \(14\)](#). All QM functions that accept password now support encrypted passwords.
26. New help topic about [user-defined functions \(150\)](#).
27. Other enhancements and bug fixes.

QM 2.1.7 (Jun 14, 2005)

1. [Integrated help \(245\)](#): Tips for beginners, and function help on F1. Reassigned F1 and F2 hotkeys, and unassigned

F3.

2. New functions and classes: [scan \(find image on screen\)](#), [perf \(91\)](#) (get time in microseconds), [matchw \(196\)](#) (compare strings using wildcard characters), class [Htm](#) (wraps html element functions; it also includes mouse actions and enhanced selecting of combo box items), class [ExcelSheet](#) (get/set cell values), functions [FE_ExcelRow](#) (get cell values), [GetAttr](#), [SetAttr](#) (file attributes), and several other.
3. Several qm internal functions now also are [exported as dll functions \(114\)](#).
4. More options in: [wait \(89\)](#) (wait for image on screen), [opt \(97\)](#) (slowmouse, slowkeys), [ref \(160\)](#) (declare identifiers on demand but always see them in the main popup list), [web \(94\)](#) (get window handle, use *), [sel \(125\)](#) (use *), [ShowText](#) (rich text).
5. Work better: wait for CPU on Windows 2000/XP and later (eliminated first-time delay and high memory consumption).
6. Toolbars: menu -> [Quick Icons \(25\)](#). Other performance enhancements. Toolbar option [/bsiz \(26\)](#) (max button size).
7. QM runs in [safe mode \(8\)](#) if F8 is pressed when starting.
8. You can [change My QM folder path \(15\)](#). It is where your default macro list file and various other files used in QM are stored.
9. [Command line \(18\)](#): you can specify some other character to use instead of ".
10. Wildcard characters: functions and triggers, where * can be used, now also support ?.
11. [Dialogs \(63\)](#): now you can use **graphical ActiveX controls**, including the web browser control.
12. COM [event functions \(169\)](#): simpler access to the event-source object.
13. New help topic about [using COM components \(163\)](#).
14. [Thread variables \(142\)](#): no more "invalid pointer" errors, because the declaration statement now may be skipped during the run time. If there is a constructor, now it runs at the beginning, not where the variable is declared.
15. Type libraries: now are supported names containing invalid characters.
16. Various small enhancements and bug fixes.

QM 2.1.6 (Mar 17, 2005)

1. [Quick Start \(2\)](#) topic.
2. New commands: [deb \(99\)](#) (set debug mode), [net \(101\)](#) (run macro on other computer).
3. Easier [network setup \(258\)](#).
4. Command line parameter [N \(18\)](#) (wait for a network file).
5. Improved menu recording. Toolbar menu item Record Menus.
6. Toolbar option [/lock \(26\)](#).

QM 2.1.5 (Feb 2, 2005)

1. New functions: [htm \(86\)](#) (replaces [HtmlFind](#)), [web \(94\)](#) (open web page; replaces [leNavigate](#)), more Internet functions (connect, disconnect, get/set offline mode, download, http post, functions and classes for sending and retrieving email), [mkdir \(68\)](#) (replaces [MkDir](#)), [inpp \(61\)](#) (replaces [Password](#)), [zip \(71\)](#) (compress/extract files), [ref \(160\)](#) (add reference file), [ARRAY.remove \(146\)](#), [ShellDialog](#) (Run, Shut Down and other dialogs), [Play](#) (play audio files) and several other user-defined functions.
2. More options in: [wait \(89\)](#) (wait for color, cursor, web page, and more), [str.escape \(210\)](#) (urlencode), [str.encrypt/decrypt \(209\)](#) (various algorithms), [findrx \(197\)](#) (get n-th submatch), [acc \(85\)](#) (wait), [typelib \(164\)](#) (load on demand), [foreach \(128\)](#) (without VARIANT), [call \(132\)](#) (function name as string), [shutdown \(104\)](#) (softly end current thread), [qmitem \(107\)](#) (datemod), [str.expandpath \(231\)](#) (env. var.), Tray functions, [Ftp.Cmd](#) ("write" commands), [ChDir](#) (autorestore), [toolbars \(25\)](#) (all virtual desktops).
3. Work better: [act](#) (activate next window), [ArrangeWindows 0](#) (show desktop), [ShowText](#) (removed 64 KB limitation), [rget](#) (supports REG_EXPAND_SZ), new Internet functions.
4. The following commands can also be used as functions (variable=function(...)): [min](#), [max](#), [res \(74\)](#), [hid \(76\)](#), [men \(80\)](#), [dis \(102\)](#), [spe \(90\)](#), [opt \(97\)](#).
5. Type-info: declared several useful type libraries - Wsh (Windows Script Host), Shell32, [Services \(121\)](#), etc, and reference file WINAPI containing Windows declarations (type `WINAPI` and select from list).
6. Embedding multiline [strings \(137\)](#) without escaping.
7. Menus, toolbars, TS menus: can be used escape sequences.
8. You can [selectively disable \(13\)](#) key, mouse, window and command line triggers.
9. Disabled items don't run from command line triggers (shortcut, scheduler, etc). Also, special items now also don't run if

file is disabled.

10. Menu Edit -> Copy BBCode. Copies colored code for the QM forum.
11. [File viewer \(17\)](#). Makes importing and undeleting easier and safer.
12. [Spam filter](#). Checks email and deletes spam.
13. QM extensions now are better protected from initialization failures. In Options, added 'Extensions: always use' checkbox and 'Check extensions' button. The System folder now is read-only.
14. Items in read-only folders cannot be modified in any way. Previously, read-only was only text. The 'Properties' dialog now is always accessible (even if macro is encrypted), but properties cannot be changed if macro is in read-only folder.
15. Now, variables can be used in all fields of dialogs from the floating toolbar. Instead of using 'Variable' checkbox, enclose variables into parentheses. This is necessary only for string fields, and must not be used where there is 'Variable' checkbox.
16. More dialogs, more options in existing dialogs.
17. New [command-line \(18\)](#) parameter MS allows you to execute functions synchronously.
18. When exporting, you can set allowed file-open methods (open, import, use as shared file).
19. New setup program.
20. QM now works better on computers with multiple user accounts. QM now is installed for all user accounts. The default place where QM now saves various files is the My QM folder in My Documents (previously - QM folder), that is, separate files for each user. Some files are moved from QM folder to My QM folder.
21. [My Macros \(8\)](#) window. View all triggers in one list.
22. Menu File -> Sort.
23. Windows XP visual styles (themes) applied. [Possible problems](#)

In user-defined dialogs (created in QM Dialog Editor), lines become invisible. To make them visible, replace them with Static controls with SS_ETCHEDHORZ or SS_ETCHEDVERT style. To do it quickly, use Find dialog. Check Regexp. In first edit field enter `^(\d+)Button 0\d+7 0\d+(.+ ""\r\n)`. In second edit field enter `$1Static 0x54000010 0x20004$2` (for horizontal lines) or `$1Static 0x54000011 0x20004$2` (for vertical lines). Replace all found lines in dialog definitions.

Note: after upgrading to 2.1.5 version, many settings are reset to defaults, like after installing first time.

QM 2.1.4 (Jun 27, 2004)

1. [Recording \(43\)](#): improved reliability; can use variables.
2. More options in [act \(72\)](#), [opt \(97\)](#), [getopt \(98\)](#) and [win \(77\)](#).
3. Changed behavior of [end \(131\)](#) in callback function.
4. Functions from System [shared folder \(17\)](#) now have higher priority.
5. Removed all triggers from the System folder.
6. [Toolbar startup options \(26\)](#): better parsing; can be used functions.
7. Better support for drag and drop in toolbars. You can drop QM items, Internet links, multiple files and virtual folders/objects.
8. Fixed several bugs.

QM 2.1.3 (May 30, 2004)

1. More dialogs, more options in existing dialogs.
2. Added support for multiple monitors. This includes mouse commands, mouse triggers and toolbars.
3. More options in [key triggers \(30\)](#) (next key), [mouse triggers \(31\)](#) (X buttons, window parts), [window triggers \(32\)](#) and [TS menus \(29\)](#). Window trigger coding is changed. Window triggers now are more reliable.
4. New commands and functions: [foreach \(128\)](#), [str.setmacro \(220\)](#) and several new functions in System\Functions folder.
5. More features in: [sel-case \(125\)](#), [key \(56\)](#), [newitem \(108\)](#), [child \(83\)](#), [but \(81\)](#), [list](#), [mes \(62\)](#), [run \(64\)](#), [rget rset \(106\)](#), [str.fix \(212\)](#), [ARRAY.redim \(146\)](#), [ArrangeWindows](#), [Dialog Editor \(63\)](#) and [toolbar options \(26\)](#).
6. In window triggers and all functions that search for window or other object by name ([win \(77\)](#), [child](#), [acc](#), [HtmlFind](#)), flag "Exact" is replaced with "Use *". Behavior is the same, unless string contains *, or if "Exact" was used together

with "Regexp".

7. Regular expressions: \$ now matches either newline, or carriage return and newline.
8. Fixed several bugs.

QM 2.1.2 (Feb 24, 2004)

Main new features: now QM can recognize much more types of user interface objects, including html elements in web pages (e.g., links), and programmatically interact with them.

1. Now you can work with accessible objects. Accessible objects are user interface objects contained in a window: controls, menus, links, list items, etc. New features include functions [acc](#), [acctest \(85\)](#) (find/test accessible object), [Acc](#) class (manipulate accessible object), "Find accessible object" and "Accessible object actions" dialogs.
2. Now you can work with html elements in web page (links, buttons, text input fields, etc). New features include function [HtmlFind](#) (find html element), related functions, "Find html element" and "Html element actions" dialogs.
3. New classes: [Acc](#), [Tray](#) (add tray icon).
4. More features in: [interface \(165\)](#), [str.getsel](#), [str.setsel \(216\)](#), [opt \(97\)](#), [err \(129\)](#), [spe \(90\)](#), [case \(125\)](#), [mac \(100\)](#), [newitem \(108\)](#), [Dialog Editor \(63\)](#), [toolbars \(23\)](#), [toolbar startup options \(26\)](#).
5. Now [err](#) and [opt](#) statements can handle exceptions.
6. Now formatting functions don't produce "(null)" when string variable isn't initialized.
7. Fixed bugs in: [iif](#) (when used as argument with some functions), [qमितem](#), menu/toolbar icons, and several other.

QM 2.1.1 (Jan 03, 2004)

1. Changed Help file format.
2. Now you can copy QM text in HTML format.
3. New classes: [Dir](#) (find and enumerate files, get file info).
4. More features in [Replace](#) dialog (regex menu, query saving), [run \(64\)](#) (get exit code).
5. Fixed bugs in: [end \(131\)](#) (in callback), [str.getclip \(216\)](#) (unknown format causes exception), deleting duplicate function (other function becomes unknown), [GetFileInfo](#) (wildcards), [leNavigate](#) (hwnd=3).

QM 2.1.0 (Dec 17, 2003)

Main new features: COM support, classes, categories, safe arrays, thread variables, type-info popups (members, globals), regular expressions, trigger scope for folder, new and extended functions. Below is complete list.

1. Now QM supports [COM \(162\)](#), OLE Automation variable types and type libraries.
2. Better [type info \(46\)](#).
3. New functions: [str.replace \(229\)](#), [str.findreplace \(211\)](#), [str.expandpath \(231\)](#), [str.encrypt](#), [str.decrypt \(209\)](#), [getopt \(98\)](#), [numlines \(187\)](#), [newitem \(108\)](#), [#err \(175\)](#), [ChDir](#), [CurDir](#), [RunTextAsMacro](#), [RunFileAsMacro](#), [leNavigate](#), [leWait](#), functions to execute VBScript and JScript code, [shutdown \(104\)](#), [findrx \(197\)](#), [str.replacerx \(230\)](#), [#set \(177\)](#).
4. More options in: [dll \(153\)](#), [function \(153\)](#), [type \(154\)](#), [end \(131\)](#), [atend \(103\)](#), [run \(64\)](#), [cop. ren \(65\)](#), [del \(66\)](#), [mac \(100\)](#), [wait \(89\)](#), [tok \(192\)](#), [#opt \(176\)](#), [#if \(172\)](#), [#compile \(174\)](#), [str.time \(235\)](#), [str.from \(214\)](#), [val \(186\)](#) and [TS menus \(29\)](#).
5. Deleted functions: [float](#) (now use FLOAT type).
6. More strict type checking. Use [operator + \(134\)](#).
7. Now user-defined and dll functions can have declared parameters and return values of any type. Parameters can be structures passed by value.
8. Now menu/toolbar items of "statements" type are executed as functions therefore can run while macro is running.
9. [Safe arrays \(146\)](#).
10. [COM events \(169\)](#).
11. A hotkey to end macro can be specified in Options.
12. Folder and file [properties \(11\)](#): "Scope" (scope of triggers), "Application" (private functions, variables, etc.), "Read-only items", "Disabled items" and other.
13. New variable scopes: [thread variables \(142\)](#) (function-private and shared).
14. New dialog: Navigate in Internet Explorer (open URL, Back, Stop, etc).
15. Better error handling. See [#err \(175\)](#) and [_error \(142\)](#).

16. [Regular expressions \(198\)](#).
17. New topic in Help: COM collections.
18. Removed "Strict type checking" from Options (now always strict). Use [operator + \(134\)](#).
19. Changed: now you always must use struct.member syntax to access first member of structure, unless structure has single member.
20. State data of functions [dir \(69\)](#), [findt \(190\)](#), [findl \(191\)](#), [str.gett \(223\)](#) and [str.getl \(218\)](#) now has local scope (was thread scope). Each function stores it in separate local variable: [_dir, _findt, _findl, _gett and _getl \(144\)](#).
21. [Classes \(157\)](#) and [categories \(159\)](#).
22. New classes: File, Ftp, Http, Dde.
23. Functions and classes are collected into categories.
24. [Memory-allocation functions \(148\)](#) [_new](#), [_resize](#), [_delete](#) and [_len](#).
25. Mouse triggers: "Slow" option.
26. Fixed bugs in [tok](#), [tim](#), [findrx](#), str.replacex, str.from, [bee+](#), [CB_SelectString](#) functions, "import", "insert encrypted text" features and more.
27. Fixed bugs in long numbers, break, err+.
28. Other bug-fixes and enhancements.

Window/object enumeration callback function

A callback function is a user-defined function. You pass its address to a function that supports callback functions, and then that function calls your callback function, usually multiple times, for example for each found object of some kind, providing information about the object.

In this topic described callback functions used with functions [win](#), [child](#), [acc](#) and [Acc.Find](#). Some other functions also support callback functions in a similar way.

Callback functions can be used when you need to: 1. Get array of windows/objects. 2. Compare more properties of windows/objects. 3. Display list or tree. 4. Make faster by skipping some big useless objects. 5. ...

The callback function is called for each window/object whose all specified properties match. To call the callback function for all windows/objects, all other arguments should be empty ("" or 0). With [win](#), to include hidden windows, insert [opt_hidden](#) before.

[win](#), [child](#)

With functions [win](#) and [child](#), callback function and its argument can be specified in **propCSV**, like `F"callback={&Function} {aValue}"`. Before QM 2.3.4, used **x y** and flag 0x8000.

When searching, [win](#) and [child](#) functions enumerate top-level or child windows. The callback function is called for each matching window.

The callback function must begin with:

```
function# hwnd cbParam
```

hwnd - handle of the found window.

cbParam - callback function argument passed to [win](#) or [child](#).

The callback function can return 1 to continue enumeration or 0 to stop (let [win](#) or [child](#) return **hwnd**). For example, it can test some additional window properties (rectangle, text inside, etc) and, if they match, return 0, else return 1.

A template is available in menu -> File -> New -> Templates.

[acc](#), [Acc.Find](#)

Similarly, with functions [acc](#) and [Acc.Find](#), the callback function must begin with:

```
function# Acc&a level cbParam
```

a - the found object.

level - its level in the hierarchy. If **class** is specified, it is level beginning from that child window. When searching in web page (flag 0x2000), it is level from the root object of the web page (DOCUMENT or PANE).

cbParam - **y** argument of [acc](#).

The callback function can return 0 to stop searching, 1 to continue and search possible children, or 2 to continue and skip children.

A template is available in menu -> File -> New -> Templates.

Window expressions

When window is required as part of a macro command, you can use:

1. Window **name**. Can be partial, must match case. Must be enclosed in quotes, unless it is variable. QM searches only visible windows, unless [run-time option \(97\)](#) `opt hidden 1` is set. Examples:

```
act "Notepad"

str s = "Notepad"
act s
```

If **name** begins with \$ (but not with \$\$), it is interpreted as [regular expression \(198\)](#). For example, `act "$^A.*Notepad$"` activates window whose name begins with "A" and ends with "Notepad". To do case insensitive search, use (?i). For example,

```
act "$(?i)^A.*Notepad$".
```

Empty string ("") in most cases is interpreted as the active window.

2. Window **class** name. Must be full or with wildcard characters (QM 2.3.4), case insensitive, with + prefix. QM searches all windows. Example:

```
act "+ieframe"
```

3. List of window names and/or +classnames, delimited by [] (new line). If first window not found, searches for second window, and so on. Example:

```
ifa "Window1[]$Window\d[]+Window3"
_out "One of three windows is active"
```

4. Window **handle**. Unlike window name and class, it can also be used with user-defined and dll functions. It is an integer value and is returned by functions [win \(77\)](#), [id \(82\)](#), [child \(83\)](#) and some other. Examples:

```
int h = win("Notepad")
act h

act win("Notepad")
```

Window expression (handle or name, etc) also can be stored in a variable of type VARIANT.

With macro commands, if window (still) does not exist, QM waits max 0.5 second, unless speed ([spe \(90\)](#)) is 0 or window handle (variable or function except [win](#)) is used.

Window **name** is text that is displayed in the title bar.

Each window or child window belongs to some window **class** (e.g., "IEFrame", "Button", etc). Window class defines appearance and behavior of all windows that belong to it. Window class is set by software developer and cannot be changed. However, some windows are created with different class each time (e.g., those that begin with "Afx:"). You can see window name and class name in QM status bar.

Window **handle** is an unique numeric value that the system assigns to each window and child window when creating it. It does not change until the window is destroyed.

Child window **id** is a numeric value that the developer assigns to it. In most cases, it is unique within its parent window. Sometimes id may be set at run time and it may be random value.

See also: [window styles \(275\)](#)

Window styles

Top-level windows (windows)

A top-level window is a window that is not a part of other window.

Overlapped window

An overlapped window is a top-level window that typically has a title bar, menu bar, border and client area. It is meant to serve as an application's main window. Example - Quick Macros main window.

Pop-up window

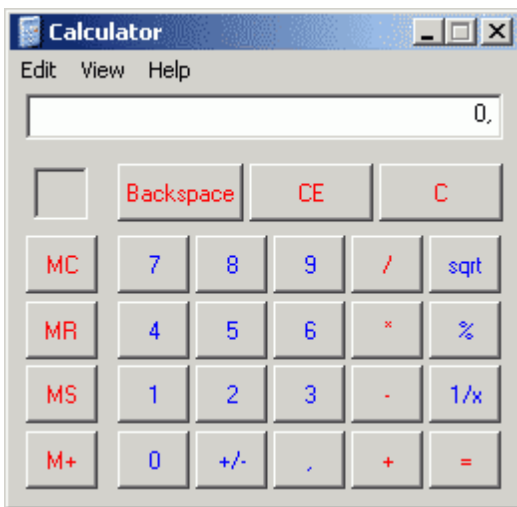
A pop-up window is a special type of top-level window used for dialog boxes, message boxes, and other temporary windows. Example - Quick Macros Options dialog. Popup windows have WS_POPUP style flag (0x80000000). Window style is displayed in QM status bar. If first character after "0x" is 8, 9, A or B, it is popup window.

Child windows (controls)

A child window (or "control") is confined to the client area of another window. Client area is window area except title bar (with buttons and icon), menu bar, scroll bars and border. The parent window (window that contains the child window) can be top-level window or another child window. An application typically uses child windows to divide the client area of a parent window into functional areas.

Almost all macro commands that work with windows also work with child windows.

In the picture, Calculator is top-level window. It has several child windows - edit, static, several buttons. For example, C is child of Calculator, and Calculator is parent of C. Buttons in the title bar, and the menu bar, are not child windows. This area, plus border, is nonclient area.



Windows keyboard shortcuts

Some useful Windows keyboard shortcuts:

Windows and documents

Wd	show desktop
Wm	minimize all
SWm	undo minimize all
We, Wf, Wr, WF1, WG	Explorer, Find, Run, Help, System
AT	previous window
AZ	next window
CT	next document
CST	previous document
CF4	close document
T, ST	select input fields in a web page

Menus and dialog controls

A{ } or F10	activate menu bar
M or SF10	context menu
Y	OK button
Z	Cancel button
T, CT, ST, CST	select controls, tabs
L, R, U, D	select group controls, list items, tabs
V, Y	click selected control

Edit

Cx	cut
Cc	copy
Cv	paste
Cz	undo
Cy	redo
Ca, CHCSE	select all
SL, SR	select left or right character
CSL, CSR	select left or right word
SH, SE	select left or right part of line

Use Alt + underlined letter to click menus, buttons, etc.

Tip: You can also use commands [men](#) and [but](#) to click menu items and buttons.

Windows Vista, 7, 8, 10; Windows 64-bit

User Account Control

Windows Vista has a new security feature - *User Account Control (UAC)*. With UAC, even on administrator accounts most processes (running programs) have limited privileges. It creates problems for many programs. This topic describes problems that may have QM when running on Vista, and gives workarounds for most of them. You also may want to read about running [QM-created programs \(51\)](#) on Vista.

Here "Vista" actually means "Vista and later". On Windows 7, 8 and 10 everything is similar as on Vista.

On Vista, there are several predefined privilege sets, also called *integrity levels (IL)*. An IL is assigned to a process (running program) before starting it, and cannot be changed while it is running. The table gives some information about different integrity levels.

IL	Comments
High	<p>The process runs as administrator, like on Windows XP.</p> <p>When starting a process that needs administrator privileges, usually is shown a dialog with name "User Account Control" (<i>consent</i> dialog). The process then is called <i>elevated</i>. The program can be marked to require such privileges, or you can run it as administrator using the right-click menu, or you can set it to run as administrator in file properties dialog, or if Windows decides that it is a setup program. A process launched by a process that has administrator privileges also has administrator privileges, but does not require a consent.</p> <p>By default, QM runs with High IL (as administrator), although you can change it in Options. To create better user experience, a consent dialog is not shown when QM starts. Otherwise QM would be blocked at Windows startup. Also, processes launched by QM have Medium IL by default.</p>
Medium	<p>The process runs as standard user, like on a non-administrator account. It has limited privileges. For example, It cannot write to Windows and Program Files folders, cannot write to most registry keys, cannot manipulate services, and much more. Also, it cannot interact (use keyboard, mouse and menu commands, send messages, use hooks, etc) with higher IL processes.</p> <p>On Vista, most processes have Medium IL. Windows Explorer also has Medium IL. QM runs with Medium IL if in Options is selected UAC: run as User, which is not recommended.</p>
uiAccess	<p>The process has Medium IL, but is allowed to interact (use keyboard, mouse and menu commands, send messages, use hooks, etc) with High IL and uiAccess processes.</p> <p>Only few programs have uiAccess privileges. QM runs with uiAccess privileges if in Options is selected UAC: run as uiAccess, which is recommended if you don't want QM running as administrator. On non-administrator accounts, QM runs as uiAccess by default.</p>
Low	<p>The process has minimal privileges. It can write only to several predefined folders and registry keys. Normally, with Low IL runs only Internet Explorer, when protected mode is on. QM does not have an option to run as Low, but macros that are set to run in separate process can run as Low.</p>
System	<p>Highest privileges. Normally only services and some system processes run with System IL. QM does not have an option to run as System.</p>

Q. What IL uses QM? Can it be changed?

A. By default QM runs as administrator. It can be changed in Options.

Q. Should QM run as administrator, uiAccess, or standard user? When it has less problems with UAC?

A. QM has less problems when it runs as administrator. Then almost everything that worked in previous operating systems also works on Vista.

Q. Is it safe if QM runs as administrator?

A. It is quite safe. Programs started by QM ([run](#)) have medium IL. If you think it is not safe, you can set it to run as uiAccess. The User mode (Medium IL) is not recommended.

Q. Can some macros run with different IL than QM?

A. Yes, if they run in separate process. You can set it in Properties.

Q. Is it possible to turn off UAC?

A. Yes, you can completely turn off UAC in Control Panel -> User Accounts. Then security level will be the same or slightly higher than on Windows XP. Also, some options can be changed in local security policy (run "secpol.msc"). For example, you can set to elevate without consent, or to show consent in default desktop.

Q. Is it possible to run a program as administrator without a consent dialog?

A. Use flag 0x10000 or 0x20000 with [run](#). Or use function [StartProcess \(114\)](#). Or, in macro properties, check "Run in separate process" and select "Administrator" or "Highest available". It is possible only from QM (except portable). Without QM, it is possible for example using Windows Task Scheduler.

Q. Is it possible to automatically close the consent dialog for some programs?

A. There is no such option in Windows. QM also cannot automate it because the dialog is created in a secure desktop. If the dialog is not in a secure desktop (you can set it in local security policy), then you can create a function that closes it.

You can find more information about UAC in Vista Help and on the Internet. If you need QM-specific information, you can ask about it in [QM forum](#).

See also: [GetProcessUacInfo \(114\)](#), [IsUserAdmin \(114\)](#)

What does not work on Vista

These problems are common to all integrity levels.

1. When you launch protected mode Internet Explorer, actually are started two processes. The first process usually exits immediately (but in IE8 not). For this reason, all "wait for" options don't work with [run](#). For the same reason, SHDocVw.InternetExplorer functions don't work if you create the object using `_create`. Possible workarounds: 1. Use [web](#) instead. 2. In Properties, check "Run in separate process" and select Low. 3. Turn off IE protected mode. 4. Turn off UAC.
2. May fail file functions that use mapped network drive (like "Z:\file"). Workaround: Use path like "\\server\share\file".
3. And maybe more (not yet discovered).

To solve most other UAC-related problems, you can set QM or separate macros to run with appropriate IL. The information below should help you decide what IL you should use.

What does not work on Vista when QM is running as Administrator or uiAccess

Some operations are not allowed between different IL processes. Although most of them are not allowed only when initiated by the lower IL process, some of them also are not allowed when initiated by the higher IL process.

1. [_getactive](#), `GetObject` (VBScript) and some other COM functions cannot get COM objects from different IL processes. [_getactive](#) also is used in some other functions. Possible workarounds: 1. In Properties, check "Run in separate process" and select same privileges as of the target app (usually User). 2. Start the target app using `_create`. 3. Run both QM and the target app as administrator. 4. Turn off UAC.
2. Drag and drop from Medium IL processes (e.g. Windows Explorer) to QM. QM uses another process to reenale its drag and drop feature. However, `WM_DROPFILES` in custom dialogs does not work. Possible workarounds: 1. Use [QmRegisterDropTarget \(114\)](#) instead. 2. In Properties, check "Run in separate process" and select User. 3. Turn off UAC.
3. And maybe more.

What does not work on Vista when QM is running as User or uiAccess

Some functions and other QM features require administrator privileges. If QM (or [exe \(51\)](#)) is not running as administrator, these features don't work.

1. Writing to some file system locations, such as Program Files and Windows folders ([cop](#), [ren](#), [del](#), [MkDir](#), [SetAttr](#), [str.setfile](#), etc).
2. Writing to some registry keys, such as `HKEY_LOCAL_SYSTEM` and `HKEY_CLASSES_ROOT` ([rset](#)).
3. Automatic COM object registration by [_create \(167\)](#) (because cannot write to the registry).
4. [RegisterComComponent \(114\)](#). You can use flag 4 to show consent dialog.
5. [GetDiskUsage \(114\)](#). It uses PDH functions that require administrator privileges. Does not work on Vista only; works on 7/8/10.
6. [SetPrivilege \(114\)](#).
7. Manipulating services.
8. Changing computer date.
9. Some COM functions, including [_getactive](#) and `GetObject` (VBScript), don't work with different IL processes. Read more above.
10. On Windows 8 QM does not see Windows store app windows.
11. And maybe more.

Everything above also does not work on non-administrator user accounts on all OS.

Possible workarounds: 1. In Properties, check "Run in separate process" and select Administrator. 2. Run QM as administrator. 3. Turn off UAC.

What does not work on Vista when QM is running as User

The following functions don't work with higher IL windows unless QM (or [exe \(51\)](#)) is running as administrator or uiAccess. This is more actual for exe, because QM can run as administrator or uiAccess.

1. Keyboard and mouse commands ([key](#), [paste/outp](#), [str.getsel](#), [str.setsel](#), [lef](#), [mou](#), [Acc.Mouse](#), [ifk](#), [wait K](#), [wait M](#), and other). Mouse commands don't work in any window if currently active window has higher IL.
2. Windows API functions that send messages ([SendMessage](#), [PostMessage](#), etc). Only few messages can be sent.
3. Many Windows API functions that manipulate windows ([SetWindowPos](#), [EnableWindow](#), etc).
4. Functions that use [SendMessage](#), [SetWindowPos](#), etc. Most of them are menu and control functions ([men](#), [but](#), [CB_x](#), [LB_x](#), [Acc.DoDefaultAction](#), etc) and window functions ([hid](#), [max](#), [mov](#), [siz](#), [ArrangeWindows](#), [Zorder](#), [Transparent](#), etc).
5. Most hooks. For example, function [BlockInput2](#) (available in the forum) uses low level keyboard and mouse hooks.
6. [BlockInput](#) does not work with all windows.
7. Toolbars cannot be attached to higher IL windows.
8. Windows 8/10: sending system hotkeys (Alt+Tab, Win+R etc) with [key](#). This does not depend on the currently active window.
9. And maybe more.

This should not be a big problem, because normally most programs don't run as administrator. Administrative programs usually are used briefly and don't need to be automated. However, currently there are quite many non-Vista-aware programs that don't work without administrative privileges. For example, if a program saves files in its home directory, which usually is in Program Files, it must run as administrator.

Possible workarounds (QM): 1. In Properties, check "Run in separate process" and select Administrator. 2. Run QM as administrator or uiAccess. 3. Turn off UAC.

Possible workarounds (exe): 1. Run exe from QM: in Properties check "Run in separate process" and select As QM (if QM runs as administrator or uiAccess) or Administrator. If you need to launch it from e.g. desktop, create shortcut to run the macro (in Properties). Of course, QM must be installed. 2. Set uiAccess="true" in the manifest, sign the exe file, and put it in Program Files folder. It works well on any computer (QM is not needed). Read more about signing in the [make exe \(51\)](#) topic. 3. Run exe as administrator. It requires consent, unless exe is started from another program that is running as administrator. 4. If possible, don't run target programs as administrator. 5. Turn off UAC.

Vista bugs

1. uiAccess programs cannot open folders using [ShellExecute\[Ex\]](#) if "Launch folder windows in a separate process" is checked in Control Panel -> Folder Options -> View. It is unchecked by default. The [run](#) command, which uses [ShellExecuteEx](#), uses a workaround for this. However it will fail in [exe \(51\)](#) running with uiAccess privileges (exe can run with uiAccess privileges only if launched by QM or marked as uiAccess in manifest). Also, if you use [ShellExecute\[Ex\]](#) or other functions that call it, they will fail. Possible workarounds: 1. Uncheck the checkbox. 2. If QM is running as uiAccess, and "Run in separate process" is checked in Properties, select something other than As QM. 3. Use `run "explorer.exe" "folder"` instead of `run "folder"`. 4. Turn off UAC.
2. QM cannot load some type libraries (maybe about 1%) because they are incorrectly registered. The OLE/COM Object Viewer also cannot open these type libraries. Possible workarounds: 1. If possible, with [typelib](#) use path instead of GUID. 2. Edit the registry: remove double quotes from type library path.
3. And maybe more.

64-bit Windows

QM is a 32-bit program, but it runs well on 64-bit Windows.

If QM is running on 64-bit Windows, [special variable \(144\)](#) `_win64` is 1, else 0.

Notes

On 64-bit Windows, there are separate System and Program Files folders for 64-bit and 32-bit programs.

Special folder "\$program files\$" expands to the 32-bit folder ("Program Files (x86)"). QM is installed there. Environment variable "%ProgramW6432%" expands to the 64-bit folder ("Program Files").

Special folder "\$system\$" expands to the 64-bit folder path ("C:\Windows\System32"), but actually is used the 32-bit folder ("C:\Windows\SysWOW64"). That is, [run \(64\)](#) will launch 32-bit versions of programs, unless flag 0x4000 used. See also [this forum post](#).

Also, there are some separate registry locations for 32-bit programs.

[More info.](#)

Set window always-on-top

Obsolete. Use function [Zorder](#) or [GetWinStyle](#). To create code, use dialog "Window/control actions".

Syntax1 - set state

```
ont [-] [window]
```

Syntax2 - get state

```
int ont([window])
```

Parameters

(274)window - [top-level \(275\)](#) or child window. Default: active window.

Options:

Default	on top.
-	not on top.

Remarks

Syntax1: Makes window always-on-top or not always-on-top, and also activates. Only top-level windows can have always-on-top style. If it is child window, brings it to the top or bottom of the [Z order](#) of direct parent window.

The speed depends on [spe \(90\)](#).

Syntax2: Returns 1 if window has always-on-top style, or 0 if not. If it is child window, returns 1 if it is in the top of the Z order, or 0 if not.