# All About Macros

The information in this document will help you understand and create macros. It includes the following
sections:

- An explanation of macros and some terms to help you understand the rest of this document.
- How to write a macro in Microsoft Project.
- Tips for writing macros including samples that you can copy and paste into your own macros.
- Quick reference list of all the commands used in macros.

For a list of the full command syntax for all the commands used in macros, see the file
COMMANDS.WRI. The information in COMMANDS.WRI is the same as that included in the
Commands Used In Macros topics in Help.

For information about messages you may see when writing or running a macro, use online Help. Press F1 or choose the Help button for an explanation of the causes of the error and suggestions for solutions.

This document does not include information about writing and running Microsoft Project macros from
other applications using DDE. For information about DDE and macros, see the DDEINFO.WRI file.

## What is a Macro

A macro is nothing more than a series of commands. When you find that you often repeat the same set
of commands, such as you might do when creating month-end reports, you can create a macro that
carries out these commands. Then, at the end of the month, you can run the macro and have Microsoft
Project create the reports for you.

The following definitions of commands and arguments will help you understand the information in this
document.

**Command** - A command is an instruction you use in a macro to tell Microsoft Project what to do. The
commands you can use in macros include most of the commands on the Microsoft Project menus plus
additional commands for other actions such as moving and selecting information, working with
windows, or working with an outline. The macro command names for those commands on menus are a
combination of the menu name and the command name. For example, the Open command on the File
menu is called FileOpen. For the action commands, the name indicates the

functionality. For example,
the commands for outlining are preceded with Outline, such as OutlineCollapse and OutlinePromote; the
commands used for selecting information begin with Select, such as SelectAll or SelectCellDown. All
the macro commands are listed at the end of this document and are also in the file COMMANDS.WRI
and in the Commands Used In Macros topics in online Help.

**Argument** - An argument is like an option in a dialog box. Many, but not all, of the macro commands
have arguments. For example, the FileOpen command has two arguments, one of which is a Name
argument that you use to indicate the name of the file you want to open. Each argument is composed of
an **argument name**, such as Name in the FileOpen example, and an **argument value**, such as the actual
name of the file you want to open. The argument name is preceded by a period, and followed by an
equal sign and then the argument value. If the argument value contains characters other than standard
letters or numbers, or contains spaces or the list separator character, place the value in brackets ([ ]) or
quotes (" ").

If you need to use brackets in the argument value, place the entire value in quotes. Or if you need to use
quotes in the argument value, place the value in brackets. There may even be cases when you need to
use both brackets and quotes in the argument value. In that case, use quotes around the entire argument
value and use a double set of quotes where the quotes are required within the argument value. For
example, sending a DDEExecute command to Microsoft Excel may require both quotes and brackets.

For example, if you used the Open command in a macro, it would look like this:

```
FileOpen .Name=[project.mpp]
```

FileOpen is the command, .Name is the argument name, and [project.mpp] is the argument value.

The first argument following a command does not have to include the period, the argument name, or the
equal sign. This is the only exception. For all other arguments, these items must be included because the
argument name indicates what the value is for.

Arguments can follow their command in any order. The argument name specifies what the value
following it is for.

There are two types of arguments: required and optional. A **required argument** is one that must be used for the command to be carried out. For example, the Open command can do nothing unless you tell it a file to open. If you include the required argument values, the command is carried out and the macro continues. If you do not include a required argument value and the command normally displays a dialog box, the dialog box is displayed by the macro so you can enter a value, such as selecting the file to open.
If you do not include a required argument value for a command that does not display a dialog box, the macro won't run until you include values for the required arguments.

An **optional argument** is one that you don't have to include but may want to. For example, the FileOpen command includes the optional argument .ReadOnly=. Microsoft Project can open a file regardless of your preference for this argument. If you want the file opened as read-only, however, you can include this argument. For example, to open the file PROJECT.MPP as read-only, the command line would look like this:

```
FileOpen .Name=[project.mpp] .ReadOnly=[Yes]
```

Some optional arguments have a default value. For example, the ReadOnly argument value is No by default. Other optional arguments have no value unless you use the argument. For example, to print a range of pages, you enter values for the .FromPage and .ToPage optional arguments for FilePrint. If you don't enter an argument, all the pages are printed. The FilePrint command also has an optional argument for the number of copies. The .Copies argument has a default value of 1, which means 1 copy will be printed if you don't use the argument. The default values for those optional arguments that have them are included in the description of each command.

Some commands have only optional arguments. For example, all arguments for the FilePrint command are optional. If you use this command in a macro, the active view will be printed without displaying the Print dialog box. If you want to display the dialog box so you can change the options, add a question mark (?) to the end of the commands. For example, FilePrint? would cause the Print dialog box to be displayed.

There are two ways you can put commands into the macros you write. One way is to type the command followed by the arguments you want to use. If you are very familiar with writing

Microsoft Project
macros, this method may work just fine for you. If you are not familiar with Microsoft Project macros,
there's an easier way. You can scroll through a list of all the macro commands and select each one that
you want to add to the macro. When Microsoft Project pastes a command into the macro, it also pastes
all the required and optional arguments. You can turn this option off if you want, but when you first start
out, its best to paste the arguments so you can pick and choose which you want to use. Arguments are
always pasted with brackets around the default value or as a place holder for the value, so you don't have
to remember to include them. Just type the argument value between the brackets.

When the arguments are pasted with the command, you can distinguish required arguments from
optional arguments by their position relative to the two slashes (//). Information following the two
slashes is interpreted as a comment and ignored when the macro is run. If there are no slashes, all
arguments are required. Otherwise, required arguments are to the left of the slashes and optional ones are
to the right. Enter an argument value for each required argument; select and move any optional
arguments you want to use to the left of the slashes and then fill in the argument values. If you want to
use all the optional arguments, just delete the two slashes. Be sure to enter values for all the arguments
to the left of the slashes. Don't leave any pair of brackets without a value between them. Because the
arguments to the right of the slashes are ignored, you can leave them in your macro without values even
though you aren't using the arguments.

## How To Write a Macro

There are several basic steps to follow when you want to write a macro.

1. Plan the macro. Think about what you want it to do and the commands you would use to complete the
same actions.

2. From the Macro menu, choose Define Macros, and then choose the New button.

3. Enter the commands you want to use in the macro, one command per line. The easiest way to write a
macro is to paste the commands you want in the macro. When you paste the commands, each is pasted
on a new line and the arguments can also be pasted, making it almost impossible to get an error from
mistyping the commands or arguments, or forgetting to include required arguments. To paste a

command, choose the Commands button. Scroll through the list of commands in the Commands dialog
box until you see the one you want. Double-click it, or select the command and choose the Paste button.

4. When you paste commands with arguments, the required arguments are to the left of two slashes (//)
and the optional arguments are to the right of the two slashes. Type a value for each required argument.
Move any optional arguments you want to use to the left of the slashes and type a value for each. To
move an argument, select it and choose the Cut button. Move to the left of the slashes and choose the
Paste button. If you want to use all the optional arguments, just delete the slashes. If you do not remove
the slashes, the arguments following the slashes will be ignored when the macro is run. If you need a
description of an argument or its value, print the COMMANDS.WRI file or check the Commands Used
In Macros topics in online Help. The information in COMMANDS.WRI and in online Help is the same.
COMMANDS.WRI is included so you can print the complete list for reference.

5. Add comments about the macro to help you remember details about the macro construction or to help
others understand it. To enter a comment, type **//** or **--** and then the comment. The two slashes or two
hyphens let Microsoft Project know that the text following is a comment and to ignore it when running
the macro.

6. Use conditional statements or loops as appropriate to control which steps in the macro are used or to
repeat a series of commands. These functions are described later in this document, and several examples
are included in the Tips section.

7. If you are writing a macro for others to use, you may want to add error handling to it and messages
about what is happening or what caused an error. This is described in the Tips section later in this
document.

8. When you are finished writing the macro, choose the OK button and then either choose the Run
button to run the macro or the Close button to close the dialog box.

**Halting a Macro**

RETURN and HALT are used to stop a macro. Use RETURN in an IF conditional statement to end the
macro. Use HALT to end the macro from within a submacro, such as to prevent a macro from returning

to the macro that called it.

Macros are also stopped if errors are on (Error .Halt=Yes) and an error message is displayed or when
you press ESC or the Cancel button outside a loop or If conditional statement.

**Using Conditions to Branch and Loops to Repeat**

Conditional statements and loops are used to control how the steps in the macro are executed.

- IF command ENDIF is used when you want to decide which of two paths to follow through a macro.
- LOOP command ENDLOOP is used when you want to repeat a series of commands.
- NOT can be used with either IF or LOOP to do the opposite of what the command returns.

IF and LOOP depend on the value returned by the command immediately following on the same line.
Each macro command can return one of three values:

- **TRUE** - If a command successfully performed its function, such as if the FileOpen command opens a file, Microsoft Project returns TRUE. TRUE is also returned if the command following IF displayed a dialog box, and you chose the OK button to continue the macro.
- **FALSE** - If a command did not perform its function, Microsoft Project returns FALSE. For example, if you used the FindNext command in a macro, but the macro did not find another task or resource meeting the specified criteria, Microsoft Project would return FALSE. FALSE is also returned if the command following IF displayed a dialog box and you chose the Cancel button or closed the dialog box using the dialog box control menu. If you are within an IF statement or loop, FALSE takes you to the step beyond ENDIF or ENDLOOP; if not, FALSE ends the macro.

- **ERROR** - If an error occurred when the command was carried out, Microsoft Project returns ERROR. For example, ERROR would be returned if the argument for a command were not valid, such as entering a nonexistent format for the FileSaveAs command, or if the command cannot be used on the active view, such as using FormatPalette when the Task Form is the active view. An error always terminates the macro unless you have turned errors off (Error .Halt=No). If you do turn off errors, you can then intercept the errors and display appropriate messages.

**Branching in a Macro**

IF is used to control the path taken through a macro when you have more than one path. For example,
you might have one set of commands you want carried out if you are working with tasks and another set
carried out if you are working with resources. To make the decision about which path to take, IF tests
whether the command following IF on the same line was successfully completed. Depending on the

value returned (TRUE, FALSE, or ERROR), the block of commands following are either carried out or
skipped.

The syntax for IF is:

IF *command*
*command1*
*command2*
*command3*
*etc*
ENDIF

If *command* returns TRUE, which means it was completed successfully, the macro executes the
commands following IF and ending with ENDIF. If the command returns FALSE or error handling is
off, the commands between IF and ENDIF are skipped and the macro either continues with the
command following ENDIF or the macro ends if there are no more commands.

If you use NOT following IF, the opposite happens. For example, IF NOT *command* means that if the
command returns TRUE, the block of commands following IF would not be carried out.

To use the equivalent of IF/ELSE/ENDIF, you use more than one IF statement. For example, if you want
one procedure for tasks and another for resources, you could have an IF statement that started by testing
for a task view and a second IF statement that tested for a resource view. The first set of steps would be
carried out for tasks and ignored for resources. The second set of steps would be carried out for resources
and ignored for tasks.

**Repeating Steps in a Macro**

Loops are used to repeat operations. The macro executes the statements between LOOP and ENDLOOP
until the command following LOOP on the same line returns FALSE or until there is an error in one of
the commands. Instead of testing a command, you can also specify a number of times you want the loop
repeated. If you don't specify a command to test or a number of times for the loop to repeat, the
commands between LOOP and ENDLOOP are repeated endlessly or until one of the commands returns
an error.

The syntax for LOOP is:

LOOP *command*
*command1*

or

LOOP *n*
*command1*

| | |
|---|---|
| *command2* | *command2* |
| *command3* | *command3* |
| *etc* | *etc* |
| ENDLOOP | ENDLOOP |

If the command following LOOP on the same line is successful and returns TRUE, the macro executes
the group of commands between LOOP and ENDLOOP. After executing the commands, the macro
loops back to the LOOP statement and again evaluates the command. If the command still returns
TRUE, the macro repeats the group of statements; if the command returns FALSE, the macro stops the
loop and executes any commands that follow ENDLOOP.

If you use NOT following LOOP, the opposite happens. For example, LOOP NOT *command* means that
if the command returns TRUE, the block of commands would not be executed.

When you use a number instead of a command following LOOP, the loop is repeated the number of
times you specify. For example, if you want to open five files, you could create a loop that was repeated
five times, as follows:

```
LOOP 5
FileOpen
ENDLOOP
```

By not including the required Name argument after FileOpen, the dialog box would be displayed five
times so you could select each file you wanted to open.

## Tips For Writing Macros

This section includes tips and ideas for writing macros. It includes information about:

- Errors and alerts and how to handle them
- Using messages
- Selecting project information in a macro
- Outlining in a macro
- Sending keystrokes from a macro
- Changing the timescale in a macro
- Creating interactive macros
- Printing macros
- Keeping track of loops and conditional statements
- Calling other macros
- Determining the active view
- Checking or replacing values in the project fields
- Setting up a view
- Writing a macro in a word processing application
- Using AppExecute to run a macro in another application
- Using DDEExecute to send commands to another application

- File handling
- Writing macros for others
- International notes

**Alerts and Errors and How to Handle Them**

Microsoft Project includes two commands that you can use to control what happens when an alert
message appears or an error occurs.

**Alerts** - Turns Microsoft Project messages on or off. For example, if you close a file that contains
unsaved changes, Microsoft Project asks if you want to save the changes. If you turn alerts off, this
message will not appear. If you don't want the normal messages from Microsoft Project to be displayed
during your macro, use this command to turn them off. When they are off, Microsoft Project
automatically uses the default response to the message.

To turn off the display of Microsoft Project messages, use the following line:

```
Alerts .Show=[No]
```

Since .Show is the first argument, you could leave out ".Show=" and use: Alerts No

This command has no effect on messages that appear when a dialog box is open.

**Error** - Controls Microsoft Project's response to errors that occur while running a macro. You can turn
off error checking with this command. Microsoft Project will then ignore all errors, and continue running
the macro without displaying any error or messages at all. This state is always changed back to its
default (on) when the macro finishes so if you want errors ignored, you must turn errors off in every
macro.

If you want to control how errors are handled at each point in a macro where you think an error could
occur, use this command to turn off errors, and then use this command again with an IF statement at
each place in your macro where you want to check for an error. Immediately after an error occurs, Error
returns TRUE; if no error occurs, Error returns FALSE. You can then make the macro respond
appropriately for each case.

To turn errors off at the beginning of a macro, include the following line:

```
Error .Halt=[No]
```

Since .Halt is the first argument, you could leave out ".Halt=" and use: Error No

To handle an error in a place in the macro where you expect an error could occur, use the following
lines:

If Error
*Message about what happened*
Return
EndIf

This example says: "If Error returns TRUE, the command carried out just before the IF Error statement caused an error; your message is displayed explaining the error and Return causes the macro to end. If Error returns FALSE, no error occurred in the previous step, your message isn't displayed, and the macro continues with the steps following ENDIF.

**Using Messages**

You can use the Message command to display any message you want. You can display a message about what the macro does, or use it as shown in the previous example to explain an error condition. You can control the buttons in the message box to have an OK button only, OK and Cancel, Yes and No, or Yes, No, and Cancel. The OK and Yes buttons return TRUE; No returns FALSE; Cancel returns ERROR. If you use one of the options with Yes and No, you can change the text on these buttons to anything you want. Using the buttons in the dialog box, you can have the user make a choice about what he or she wants to do and then have the macro carry out the appropriate set of steps.

The following line shows how to display a simple message:

Message "This macro requires the Task Name Form to be available."

When the macro encounters this line, it displays the message text in a dialog box along with an OK button. Although the OK button was not specifically included with the command, it is included by default if you don't include one of the arguments for buttons.

The following line shows how you can use a message following IF to control where the macro goes next based on input from the user:

If Message "Do you want to add an object to a task or a resource?" .Type=3 .YesText=Task .NoText=Resource

When the macro encounters this line, it displays the text "Do you want to add an object to a task or a resource?" in a dialog box. The .Type argument value of 3 specifies that you want to include the Yes, No, and Cancel buttons in the dialog box, with the text "Task" and "Resource" instead of "Yes" and "No". If you choose the Task button, the command returns TRUE, and the commands following the IF statement are carried out. If you choose the Resource button, the command returns FALSE; the macro skips the commands following the IF statement and executes the commands following ENDIF instead. If

you choose the Cancel button, the command returns ERROR, which you can also check for and have the
macro respond as appropriate.

**Selecting Project Information in a Macro**

There are several commands you can use in a macro to move around the project. The commands
beginning with "Select" are used to specify what you want selected in the project so the following
command can act on the appropriate data. These commands include SelectAll, SelectBeginning,
SelectCellDown, SelectCellLeft, SelectCellRight, SelectCellUp, SelectColumn, SelectEnd, SelectRow,
SelectRowEnd, SelectRowStart. You can also use the ExtendSelection command to extend the selection
to include noncontiguous fields.

For example, you might use the SelectAll command to select all tasks in a project before collapsing the
outline to print just the main project phases. Or you might use the SelectBeginning command to move to
the start of the project, and then use SelectCellDown to step down through each displayed resource to
check for certain information.

You can also use the SendKeys command to send the keystrokes you would use to select in your project.
For more information about SendKeys, see "Sending Keystrokes from a Macro" later in this section.

**Outlining in a Macro**

In a macro you can work with an outline, just as you can using the outline buttons on the entry bar. Use
the commands beginning with Outline to do this: OutlineCollapse, OutlineDemote, OutlineExpand,
OutlineExpandAll, and OutlinePromote. Before including the Outline command, include the appropriate
commands to select what you want to work on.

For example, the following steps would collapse the entire outline:

```
SelectAll
OutlineCollapse
```

The following steps would now expand the first summary task in the collapsed outline:

```
SelectBeginning
OutlineExpand
```

You could also use EditFind to find a certain task and then promote, demote, collapse, or expand it.

**Sending Keystrokes from a Macro**

The SendKeys command sends keystrokes to Microsoft Project, just as though you had pressed the keys yourself. You can use this command to open a dialog box and select options, to move around Microsoft Project, or any time you want to control Microsoft Project with keystrokes.

For example, the following line display the Text dialog box:

```
SendKeys .keys=[%rt]
```

The percent sign is the macro notation for ALT, followed by "r" for the Format menu, and "t" for the text command. These are the keys you would press to display the Text dialog box. Of course, you could also use the FormatText command to do the same thing in a macro. If you want to display the dialog box and then have the macro select options in the dialog box, use SendKeys to do this. If you want to display the dialog box, and have the user select options in the dialog box, you can use either method. If you display a dialog box using its command, such as FormatText, the macro stops until you respond to the dialog box so a SendKeys command sending keystrokes to the dialog box won't be carried out.

The following line displays the Text dialog box, and then changes the text size to 10 for critical tasks on the Task Sheet:

```
SendKeys .keys=[%rt{down 2}%s10{return 2}]
```

As in the previous example, "%rt" displays the Text dialog box. "{down 2}" is equivalent to pressing the DOWN ARROW key twice. Since the Item To Change box is active when the Text dialog box is displayed, DOWN ARROW twice selects Critical Tasks, the third item in the Item To Change box when the Task Sheet is active. Because the position of Critical Tasks in the Item To Change box is not the same for all task views, the number of times you move down may be different for another task view. You could replace "{down 2}" with "c" instead. "c" would move to the first item in the list that starts with C, which is Critical Tasks. "%s" selects the Size box and "10" types 10 in the box. "{return 2}" sends Return twice, first to enter the value 10 in the Size box and then to choose the OK button to close the dialog box.

The keys you can use, such as {return} and {down} are listed in the SendKeys description in COMMANDS.WRI and in Commands Used In Macros in online Help. Also described are the characters you use to specify key combinations beginning with SHIFT, CTRL, and ALT.

**Changing the Timescale in a Macro**

You may want to change the timescale from within a macro, perhaps while printing a series of reports. You can do this in a couple ways. One way is to use the timescale commands:

TimescaleZoomIn and
TimescaleZoomOut. These commands are equivalent to the buttons on the tool bar.

The other way you can change the timescale is to use the FormatTimescale command. This command
does not have arguments; instead, it displays the Timescale dialog box so you can select what you want
in the dialog box. If you want to automate this process, use the SendKeys command to send the
keystrokes for opening the dialog box and selecting options. If you do use SendKeys to change options
in the Timescale dialog box, there are a couple of ways to move to the option you want in the Units or
Label box. After you select the box you want, you can either send the letter for the first letter in the
option name, or first move to the top or bottom of the list and then move down or up the appropriate
number of times to select to the option you want. End with {return 2} to complete the last selection and
close the dialog box.

For example, the following line selects Quarters for the major timescale and None for the Minor
timescale:

```
SendKeys .keys=[%rmq%nn{return 2}]
```

"%rm" displays the Timescale dialog box, "q" selects Quarters (you could also use "{up 6}" to move to
the top of the Units options, and then "{down 1}" to move down one to select Quarters). "%n" moves to
the Units box for the Minor timescale, and "n" moves to the None option (you could also use "{down 7}"
to move to the last option). "{return 2}" completes the selection and closes the dialog box.

If you have a view you use often with a certain timescale settings, the easiest method is to create a new
view with the timescale set as you want it. Then apply the view in the macro instead of using the macro
to change the timescale.

**Creating Interactive Macros**

You can create macros that prompt for information when they run, similar to the way interactive filters
work. To create an interactive macro, you can use the EditFind command and write the message such
that it prompts for the task or resource type you want to find.

For example, in the Search Notes macro, the following line is used to prompt the user for the
information they want to find in the notes:

```
EditFind .Field=Notes .Test=Contains .Value="Search for tasks with what value
```

in the Notes
field?"?
The syntax for the Value argument is exactly the same as what you would type in the Filter Definition
dialog box to create an interactive filter. Each time you run the macro, a dialog box is displayed that
says "Search for tasks with what value in the Notes field?" along with a box in which you type the text
you want to search for. Note that the .Next argument is not included so the notes in the selected task are
searched instead of searching forward or backwards.

You can also use the Message command with an If statement to allow the user to choose between two
options. An example of choosing between tasks or resources was included earlier in the description of
the Message command.

If you have a prompt inside a LOOP, you can use two question marks to have the question displayed
every time the loop is repeated. For example, the following statement:

Edit Find .Field=Duration .Test=Equals .Value="What value?"??

means that a dialog box would be displayed every time the loop is repeated asking for the value.
Otherwise, Microsoft Project asks for the value the first time through the loop only.

**Printing Macros**

To print the lines in your macro, copy the lines to a word processing application. Select all the lines in
the macro and then use the Copy button in the Macro Definition dialog box to copy the lines. Paste them
into a word processor and print. Since the Macro Definition dialog box uses the Windows Clipboard, you
can also write or edit macros in a word processor and then copy and paste the finished macro into the
dialog box.

**Keeping Track of Loops and Conditional Statements**

As you write more and more complex macros, you may find that you are having trouble keeping track of
your conditional statements and loops. If you don't include an ENDIF for each IF and an ENDLOOP for
each LOOP, a message will let you know you need to add these statements. One way to help you keep
track of the levels is to indent each time you have start an IF or LOOP block. The sample macros
included with Microsoft Project use this method. You could also include a comment each time you use
IF or LOOP. Include in the comment the number of the level. For example, the first IF you use would be
level 1. If you include a second IF within the first, this would be level 2. Before you

can end this macro,
you need two ENDIF statements, one for each level.

**Calling Other Macros**

You can call other macros from within a macro by using the Macro command. You may want to do this
if you find you are using a group of commands repeatedly. Rather than listing the commands over and
over, you can place those commands in their own macro and then call that macro each time you want to
use that set of commands. You might want to test for errors in submacros and if an error occurs, use Halt
to halt the macro. Otherwise, the macro may continue when you don't want it to.

Each macro you write is added as a command to the list of macro commands. For example, if you have
two macros, Combine Projects and Search Note, you'd see Macro [Combine Projects] and Macro [Search
Notes] in your list of macro commands. You can use these macros in any other macro by simply
including the command as a line in the macro you are writing.

**Determining the Active View**

If you are writing macros for others to use, your macros should include checks that the user is displaying
what you expect, such as the correct view being active. For example, what the macro does may depends
on either a task view or resource view being displayed, or maybe even a certain view being active. The
easiest way to do this is to start a macro by using the View command to display a certain view. You can
also check to see what type of view is displayed.

For example, to display the Task Sheet, use the following line in the macro:

```
View [Task Sheet]
```

If errors are off (Errors No), and there is no Task Sheet in the open view file, a default Task Sheet view
is displayed, which works like the Task Sheet.

To check to see if a task view is active and is the top view if you are using a combination view, you can
try a command that should work in this situation and see if it is successful. For example, since the All
Tasks filter is available only for task views in the top pane of a combination view, but not for resource
views or views in the bottom pane, you could use the following lines to see if the All Tasks filter works:

```
Error No
Filter [All Tasks]
If Error
commands [here you can change the view or use the Message command to
```

*tell the user to start*
*from another view and run the macro again and return]*
EndIf
If a task view is active and is in the top pane, no error will result from the Filter [All Tasks] line, Error
will return FALSE and following lines to ENDIF will be skipped. If Filter [All Tasks} does create an
error, which will happen if the active view is a resource view or the active view is the bottom of a
combination view, Filter [All Tasks] will return TRUE, and the steps following If Error will be carried
out. You can change views or warn the user that they need to do so.

You can check for a view by using any command that is available only for the view or type of view you
want. Since many of the commands on the Format menu are view specific, these are good commands to
use and then test for an error. For example, the FormatZoom command is available only on the PERT
Chart so you could use this command to check that the PERT Chart is the active view.

If you use different view files, you may also want to make sure the correct view file is open before
displaying a view. To do this, you use the ViewFileOpen command.

For example, the following lines open the view file MYVIEWS.MPV and then display the Resource
Graph:

```
ViewFileOpen .Name=[myview.mpv]
View [Resource Graph]
```

Whenever you use a procedure like this in a macro, make sure that it doesn't change anything
permanently for the user or affect any data.

**Checking or Replacing Values in the Project Fields**

Microsoft Project includes three commands to check or set fields.

**CheckField** - Checks the selected tasks or resources to see if they meet the criterion you specify. If all meet the criterion, the command returns TRUE. For example, the following line

```
CheckField .Field=[Milestone] .Value=[Yes] .Test=[Equals]
```

returns TRUE if all the selected tasks are milestones; FALSE if not all the selected tasks are milestones or if all selected rows are empty; or ERROR if it is a resource view.

**SetField** - Enters the value in a field for the selected tasks. For example, the following line

```
SetField .Field=[Marked] .Value=[No]
```

sets the Marked field to No.

**SetMatchingField** - Combines CheckField and SetField so you can check for values

and set values using one command. It sets a field only on those tasks or resources that match the criterion you specify.

**Setting Up a View**

While you can create views, add formatting, change the timescale, and so on from a macro, you may find it more efficient to create the view first using Define Views on the View menu, and then write the macro to display the new view. For example, if you want to display months on a Gantt Chart, use bold text, and change the gridlines, instead of using SendKeys to open the various dialog boxes and change the options, create a view called "Month Gantt" with these options already set. Since tables, filters, and everything set using commands on the Format menu are saved with the view, it is much easier to create the view first than to create in from within the macro and check for the errors that could occur along the way.

**Writing a Macro in a Word Processing Application**

Once you know how to write macros, you may find it much easier to write them in a word processing program. Just type in all the commands and arguments you want to use, one command per line. When the macro is as you want it, select all the lines and choose Copy. In Microsoft Project, choose Define Macros from the Macro menu, and then choose the New button. In the Macro Definition dialog box, choose the Paste button. All the text you copied from Word will be pasted into your macro.

Some word processing applications, including Microsoft Word for Windows and Microsoft Write, add line breaks where each line wrapped. You can either delete the line breaks after you paste the macro into the Macro Definition dialog box, or use lines long enough in the word processor that none of your lines wrap.

**Using AppExecute to Run a Macro in Another Application**

You can use the AppExecute command to start other applications or to run a macro created in another application. For example, the following line starts Microsoft Excel and runs the macro STOCK.XLM:

```
AppExecute .Command=[c:\excel\excel.exe c:\excel\stock.xlm]
```

For more information about using Microsoft Project macros with other applications, see the file DDEINFO.WRI.

**Using DDEExecute to Send Commands to Another Application**

You can use the DDEExecute command to send macro commands to other applications that are already running. For example, the following lines will open a DDE communications channel with Microsoft Excel, open the file SURVEY.XLS, and close the file:

```
AppExecute .Command=[c:\excel\excel.exe]
DDEInitiate .Application=[excel] .Topic=[system]
DDEExecute .Command=[[open("c:\excel\survey.xls")][close()]]
DDETerminate
```

Note that there are two sets of square brackets in the DDEExecute command. In Microsoft Project, you use brackets to enclose the commands being sent to the other application. Microsoft Excel also expects each command to be enclosed in square brackets. You can send multiple commands in one DDEExecute statement by enclosing each command statement in brackets and then enclosing all the

commands in another set of brackets. Different applications have different command formats: see your application manual to determine the syntax the application expects.

For more information about the DDE macro commands, see the file COMMANDS.WRI.

**File Handling**

If you are working with files in your macro, include the full path, if appropriate, so it will work if you change the current directory. If the macro will be used on other machines, however, you probably won't want to include the full path, but just the filename.

**Writing Macros for Others**

When you are writing macros that others will be using, think about including the following in the macros as appropriate:

- You may want to add a lot of explanatory messages. To break a message into lines, use **^n** where you want the line to break.
- Make sure you don't leave the project in a strange state, such as with all information selected, or a special filter applied. End the macro with something like SelectBeginning to select the first task and FilterAllTasks to display all information.
- If you are going to change the views, make a copy of the view before you start and change the copy.
- For errors, think about all possible cases that might cause problems. For example, if you use EditPaste, is there a chance that whatever is pasted will cause an error because there are more than 9999 tasks or resources? Or if you are using WindowNewWindow, if there are already 20 windows open, it will cause an error. Think of every possible condition for every command you use. Check what each command does in all views, top and bottom pane. Check what it does with resource views and task views; check what it does if nothing is selected, such as EditCut with nothing selected. When you figure out all possible situations, add errors and messages for each so those running the macro will understand why it didn't work. For example, look at the Combine Projects macro. This will work if the user has any number of projects open. Once it gets to the last one, it quits.
- If you want to act on all tasks, make sure that outlines are expanded and that FilterAllTasks is selected. If you want to act on all resources, make sure that FilterAllResources is selected.
- If it is important that a task is selected instead of a resource, check it. You can use something like FilterAllTasks to verify that it is a task view and then you know that any selection is a task.
- Don't change anything you don't have to change. What you change in the macro remains after the macro is finished. Since preferences are saved in workspace files, you might want to offer to save their current state in a workspace so that you can restore it at the end.

**International Notes**

If you are writing a macro that could be used with a different language version of Microsoft Project, think about the following as you write the macro:

- To use a macro in a different language version of Microsoft Project,  create the

macro using English for the macro commands, with the argument values in the language of the product. When you open the macro in the other language version, replace the argument value with the new product language.

- For argument values requiring a Yes or No, you can use numbers to avoid replacing the argument value in the new language version. For argument values that are Yes or No, you can type 1 for Yes and 0 for No. For example, if you want to turn off Gantt Bar Rounding, use 0 as the argument value rather than No.
- If you are using SendKeys to change options in a dialog box, use the arrow keys and TAB to move around instead of ALT+underlined character. The underlined character may change depending on the language for the product so it is safer to use the arrow keys or TAB to move to the appropriate option.
- Be aware of potential product differences in the Preferences dialog box. For example, if a user in Europe has semicolon as the list separator character, you want to make sure that things with a semicolon in them are handled correctly. For this reason, it is safest to use spaces between argument names instead of the list separator character.
- If you depend on a certain list separator character, date format, cost format, and so on, set it at the beginning of the macro.
- When you do things like set views, tables, or filters, remember you are referring to a specific name, such as Gantt Chart. Because you may not know what the name will be in another language, you may want to open a view file first so you can control the available views. If the specifics in a view are not important, but just that a certain type of view is available, such as a task view, use the ideas earlier for determining the active view.

## Quick Reference to Macro Commands

The following list summarizes the commands used in macros. If you want a printed list of all the commands and their syntax, print the file COMMANDS.WRI. The information in COMMANDS.WRI is the same as that in Commands Used In Macros in online Help.

**DDE Commands**

**DDEExecute** - Sends a command to another application.
**DDEInitiate** - Starts a DDE conversation with another application.
**DDETerminate** - Ends the active DDE conversation.
**UpdateDDELinks** - Updates all links.

**Edit Menu Commands**

**EditAssignment** - Adds, replaces, or removes resource assignments.
**EditClear** - Clears the information in the selected fields.
**EditCopy** - Copies the selected information and stores it on the Clipboard.
**EditCopyPicture** - Copies the view as an object or the selected information as a static picture.
**EditCut** - Deletes the selected information and places it on the Clipboard.
**EditDelete** - Deletes the selected information or object.
**EditFillDown** - Copies the information in the top selected field to the remaining selected fields.
**EditFind** - Finds the next task or resource that meets the criterion you specify.
**EditForm** - Displays the Task Edit Form or Resource Edit Form, depending on whether a task or resource is selected.
**EditGoto** - Moves to the ID number or date you specify.

**EditInsert** - Inserts a blank row into a table or a node into the PERT Chart. If a column is selected, inserts a new column.
**EditInsertObject** - Creates an object in the application you specify and attaches it to the selected task or resource.
**EditLinkTasks** - Links the selected tasks with finish-to-start relationships.
**EditObject** - Edits the selected object.
**EditPaste** - Inserts the information on the Clipboard into the project.
**EditPasteLink** - Inserts the information on the Clipboard such that it is linked to the original information.
**EditUndo** - Reverses the most recent command.
**EditUnlinkTasks** - Unlinks the selected tasks.
**FindNext** - Finds the next task or resource that contains information meeting the test and value specified in the Find dialog box.
**FindPrevious** - Finds the previous task or resource that contains information meeting the test and value specified in the Find dialog box.

**File Menu Commands**
**FileClose** - Closes the active project.
**FileCloseAll** - Closes all open projects.
**FileLinks** - Displays the Links dialog box where you can create, update, and delete links, and open linked files and applications.
**FileLoadLast** - Opens one of the files listed at the bottom of the File menu.
**FileNew** - Creates a new project.
**FileOpen** - Opens an existing project.
**FilePageSetup** - Specifies page formatting including headers, footers, margins, and legends.
**FilePrint** - Prints the active view.
**FilePrintPreview** - Displays a view as it will appear when printed.
**FilePrintPreviewReport** - Displays a report as it will appear when printed.
**FilePrintReport** - Prints the report you specify.
**FilePrintSetup** - Lists printers and options for the selected printer.
**FileResources** - Specifies whether the project should use its own resources or the resources stored in another project.
**FileSave** - Saves the active project on the disk.
**FileSaveAs** - Names or renames an existing project and saves it on the disk.
**FileSaveWorkspace** - Saves a list of the open files and the preferences in the Preferences dialog box as a workspace file on the disk.

**Filters**
**Filter** - Applies an existing filter.
**FilterDefineFilters** - Displays the Define Filters dialog box so you can apply, create, copy, or edit a filter.

**Format menu commands**
**FormatAvailability** - Displays the amount of work for which a resource is available on the Resource Graph or Resource Usage view.
**FormatBorders** - Displays the Borders dialog box so you can change the borders on the PERT nodes.
**FormatCost** - Shows cost information for resources on the Resource Form, Resource Graph, or Resource Usage view. .
**FormatCumulativeCost** - Displays the cumulative cost for resources on the Resource Graph or Resource Usage view.
**FormatCumulativeWork** - Displays the cumulative work for resources on the

Resource Graph or Resource Usage view. .
**FormatGridlines** - Displays the Gridlines dialog box so you can change the gridlines in the active view.
**FormatLayout** - Displays the Layout dialog box so you can change the lines, arrows, and page-break sensitivity in the PERT Chart.
**FormatLayoutNow** - Arranges the PERT nodes.
**FormatNotes** - Displays the Notes box at the bottom of the Resource Form or Task Form. .
**FormatObjects** - Displays the Objects box at the bottom of the Resource Form or Task Form. .
**FormatOutline** - Displays the Outline dialog box so you can change the outline format in the active view.
**FormatOverallocation** - Displays the amount of work for which a resource is allocated to work over capacity on the Resource Graph or Resource Usage view.
**FormatPageBreaks** - Displays the page breaks on the PERT Chart.
**FormatPalette** - Displays the Palette dialog box so you can change the format, placement, alignment, and color of information on the Gantt Chart, PERT Chart, or Resource Graph. .
**FormatPeakUnits** - Displays the peak units for resources during the time period on the Resource Graph or Resource Usage view.
**FormatPercentAllocation** - Displays the percentage that a resource is allocated for the time period on the Resource Graph or Resource Usage view.
**FormatPredecessorsSuccessors** - Displays the predecessors and successors fields at the bottom of the Task Form.
**FormatRemoveAllPageBreaks** - Removes all manual page breaks in the project.
**FormatRemovePageBreak** - Removes the manual page breaks above the selected task or resource.
**FormatResourceCost** - Displays the resource cost fields at the bottom of the Task Form.
**FormatResourceSchedule -** Displays the resource schedule fields at the bottom of the Task Form.
**FormatResourcesPredecessors** - Displays the resources and predecessors fields at the bottom of the Task Form.
**FormatResourcesSuccessors** - Displays the resources and successors fields at the bottom of the Task Form.
**FormatResourceWork** - Displays the resource work fields at the bottom of the Task Form.
**FormatSchedule** - Displays the schedule fields at the bottom of the Resource Form.
**FormatSelectedTasks** - When the Resource Usage view is on the bottom of a combination view, displays usage information for just the tasks selected in the top view.
**FormatSetPageBreak** - Inserts a manual page break above the selected task or resource.
**FormatSort** - Displays the Sort dialog box so you can sort the tasks or resources in the active view.
**FormatText** - Displays the Text dialog box so you can change the text in the active view.
**FormatTimescale** - Displays the Timescale dialog box so you can change the timescale in the active view.
**FormatWork** - Displays the work fields at the bottom of the Resource Form.
**FormatZoom** - Zooms the PERT Chart in or out.

**Help**

**HelpIndex** - Displays the Help index.
**HelpPlanningWizards** - Starts PlanningWizards.
**HelpTutorial** - Starts the Tutorial.

**Macros**

**Macro** - Runs an existing macro.
**MacroDefineMacros** - Displays the Define Macros dialog box.

**Miscellaneouss commands**

**Alerts** - Turns Microsoft Project messages that ask a question on or off.
**CheckField** - Checks the selected tasks or resources to see if they all meet the criterion you specify.
**CreateMSGraph** - Displays the Create Graph Object dialog box.
**Error** - Tells the macro how to handle an error message.
**Form** - Displays an existing form.
**GotoNextOverAllocation** - Moves to the next overallocation on a view with a timescale.
**GotoTaskDates** - Scrolls the timescale to display the selected task.
**Message** - Displays a message.
**SendKeys** - Specifies keys to send to Microsoft Project.
**SetField** - Enters a value in the field you specify.
**SetMatchingField** - Equivalent to the combination of filtering the tasks or resources, selecting the filtered list, and then setting a field in these tasks or resources to a certain value.

**Options menu commands**

**OptionsBaseCalendars** - Displays the Base Calendars dialog box so you can change the working days and hours in the base calendar you select.
**OptionsCalculateNow** - Calculates the open projects.
**OptionsCalculateProject** - Calculates the active project.
**OptionsCalculation** - Specifies automatic or manual calculation of the schedule.
**OptionsCustomForms** - Displays the Custom Forms dialog box.
**OptionsLeveling** - Specifies automatic or manual leveling and other leveling options.
**OptionsLevelNow** - Levels resources to resolve resource conflicts.
**OptionsPreferences** - Sets the preferences in the Preferences dialog box.
**OptionsProjectInfo** - Specifies general information about the project.
**OptionsProjectStatus** - Displays the scheduled, planned, and actual start and finish dates; the duration, work, and cost totals; and the duration and work percent complete for the project.
**OptionsRemoveDelay** - Removes the delay from all tasks or selected tasks.
**OptionsResourceCalendars** - Displays the Resource Calendars dialog box so you can change the working days and hours in a resource calendar you select.
**OptionsSetActual** - Records progress on specified tasks.
**OptionsSetPlan** - Creates a baseline plan using current dates for selected tasks or all tasks.
**OptionsSpelling** - Starts the spelling checker.
**OptionsSpellingOptions** - Displays the Spelling Options dialog box so you can change the options used when checking spelling.

**Outlining**

**OutlineCollapse** - Collapses the selected summary tasks.

**OutlineDemote** - Demotes the selected tasks.
**OutlineExpand** - Expands the selected summary tasks.
**OutlineExpandAll** - Expands all the summary tasks.
**OutlinePromote** - Promotes the selected summary tasks.

**Selecting Fields**
**ExtendSelection** - Extends the selection from the current location to the new location.
**SelectAll** - Selects all the tasks or resources in the project.
**SelectBeginning** - Moves to the first unlocked column in the first row of the view.
**SelectCellDown** - Selects a field below the active field.
**SelectCellLeft** - Selects a field to the left of the active field.
**SelectCellRight** - Selects a field to the right of the active field.
**SelectCellUp** - Selects the field above the active field.
**SelectColumn** - Selects the column containing the active field.
**SelectEnd** - Moves to the last field in the last row that contains information.
**SelectRow** - Selects the row containing the active field.
**SelectRowEnd** - Moves to the last field in the current row.
**SelectRowStart** - Moves to the first field in the current row.

**Tables**
**ColumnBestFit** - Changes the width of the column so it best fits the information in the column.
**ColumnEdit** - Displays the Column Definition dialog box.
**Table** - Applies an existing table.
**TableDefineTables** - Displays the Define Tables dialog box.

**Timescale commands**
**TimescaleZoomIn** - Shows more detail by displaying a smaller period of time on the timescale.
**TimescaleZoomOut** - Shows less detail by displaying a greater period of time on the timescale.

**Views and View Files**
**View** - Displays an existing view.
**ViewDefineViews** - Displays the Define Views dialog box.
**ViewFileOpen** - Opens a new view file.
**ViewFileSave** - Saves a view file with the same name.
**ViewFileSaveAs** - Saves a view file with the name you specify.

**Working with Windows**
**AppExecute** - Specifies the application window and the pane that you want to be active.
**AppMaximize** - Enlarges the Microsoft Project window.
**AppMinimize** - Shrinks the Microsoft Project window to an icon.
**AppMove –** Moves the Microsoft Project window.
**AppRestore** - Restores the Microsoft Project window to its previous size and location.
**AppSize** - Changes the size of the Microsoft Project window.
**DocClose** - Closes the active window.
**DocMaximize** - Enlarges the active project to fit the Microsoft Project window.
**DocMove** - Moves the project window. If you don't include the arguments, the position is unchanged.

**DocRestore** - Restores the project window to its previous size and location.
**DocSize** - Changes the size of the project window.
**PaneClose** - Closes the bottom pane in a combination view.
**PaneCreate** - Splits a single-pane window; places the Task Form in the bottom view if the original view was a task view or the Resource Form in the bottom view if the original view was a resource view.
**PaneNext** - Moves to the next pane.
**WindowActivate** - Specifies the window and the pane you want to be active.
**WindowArrangeAll** - Resizes and rearranges the open windows so all are visible.
**WindowHide** - Hides the active window.
**WindowMoreWindows** - Displays the Window Activate dialog box so you can select the window you want to be the active window.
**WindowNewWindow** - Opens another window on the active project or combines multiple projects in one window.
**WindowNext** - Moves to the next window.
**WindowPrev** - Moves to the previous window.
**WindowUnhide** - Makes a hidden window visible.