Autodesk® PowerShape®

Using Macros with PowerShape

AUTODESK

Contents

Customizing PowerShape   4

Autodesk Legal Notice   233

Customizing PowerShape

You can customize the behaviour of PowerShape by creating:

- Macros (see page 5)
- OLE applications (see page 130), which can be either:
  - HTML-based
  - Add-in applications

Both macros and add-in applications use special object information (see page 143) macros.

These features enable you to tailor PowerShape to your needs.

Macros

A macro file contains commands and comments. The main use of a macro file is to store frequently-used or complex sequences of commands for repeated use.

When you have mastered writing macro files, you can tailor PowerShape for your own use, and greatly enhance its power and flexibility. For example, you may need to create a number of standard mold parts, such as nuts and bolts in your model. You can write a macro to create nuts and bolts of any size and at any position. So, any time you wish to add a nut, you just run the macro and define the size of the nut and its position.

In the example macros, it is important to remember the following:

- *Commands are not case-sensitive, so `if` and `IF` are interchangeable.*

- *Any blank lines start with // or $$ to indicate a comment line. Any blank lines in the following examples are there to improve readability.*

Creating macros

Recording macros

An easy way to create a macro file is to record the commands as you are working. When you record macros, you create a set of commands that are carried out in the order you record them.

To record a macro:

1 Select Home tab > Macro panel > Record to display the **Record Macro** dialog:

2 Select the location you want to save the macro to using the **Save in** drop-down list.

3 In the **File name** box, enter the name of the file you want to record to. If you enter the name of an existing file, it is overwritten with the new commands.

4 Click **Save** to begin recording the macro.

5 Perform the actions you want to record.

6 Select Home tab > Macro panel > Record to stop recording. You can use any text editor to view and edit a macro.

If you record a macro of a Paint Triangles or Mesh command, an extra macro file is created. This file is named with the following convention:

```
<psmacroname>_cc_<nnnn>.mac
```

where

- `<psmacroname>` is the name of macro being recorded.

- `<nnnn>` is a four digit number. This number is incremented by one each time an embedded mode is used.

For further information see the Macro tutorial (see page 52).

Writing macros

Written macros can be longer, and more elaborate than those you record. For example, you can add comments or add testing conditions.

Use any text editor to create or edit your own macro files:

1 Type your own macro commands into the text editor.

2 Save the file in .mac format.

    You can then run the macro (see page 9).

When you interact with the PowerShape interface, commands are sent to the program. These are the commands that you must write into your macro file if you want to drive PowerShape using a macro.

To find macro commands used by PowerShape:

1 Record a macro (see page 6) of the operations you want to find the commands for. This records the operations as command lines.

2 Open the macro file using a text editor.

3 Copy the commands into your macro.

The following table lists examples of commands you can add to macros:

`$$` `//`	add comments to remind you what each part of the macro does
`input`	allow users to input information whilst the macro is running
`print`	output information from the macro

`let`
`int`
`string`
`real`
	store information in variables

`let$e=5+(6*)`	build up expressions (for example, 5+(6*2)) and assign their values to variables

`if` `switch`	decide which commands are carried out next depending on the value of a variable
`while`	repeat a set of commands a number of times
`goto` `label`	jump from one command line to another
`macro run`	run one macro from within another and pass information to a macro
`export`	export variables from a macro
`execute step` `execute run`	step a block of commands in a macro while it is running
`execute` `command $var`	run the command indicated by the variable (see page 93)
`skip`	skip a block of commands
`input free` `execute pause`	pause a running macro
`return`	end a macro

Adding comments in macros

It is good practice to put comments into a macro file to explain what it does. A comment is a line of text which has no effect on the running of the macro file, but which helps other users of the macro to understand what it does. Comment lines start with `//` or `$$`. For example:

```
// This macro file deletes any coincident
// Pcurves from a surface.
```

It is also good practice to have comments explaining what each section of the macro file does.

A `$$` comment can be added only at the beginning of a line. You cannot put a `$$` comment on the same line as a command (except after a label). For example, the following string is NOT allowed:

```
LET $a = ($b*9/360) $$ This calculates the angle
```

However you can use this syntax when using the `//` comment. For example, this is allowed:

```
LET $a = ($b*9/360) // This calculates the angle
```

We suggest that you put the comments to describe commands and then the commands. For example:

```
// Calculating the angle
```

```
LET $a = ($b*9/360)
```

Another use of comments is to temporarily remove a command from a macro. To do this, insert `$$` or **//** at the beginning of the line which contains the command you want to remove. For example:

```
// LET $a = ($b*9/360)
// PRINT $a
```

This is known as *commenting out* a command.

Running macros

To run a previously recorded macro:

1 Select Home tab > Macro panel > Macro > Run to display the **Run Macro** dialog.

2 Select the macro you want to run.

3 Click **Open** to run the macro.

If you want to stop the macro while it is running, press the **Esc** key. The macro finishes the command it is currently processing and stops.

In addition to the Run option, the Step option runs one command in the macro at a time. This enables you to watch the macro in detail and check that each section is working correctly.

To step through a macro:

1 Select Home tab > Macro panel > Macro > Step. The **Step Through Macro** dialog is displayed.

2 Select the macro you want to run.

3 Click **Open** to start the macro.

   The Command window is displayed, showing the first command in the macro, for example:

   ```
   Macro 1: Line 1: command in first line>
   ```

4 Select the Command window, and press the **Enter** key to run the command.

   The next command is displayed:

   ```
   Macro 1: Line 2: command in second line >
   ```

5 Continue pressing the **Enter** key to run each command in the macro.

To stop a macro at any point, select Home tab > Macro panel > Macro > Abandon.

Creating user-input information into a macro

Most macros require some user interaction. For example, asking the user to enter the position or the dimensions of the object. The information supplied by the user interaction is stored in a variable within the macro.

There are two ways to enter values into a macro variable:

- Set all the variables in a macro file before running it.

  This requires you to edit the macro every time you want to change the variables. This can make it difficult for anyone other than the originator to use it.

- Prompt the user for values when the macro is running.

  This is a neater method and you may also input values as part of the macro's initiation command.

  To prompt a user for information, use the **INPUT** command. For each command, you can ask for:

  - a point (see page 10)

  - a selection of items (see page 11)

  - a number (see page 12)

  - a string (see page 13)

  - a yes or no response to a query (see page 13)

Point information

If you want the user to enter a point, use the command:

```
INPUT POINT 'string' $variable_name
```

This command displays a dialog.

The characters in the string are displayed on the dialog and the X, Y, Z coordinates of the point entered are assigned to three variables:

- *variable_name_x,*

- *variable_name_y*

- *variable_name_z.*

For example:

```
INPUT POINT 'Enter a point' $centre_pos
```

This displays the following dialog when the macro is run.
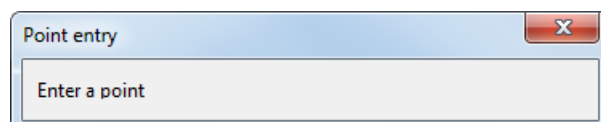
The user enters a point by:

- clicking on the screen
- entering values into the status bar
- using the **Position** dialog.

The values of X, Y and Z are then assigned to variables:

- *centre_pos_x*
- *centre_pos_y*
- *centre_pos_z*

Print out the X, Y, Z values of the point you entered using the following:

- print $variable_name_x
- print $variable_name_y
- print $variable_name_z

To print out the X value of the point entered above, use **print $centre_pos_x**

The value of X will be printed in the Command window. To find the values for Y and Z, substitute y or z for x.

Selection information

If you want the user to select one or more objects for use in the macro, use the command:
```
INPUT SELECTION 'string'
```

This command displays a dialog, which shows the number of objects selected. The characters in the string are used for the title on the dialog. When objects are selected, the number of objects selected is shown in the dialog.

No objects must be selected before using the input selection command.

For example:
```
INPUT SELECTION 'Select items'
```

When a macro containing this command is run, the following dialog is displayed:
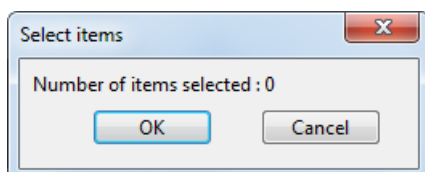
When items are selected, the dialog shows the number of objects that are selected.

When you click **OK**, the macro can use `selection` object information to display the number of selected objects. For example:

```
print selection.number
```

prints the number of objects selected.

```
print selection.object[0]
```

prints the type and name of the first object in the selection.

You can use the selection object information (see page 143) to check that the correct number or types of objects are selected.

Example - Select a line and check selection:

This example asks the user to select a line. The macro then checks that a single line is selected. If a single line is not selected, an error message is displayed.

```
LET $no_line = 1
WHILE $no_line {
select clearlist
INPUT SELECTION 'select a line'
IF (selection.number == 1) {
LET $no_line = !(selection.type[0] == 'Line')
}
IF $no_line {
PRINT ERROR 'You must select a single line'
}
}
```

For further information see:

- `IF` (see page 34)

- `WHILE` loop (see page 39)

Number information

Use this command to ask the user to enter a number.

```
INPUT NUMBER 'string' $variable_name
```

This command displays a dialog where:

- *'string'* is used as the dialog title.

- *variable_name* is the label of the text box.

For example:

```
INPUT NUMBER 'Input radius of arc 1' $Radius1
```

When the macro is run, the following dialog is displayed:
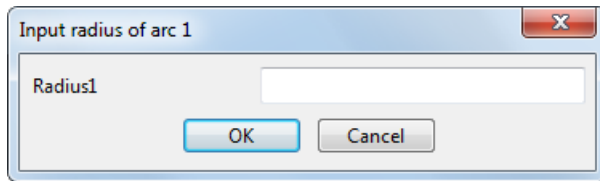
Enter a value and click **OK**. The value is assigned to variable *Radius1*.

String information

Use the following command to request a text string:

```
INPUT TEXT 'string' $variable_name
```

Like `INPUT NUMBER`, this command displays a dialog where:

- the *'string'* characters are used for the dialog title.
- *variable_name* is the label of the text box.

For example:

```
INPUT TEXT 'Reverse the surface? Y/N' $Answer
```

When the macro is run, the following dialog is displayed:
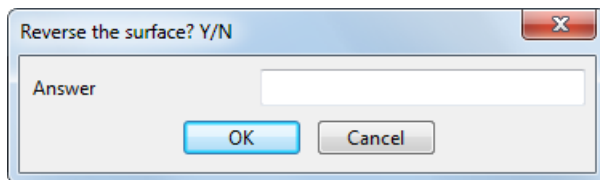
Enter a value and click **OK**. The value is assigned to variable *Answer*.

Query information

If you want to ask a question that requires a yes or no answer, use:

```
INPUT QUERY 'string' $variable_name
```

This command displays a dialog with **Yes** and **No** buttons. The question you want to ask is contained in the string. If the user selects **Yes**, then `$variable_name` becomes *1*, otherwise it becomes *0*.

For example:

```
INPUT QUERY 'Do you want to exit the macro?' $prompt
```

When the macro is run, the following dialog is displayed:
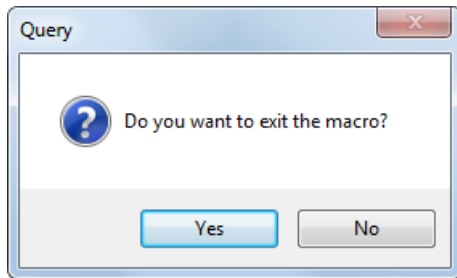
- If you click **Yes**, the variable `$prompt` becomes *1*.
- If you click **No**, the variable becomes *0*.

Entering values during macro initiation

A user may initiate a macro so that the information required within the macro is also given. For example:

```
macro run name_of_file.mac var1 var2 ... varN
```

where *var1*, *var2*, … ,*varN* are values of variables used in the macro.

If the name of a macro file contains spaces, the name must be included in double quotes. For example:

```
macro run "name of file.mac" 1 2.4
```

To import variables, you must declare them at the start of the macro using the following syntax.

```
ARGS{
TYPE variable1
TYPE variable2
.
.
.
TYPE variableN
}
Rest of macro
```

where `TYPE` is one of *INT*, *REAL*, or *STRING*.

To display the Command window, double-click the Command box in the status bar.

For example:

To run macro *test.mac* with values *1*, variable *$two* and string *'three'*, type the following in the Command window:

```
macro run test.mac 1 $two 'three'
```

In the macro, these values are defined as variables with their types at the start as:

```
ARGS{
Int variable1
Real variable2
String variable3
}
Rest of macro
```

So, in the following macro you must enter values that match the variable types.

```
ARGS{
Int i
Real j
String k
}
print $i
print $j
print $k
```

Start the macro using the following command:

```
macro run macro1.mac 34 78.7 'mouse'
```

It will print out

34

78.7

mouse

> *ARG{* and *ARG {* are both valid formats. Comments can appear at the start of a macro with arguments.

Outputting information from a macro

Displaying information

To display a message that does not require any information from the user, use PRINT command.

```
PRINT 'Type your message here'
```

For example:

If a user provides an incorrect response, a macro displays an error message and prompts for another response:

```
PRINT '***Invalid response. Please try again.***'
```

You can also display error message dialogs when an invalid answer has been given, using:

```
PRINT ERROR '***Invalid response. Please try again.***'
```

This displays the following error on the screen.
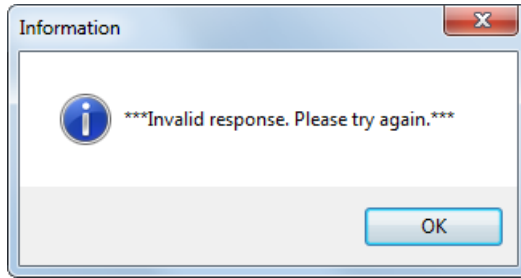
To close the message, click **OK**.

> *Messages can be displayed in the Command window or in dialogs. To open the Command window, double-click the Command box in the status bar.*

Output from the `PRINT` commands can also be sent to an `OUTFILE`.

To produce an outfile:

1   Open an OUTFILE. This can have a predefined name, or you can use a name that is entered at run time.

Use one the of the following methods:

- Open an OUTFILE with a given name

```
let filename = 'e:/homes/fred/report.txt'

FILE OUTFILE OPEN REPLACE $filename
```

You must give an absolute pathname to the file.

`REPLACE` gives permission to overwrite any existing file. If the file exists and `REPLACE` is omitted then you will be asked to confirm that the file can be overwritten.

- Open an OUTFILE with a name obtained from the user

```
FILE OUTFILE OPEN DIALOG

TITLE Create a report file

FILETYPES TXT File (.txt)|*.txt|txf

RAISE
```

The `TITLE` and `FILETYPES` are optional. The `FILETYPES` string consists of:

```
File type name | Regular expression | Default file extension
```

Example — to prompt the user to create an HTML file:

```
FILETYPES HTML File (.html) | *.html | html
```

2   Generate your report using the PRINT command.

```
PRINT ...
PRINT 'This file is ' outfile.name '.'
PRINT 'Report generated on ' date ' by ' user.name '.'
PRINT ...
```

3 Close the OUTFILE:

```
FILE OUTFILE CLOSE
```

4 Display the file in the browser:

```
BROWSER SHOW
BROWSER GO $filename
```

The filename must start with a drive letter.

Displaying values of variables

Use the `PRINT` command to display the values of variables. For example:

```
PRINT 'Lateral ' $lat_no ' does not exist.'
```

You may need to add spaces in strings to separate items in a **print** command.

For example:

```
PRINT 'Lateral ' $lat_no ' does not exist.'
```

displays
```
Lateral 5 does not exist.
PRINT 'Lateral' $lat_no 'does not exist.'
```

displays
```
Lateral5does not exist.
```

The PRINT command works for expressions that evaluate strings, vectors and lists.

For example:

```
print concatenate('abc'; 'def')
```
prints the string
```
abcdef
```

```
print cross([1; 2; 3]
```
prints the resulting vector
```
[40; -50.5; 76.23]
```

```
print atan2(-30; 40)
```
prints the arctangent

Example macro to generate and display a report file

```
args{
  string filename
}
// report_example.mac
//
// An example of how a macro can generate and display a
report file.
// ---------------------------------
//
// Open an html outfile to hold the report.
let use_dialog = $filename == 'dialog'
if $use_dialog {
  file outfile open Dialog
  Title Create a graphics report file
  FileTypes HTML File (.html)|*.html|html
  Raise
} else {
  // This must be an absolute filename.
  file outfile open replace $filename
}
//
// -------------------------------------
// Print the report.
print '<html>'
print '<head>'
print '<title> Example of a Report File Generated by a
Macro</title>'
print '</head>'
print '<body bgcolor="#CCCC66">'
//
print '<h1> Example of a Report File Generated by a
Macro</h1>'
//
print 'This HTML file was generated and displayed in the
browser window'
print 'by a macro. It shows how'
print 'information about the graphics system can be
generated and'
print 'displayed.<p>'
//
print '<p>'
//
// The values of some graphics properties:
print 'Display lists are ' graphics.displaylists '.<br>'
print 'Vertical sync is ' graphics.verticalsync '.<br>'
print 'OpenGL version is ' graphics.openglversion '.<br>'
//
let red_bits = graphics.intparam.RED_BITS
```

```
let green_bits = graphics.intparam.GREEN_BITS
let blue_bits = graphics.intparam.BLUE_BITS
//
let colour_depth = $red_bits + $green_bits + $blue_bits
//
print 'Colour depth is ' $colour_depth '.<br>'
print 'Z-buffer depth is ' graphics.intparam.DEPTH_BITS
'.<br>'
//
print 'Window size is ' window[1].size.x ' by '
window[1].size.y ' pixels.<p>'
//
print 'OpenGL extensions supported are: <br><pre>'
//
graphics printextensions
//
print '</pre>'
//
// How to use the timer:
print 'Total test time is ' timer ' seconds.<br>'
//
print 'Test run by ' user.name ' on ' date '.<p>'
//
// print 'Mailto <a
href="mailto:someone@autodesk.com">someone@autodesk.com</
a><p>'
//
let filename = outfile.name
print 'This file is ' $filename '.<br>'
//
print '</body>'
print '</html>'
//
// -----------------------------------
file outfile close
//
browser show
browser go $filename
```

Using variables in macros

A variable enables you to store information for later use. You can define a variable using a name, for example *centre*. When you use variables in an expression, you must add a $ to their name, for example:

```
LET $a = ($centre + 1)
```

A variable name cannot be a valid macro command, for example, you *cannot* use `$PRINT`, where `PRINT` is a macro command.

You can also assign values to variables (see page 23). For example, the following line defines variable *bolts* with type *integer* and assigns it a value of 5:

```
INT $bolts = 5
```

You can also assign values to variables by performing complex calculations.

Variable types

A variable type specifies the kind of information that can be stored by the variable. A variable can have only one type. The type must be specified when you define it.

You can specify the following variable types:

- INT — integer numbers for example 1, 21, 5008
- REAL — real numbers for example 20.1, -70.5, 66.0
- STRING — for example *'hello'*
- VECTOR
- LIST
- ERROR

Determining the type of a variable or expression

Use the following command to determine the type of an expression or variable:

```
TYPE(...)
```

The command returns a string that is:

```
INT
REAL
STRING
VECTOR
LIST
ERROR
```

For example:

```
LET my_var = 17
PRINT TYPE($my_var)
```
// this prints INT

```
LET a = 12.345
PRINT TYPE($a)
```
// this prints REAL

```
PRINT TYPE('hello')
```
// this prints STRING

```
PRINT TYPE([1; 2; 3])
```
// this prints VECTOR

```
PRINT TYPE({'a'; 'b'; 'c'; 'd'})
```
// this prints LIST

```
PRINT TYPE(SQRT(-57))
```
// this prints ERROR, because you are trying to take square root of a negative number.

Converting variable types

You can use the following macro commands to convert a variable type to another variable type:

```
INTTOREAL
INTTOSTRING
REALTOINT
REALTOSTRING
STRINGTOINT
STRINGTOREAL
```

The following example fills variable *s* with value *10*:

```
INT frame_number = 10
string s = INTTOSTRING($frame_number)
```

These macro commands cannot be used with print commands.

Renaming objects using variables

When an edit dialog is displayed for an object, the `VAR_NAME` and `NAME` commands enable you to rename the object using a variable.

Use `VAR_NAME` and `NAME` to rename the following:

- lines
- chamfers
- arcs
- curves
- composite curves
- points
- primitive surfaces
- general surfaces
- primitive solids
- general solids
- workplanes

Example - Using VAR_NAME to change the name of an arc

The following uses the variable `$n` to name an arc **'joe'** when the **Arc** edit dialog is displayed.

```
let $n= 'joe'
create arc full
0 0 0
select
modify
VAR_NAME $n
accept
```

initialises the *$n* variable to '*joe*'

names the arc *'joe'*

Using environment variables

Environment variables are different from other variables. They can be written at one macro level and read at a lower macro level.

You can use environment variables in the following ways:

- Set an environment variable using the **setenv** variable.

```
let path = 'e:/tmp'
setenv path
macro run print_path.mac
```

The macro *print_path.mac* has access to a copy of the variable `path`**.** The macro called *print_path.mac* contains the following:

```
print 'path = ' $path
```

- Print the contents of the environment using the **printenv** variable.

```
select > printenv
path=e:/tmp
```

- Remove a variable from the environment using the `unsetenv` variable.

```
unsetenv path
```

- Export a variable (see page 48) into the environment of the calling macro using the `exportenv` variable

```
exportenv path
```

This allows a called macro to set up the environment for a number of other macros.

Lower level macros can access a copy of the environment variables. These macros can change the contents of the variables, but those changes are discarded when the macro returns its value.

Assigning values to variables

Values are assigned to variables using the following syntax:

LET $variable = expression

The `$` in front of the variable is optional.

You can:

- Assign constant values to variables.

```
LET $new_variable = 45
```

- Use expressions to assign values to variables.
```
LET new_variable = 45/36
```

- You may also use existing variables to assign values to variables.

```
LET new_variable = $existing_variable/36
```

- You can use a variable to define a new value to itself. For example,

```
LET $a = $a +1
```

This means add one to variable **a**.

- You can access individual characters of string variables and expressions.

```
LET my_str = 'model'
```

```
// Print the first character 'm'
```

```
Print (%my_str[1])
```

- You can get a sub-range of a string or list variable using the command:

```
RANGE(<arg1>; <arg2>; <arg3>)
```

Where:

- `<arg1>` is a string or list.

- `<arg2>` is an integer specifying the start index (index starts at 1).

- `<arg3>` is an integer specifying the number of characters or list elements to return.

- You can remove assigned values by using the command:

```
LET $a = null
```

If you are carrying out a command that you are certain does not expect a number, you can use:

```
TYPE $variable = expression
```

where `type` is **INT**, **REAL**, or **STRING**

You can also use:

```
$variable = expression
```

For example, you must use `LET` in the following:

```
create line
LET start_x = 10
LET start_y = 20
LET start_z = -50
$start_x $start_y $start_z
LET end_x = 20
LET end_y = 30
LET end_z = 50
$end_x $end_y $end_z
```

For further details, see:

- Using expressions in macros (see page 28)

Using object information

You can assign object information (see page 143) to a macro variable, for example, at the start point of a line. Object information is accessed using syntax containing specific details of an object. The syntax is typically:

```
object type [object name] sub-object names
```

For example, if you have a line with the name **2**, then all the information about line is available by referring to `line[2]`.

The start coordinates of line 2 are accessed as follows:

```
line[2].start   retrieves the start coordinates [x, y, z]
of line 2.
line[2].start.x retrieves the X coordinate of the start of
line 2.
line[2].start.y retrieves the Y coordinate of the start of
line 2.
line[2].start.z retrieves the Z coordinate of the start of
line 2.
```

Use this object information to assign values to variables.

Example: Create a full arc with its centre point at the start coordinates of line 2

```
LET $a = line[2].start.x
LET $b = line[2].start.y
LET $c = line[2].start.z
CREATE ARC
FULL
$a $b $c
```

Assigning an object to a variable

Use the following syntax to assign an object to a variable.

```
LET $t = Line[2]
```

This variable can be used to access information about the object. The following is the *x* coordinate of the start point of *Line[2]*.

```
$t.start.x
```

Comparing variables

Comparing variables lets you check information. They also allow you to decide the course of action to take in **if** and **while** commands. For further details, see:

- Making decisions in macros (see page 33)
- Repeating commands in macros (see page 39)

A result of a comparison is either *true* or *false*. When it is true, a value of **1** is output and when false, **0** is output.

A simple comparison may consist of two variables with one of the following operators between them:

==	is equal to
!=	is not equal to
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to

For example:

```
LET $C = ($A == $B)
```

`C` is true if `A` equals `B` and is assigned `1`. If `A` doesn't equal `B`, then `C` is false and assigned `0`.

The variables = and == are different. The single equal sign = means to assign a value, whereas the double equals sign == means compare two values for equality.

If you compare the type of an object with a text string, you must use the correct capitalisation. For example, if you want to check that selection.type[0] is a composite curve, you must use:

```
selection.type[0] == 'Composite Curve'
```

and not:

```
selection.type[0] == 'Composite curve'
selection.type[0] == 'composite curve'
```

For example:

```
LET $e = (($a+$b) >= ($c+$d))
```

If you are carrying out a command that you are certain does not expect a number, you can use:

```
TYPE $variable = expression
```

where `type` is **INT, REAL,** or **STRING**

You can also use:

```
$variable = expression
```

For example, you must use `LET` in the following:

```
create line
LET start_x = 10
LET start_y = 20
LET start_z = -50
$start_x $start_y $start_z
LET end_x = 20
LET end_y = 30
LET end_z = 50
$end_x $end_y $end_z
```

If in doubt, include the `LET`.

Logical operators let you do more than one comparison at a time. Logical operators are:

- AND

- OR

- NOT

Remember that **true = 1** *and* **false =0**

AND (&)

This outputs **1** if both inputs are 1.

```
0 & 0 outputs a value 0
0 & 1 outputs a value 0
1 & 0 outputs a value 0
1 & 1 outputs a value 1
```

Examples of the logical operator AND:

`(5 == 2+3) & (10 == 3 * 3) = 0,` since (5 == 2+3) is *true* but (10 == 3 * 3) is not.

`(10 == 2*5) & (CONCAT('abc';'xyz') == 'abcxyz') = 1,` since both are *true*.

NOT (!)

This outputs the inverse of the input.
```
!1 outputs a value 0
!0 outputs a value 1
```

Examples of the logical operator NOT:

`!(17 == 10+7) = 0,` since (17 == 10+7) is *true*.

`!(19*100 > 2000 ) = 1,` since (19*100 > 2000) is *false*.

OR (|)

This outputs **1** if either input is 1 or if both are 1.

```
0 | 0 outputs a value 0
0 | 1 outputs a value 1
1 | 0 outputs a value 1
1 | 1 outputs a value 1
```

Examples of the logical operator `OR`:

`(5 == 2+3) | (10 <= 3*3) =1`, since (5 == 2+3) is *true*.

`(11 == 2*5) | (CONCAT('abc';'xyz') == 'hello') = 0`, since both are *false*.

Using expressions in macros

An expression is a list of variables and values with operators, which specify a value. In the following example the operators are **+**, *****, **sine()** and **-**.

```
(5+6)*10
sine(60)
$size-10
```

You can use an expression:

- to assign a value to a variable
- to print out its value
- in another command

For example:

To assign a value to a variable:

```
LET $result = (5+6)*10
```

Variable *$result* is assigned the value *110*.

To print the value of an expression:

```
PRINT sin(30)
```

0.500000 is displayed in the Command window.

To use an expression in another command:

```
SELECT ADD ARC 'my_arc'
MODIFY
RADIUS $size * 7
```

You cannot mix numeric and string variable types within an expression.

For each variable type, the operators perform various tasks.

- Operators for integers and real numbers (see page 29)
- Operators for strings (see page 31)
- Operators for lists (see page 31)
- Operators for vectors (see page 31)
- Comparison operators (see page 32)
- Logical operators (see page 33)
- Variable for arc tangent (see page 33)

Spaces may be included on each side of the operators.

Operators for integers and real numbers

Use the following operators for integers and real numbers:

+	addition
–	subtraction
*	multiplication
/	division
%	modulus; the remainder after two integers are divided; for example, 11%3 = 2
^	power of; for example, 2^3=2*2*2=8
sin( )	sine of an angle
cos( )	cosine of an angle
tan( )	tangent of an angle
atan( )	angle whose tangent is equal to the given value
acos( )	angle whose cosine is equal to the given value
asin( )	angle whose sine is equal to the given value
abs( )	absolute value of a number (removes any minus signs); for example, absolute(-56.98) = 56.98 = absolute(56.98)

`sqrt( )`	square root of a number; for example, sqrt (81) = 9
`log()`	output the natural logarithm of a number; for example, y = logarithm(7.389056) = 2
`exp()`	outputs the exponential value of a number with respect to e, the base of the natural logarithms; for example, y = exp(2) = e^2 = 7.389056
`min(A1; A2; … ; AN)`	outputs the minimum value of the list of numbers
`max(A1; A2; … ; AN)`	outputs the maximum value of the list of numbers
`compare (A; B; C)`	outputs 1 if A and B are equal within tolerance value C and 0 otherwise
`test ? result_true : result_false`	if *test* is true then *result_true* is assigned to the variable otherwise *result_false* is assigned

For example:

`LET $x = $a>=$b ? $a+$b : $a-$b`

This assigns *a+b* to *x* if a>=b and assigns *a-b* to *x* if a<b.

Operators for strings

Use the following operators on strings:

`length( )`	outputs the number of items in a string
`concat(string1; string2; … ; stringN)`	outputs a single string which is a combination of all the other strings

For example:

```
LET $name = 'Fred'
LET $greeting = concatenate ('Hello '; $name)
PRINT $greeting
```

In the Command window, this outputs the following

Hello Fred

The operators work with strings, integers and real numbers

Operators for lists

A list is represented as *{a; b; c;...}*. The operators for lists are:

`{a; b; c;...}[n]`	outputs the *n*th element of the list
`length({a; b; c;...})`	number of items in the list
`concat({a1; a2;...; an}; {...}; ... ; {...})`	outputs all the elements in the lists as a single list

Operators for vectors

Use the following operators on vectors, where `A` equals vector *[x;y;z]* and `B` equals *[a;b;c]*.

modulus(A)

This outputs the magnitude of the vector and is calculated as sqrt((x*x)+(y*y)+(z*z)). For example:

```
// define tolerance
LET $tol = 0.00001

// find the length of this vector
// (note: could use length($vec))
LET $dist = modulus(line[1].end - line[2].start)
```

```
// test if length is less than tolerance
LET $coinc = $dist < $tol

// if true, the two points are coincident
if $coinc {
print "End of line coincident with second line."
}
```

normal (A)

This outputs the unit vector of vector A. The unit vector has the same direction as vector A, but its modulus is 1.

```
// angle between line 1 and the x-axis,
LET $cosine=normal(line[1].end-line[1].start).[1;0;0]
LET $angle = acos( $cosine )

print "Angle between line 1 and the x axis is,"
print $angle
```

length(A)

This is the same as modulus.

(A) . (B)

This outputs the dot product of two vectors. The dot product is calculated as *((x*a)+(y*b)+(z*c))*.

cross()

This outputs the cross product of two vectors. This is the vector that is perpendicular to the two vectors. For example, the cross product of the X and Y axes is the Z axis.

```
print cross([1;0;0]; [0;1;0])
returns [0;0;1]
```

Comparison operators

Use these operators to compare two given values **A** and **B**.

`A == B`	outputs 1 if A equals B and 0 otherwise
`A != B`	outputs 1 if A does not equal B and 0 otherwise
`A < B`	outputs 1 if A is less than B and 0 otherwise
`A <= B`	outputs 1 if A is less or equal to B and 0 otherwise
`A > B`	outputs 1 if A is greater than B and 0 otherwise
`A >= B`	outputs 1 if A is greater or equal to B and 0 otherwise

Logical operators

Use the logical operators to compare expressions and variables:

`A & B`   outputs 1 if A and B are true and 0 otherwise. This is known as the **AND** operator

`A | B`   outputs 1 if either A or B is true and 0 otherwise. This is known as the **OR** operator

`! A`   outputs 1 if A is false and 0 if true. This is known as the **NOT** operator

Arc tangent

Use the following variable to calculate the arc tangent:

`atan2(arg1;arg2)`

This is useful for finding the azimuth and elevation for a unit vector `[i; j; k]`

```
let azimuth = atan2(j; i)
let elevation = asin(k)
```

Making decisions in macros

The `IF` (see page 34) command enables you to choose which commands are carried out next depending on the value of a variable.

If you ask the user to enter a number for the lateral they want to move, you do not know what value the user will enter. You can use a comparison to verify that the value that is entered is valid:

- if the value is valid, continue with the operation on the lateral.

- if the value in invalid, tell the user that their input is invalid and ask them to enter another value.

IF

When a condition is met, the **IF** command can be used to execute a series of commands.

```
$variable = (condition)
IF $variable {
  Commands A
}
Commands B
```

If the conditional test after `IF` is true then *Commands A* are executed followed by *Commands B*. If the test is false, then only *Commands B* are executed.
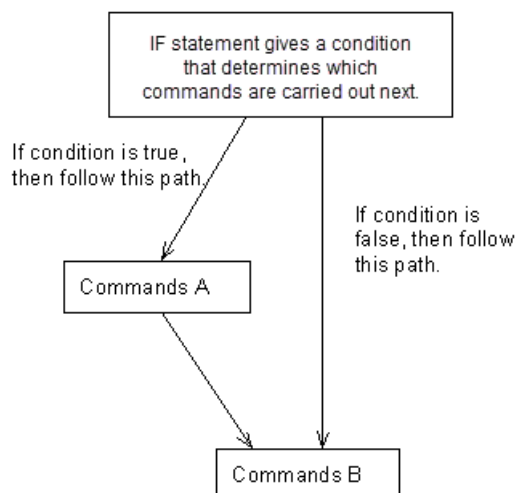
You must enclose *Commands A* in brackets **{}** and the brackets must be positioned correctly. The following command is *not* valid:

```
LET $invalid = ($radius == 3)
IF $invalid PRINT "Invalid radius"
```

To make this command valid, add the brackets as follows:

```
LET $invalid = ($radius == 3)
IF $invalid {
PRINT "Invalid radius"
}
```

The first bracket must be the last item on the line and on the same line as the `IF`*. The closing bracket must be on a line by itself.*

You can also specify commands that are performed only when the condition is *false*. These commands are specified using the `IF-ELSE` (see page 35) and `IF-ELSEIF-ELSE (see page 35)` commands.

IF-ELSE

```
IF $condition {
  Commands A
} ELSE {
  Commands B
}
Commands C
```

If the conditional test after `IF` is true, then *Commands A* are executed followed by *Commands C*. If the conditional test fails, then *Commands B* are executed followed by *Commands C*.
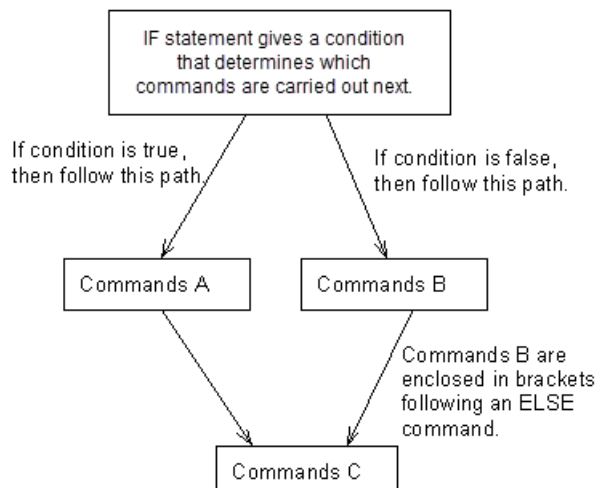
IF - ELSEIF - ELSE

```
IF $condition_1 {
  Commands A
} ELSEIF $condition_2 {
  Commands B
} ELSE {
  Commands C
}
Commands D
```

The above construct works as follows:

- If `condition_1` is true, then *Commands A* are executed followed by *Commands D*.

- If `condition_1` is false and `condition_2` is true, then *Commands B* are executed followed by *Commands D*.

- If `condition_1` is false and `condition_2` is false, then *Commands C* are executed followed by *Commands D*.
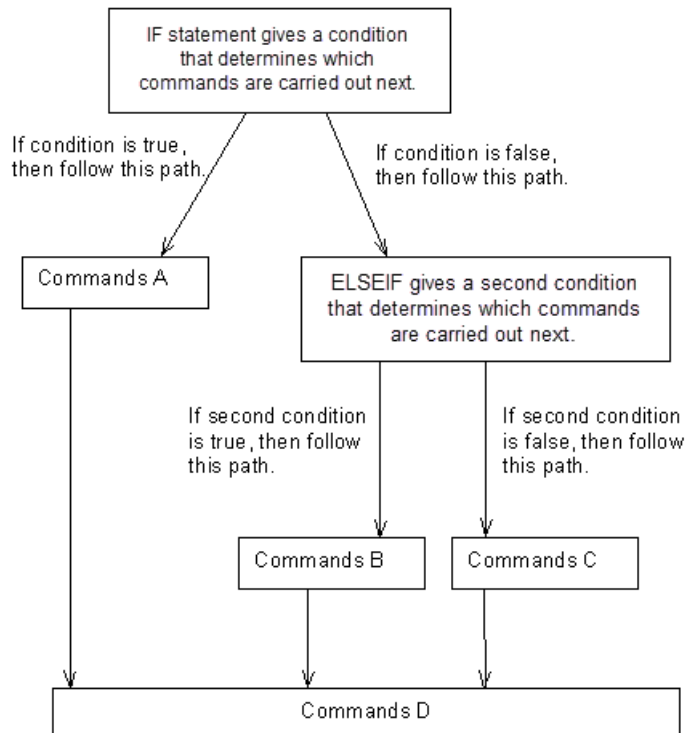
ELSE *is an optional command. There can be any number of* ELSEIF *statements in a block, but not more than one* ELSE. ELSEIF *can be written as one word or as* ELSE IF*.*

You can perform tests directly in *if* and *elseif* commands. So,

```
let e1 = $error == 1
let e2 = $error == 2
if e1 {
print e1
} elseif e2 {
print e2
}
```

can also be written as:

```
if ($error == 1) {
print e1
} elseif ($error == 2) {
print e2
}
```

Switch

When you compare a variable with a number of possible values and each value determines a different outcome, it is recommended that you use the SWITCH command (see page 101).

The SWITCH statement allows you to define a variable which is compared against a list of possible values. This comparison determines which commands are executed.

```
switch $variable {
  case (constant_A)
    Commands A
  case (constant_B)
    Commands B
  default
    Commands C
}
Commands D
```

This construct works as follows:

- if variable = constant_A, then *Commands A, B, C and D* are executed.

- if variable = constant_B, then *Commands B, C and D* are executed.

- if no match is made, then *Commands C and D* are executed.
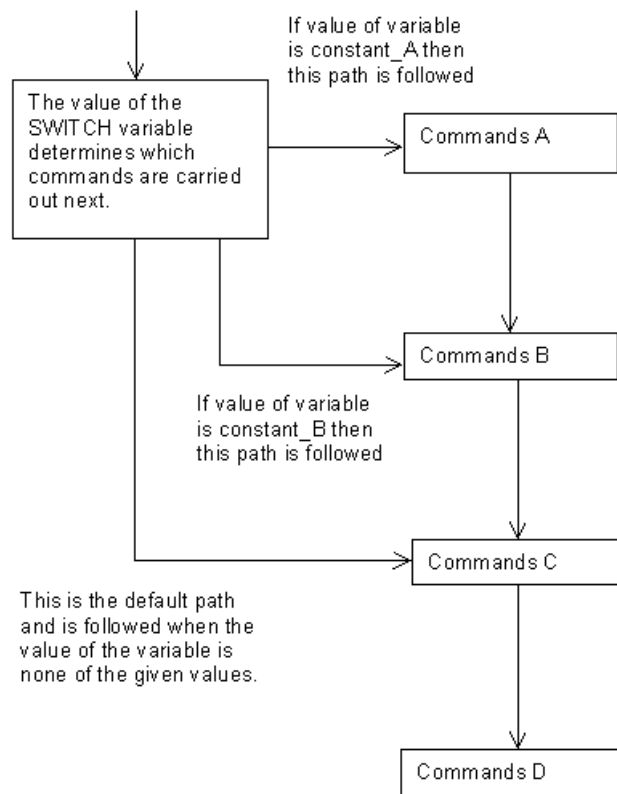
The commands are executed through the **switch** command. When a match is found, all the commands in the remaining **case** statements are executed. You may prevent this from happening by using a **break** statement:

```
switch $variable {
  case (constant_A)
    Commands A
    break
  case (constant_B)
    Commands B
    break
  default
    Commands C
}
Commands D
```

This construct works as follows:

- if variable = `constant_A`, then *Commands A and D* are executed.

- if variable = `constant_B`, then *Commands B and D* are executed.

- if no match is made, then *Commands C and D* are executed.
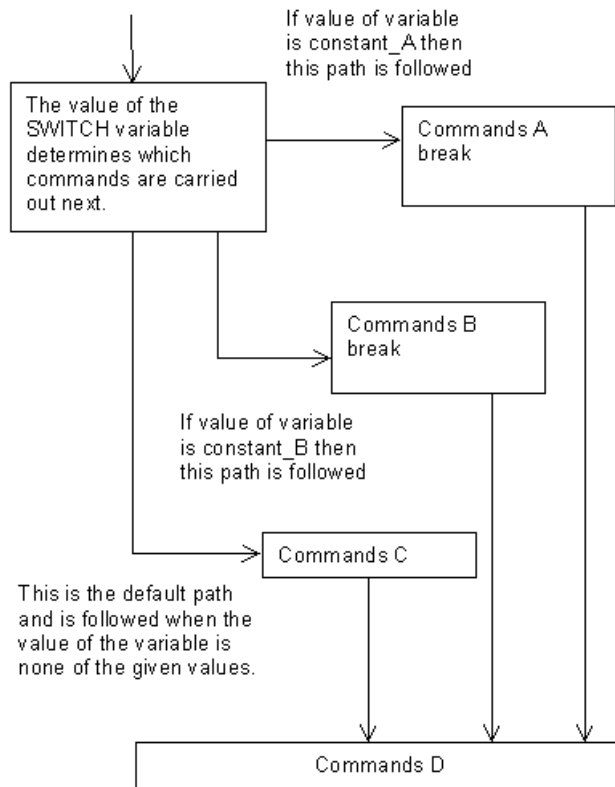
*There may be any number of **case** statements, but only one **default** statement.*

Repeating commands in macros

It is useful to repeat a command a number of times, for example, creating a circle at the start of every line in the model.

Commands that allow you to repeat commands are known as *loops*. There are two loop structures; the WHILE loop and the DO - WHILE loop.

A `WHILE` loop repeatedly executes a block of commands until its conditional test is false.

```
WHILE $condition {
  Commands A
}
Commands B
```

The construct works as follows:

1 If the conditional test after `WHILE` is true, then *Commands A* are executed and the conditional test repeated.

2 When the conditional test is false, *Commands A* are no longer executed and the program executes *Commands B*.
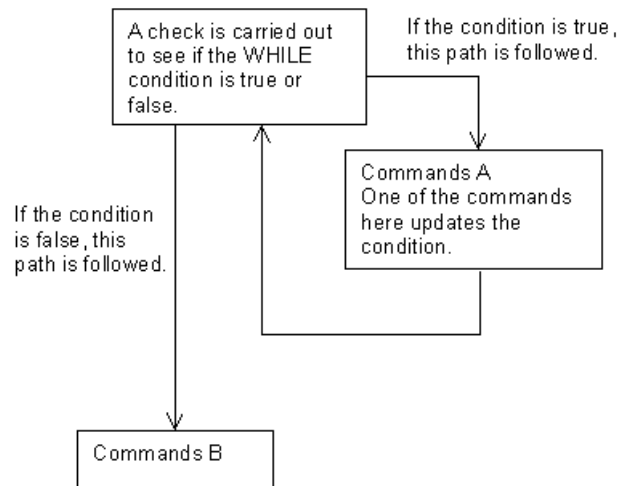
Within `WHILE` loops, you can jump to the end of the block of commands to:

- cancel the loop using the `BREAK` command

- continue with the next iteration using the `CONTINUE` command.

The `WHILE` loop checks its conditional test first to decide whether to perform its commands, by contrast, the `DO-WHILE` loop performs its commands and then checks whether to repeat.

```
DO {
   Commands A
} WHILE $condition
Commands B
```

This construct works as follows:

1 *Commands A* are executed, and if the conditional test after `WHILE` is true *Commands A* are repeated.

2 When the conditional test is false, *Commands A* are no longer executed and the program executes *Commands B*.
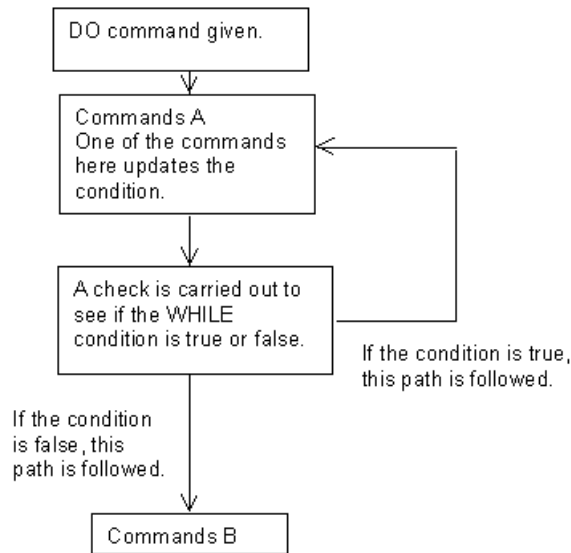
Within `DO` loops, you can jump to the end of the block of commands to:

- cancel the loop using the `BREAK` command

- continue with the next iteration using the `CONTINUE` command.

CONTINUE

`CONTINUE` causes a jump to the conditional test of any one of the loop constructs `WHILE` and `DO-WHILE` in which it is encountered, and starts the next iteration, if any.

An example is given below.

```
LET $a = 1
WHILE $a {
  INPUT NUMBER 'Input number of holes' $Holes
  LET $zerotest = ($Holes <= 0)
  IF $zerotest {
    Print "***Invalid input***"
    Print "Input must be greater than zero"
    CONTINUE
  }
  LET $a = 0
  LET $angle = (360/$Holes)
}
```

The user is asked to enter the number of holes. Before the calculation, you need to make sure that the number is valid. Using the CONTINUE command allows the user to enter the value again.
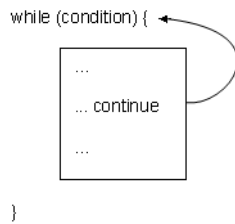
BREAK

BREAK causes a jump to the statement beyond the end of any one of the constructs WHILE, DO-WHILE, SWITCH in which it is encountered.

Nested constructs can require multiple breaks.
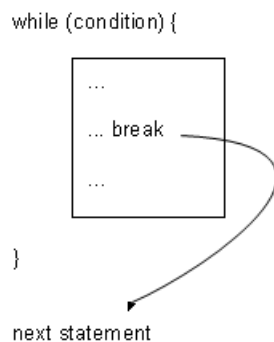
Jumping from one point in the macro to another

The GOTO command (see page 43) is used in conjunction with a label (see page 44). This construct:

- lets you jump from one point in a macro to another.
- is used mainly used with error checking; if an invalid condition is met, the macro file can be made to jump to an error message.

GOTO

The `GOTO` string causes a jump to the commands following a label (see page 44):

The following rules specify the use of `GOTO`:

- The destination label must be in the same macro as the `GOTO`.

- Jumps may be made forwards or backwards within the macro.

- Jumps may occur out of constructs (for example, out of an `IF-ELSE`, or `WHILE` block).

- Jumps may not be into constructs.

- If a jump is made out of a construct, the construct is cancelled appropriately.

`GOTO` makes a macro more difficult to follow and should be avoided where possible. However, `GOTO` can be used to make your macro clearer if used only as a forward jump, for example:

- to the end of a macro

- to lines near the end for printing error messages.

For example:

The following example shows how `GOTO` can be used. However it is better practice to use a loop instead of the `GOTO` command.

```
GOTO :input
// This jumps to the line in macro which looks like:
// :input
// :input is the label command that defines where the
goto jumps to.
:input
INPUT NUMBER "Lateral point number" $num
LET $test=(1>$num)|($num>surface[1].lateral[1].number)
IF $test {
  GOTO Error1
}
.
.
.
return

//Error messages
:Error1
PRINT '**A lateral must have more than 1 point.**'
GOTO input
```

The previous example could be written more clearly by using a `WHILE` loop to check the condition *$test*.

```
INPUT NUMBER "Lateral point number" $num
LET $test=(1>$num)|($num>surface[1].lateral[1].number)
WHILE $test {
PRINT '**A lateral must have more than 1 point.**'
INPUT NUMBER "Lateral point number" $num
LET $test=(1>$num)|($num>surface[1].lateral[1].number)
}
.
.
return
```

Labels

Labels are used in conjunction with the `GOTO` command to control progression through the macro.

Use a label as follows:

- At the beginning of any line in a macro file. They are alphanumeric prefixed with a colon :. For example:
  ```
  :draw
  ```

  The first non-space character defines the label, all other text is ignored. If text is added after the label it is treated as a comment. For example:
  ```
  :draw This text is a comment
  ```

- To jump forwards or backwards in the file to a position marked with a label.

- In macro files; it cannot be used as a typed command in the Command window.

- After a `GOTO` command:
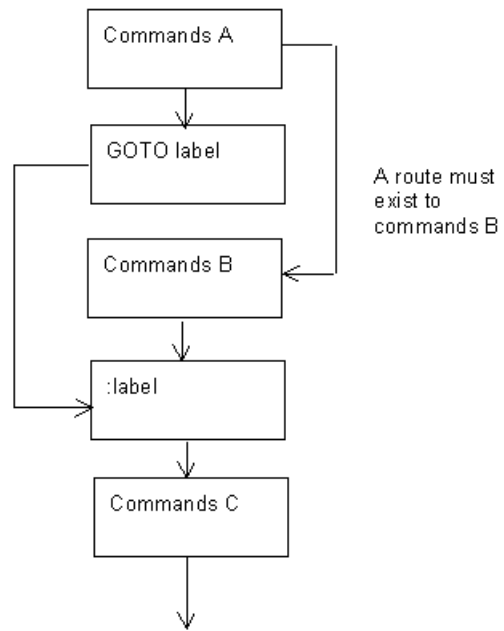
- Before a `GOTO` command
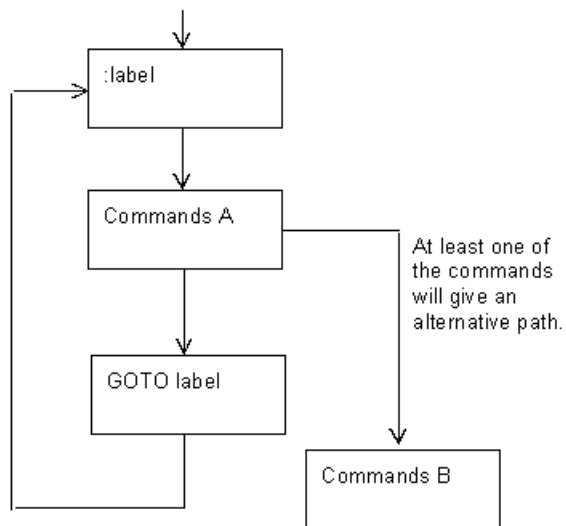
NOTE: *Ensure that a path exists to all the commands in the macro; otherwise you will have commands which are not used:*
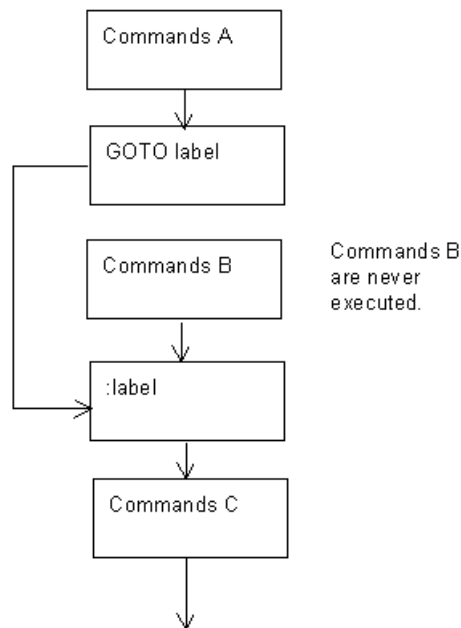
Ensure that you do not create an infinite loop (that is a loop in the macro which never exits):
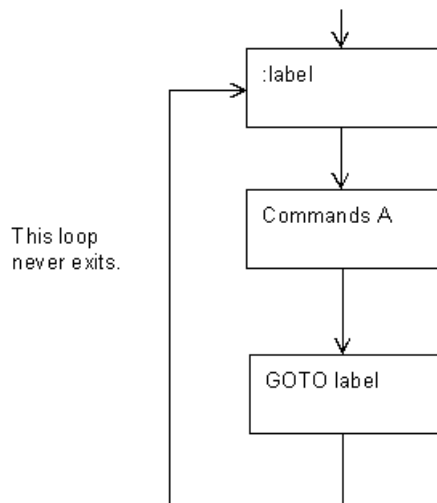
Defining a path to a directory in a macro

Use path commands to define directories where a macro looks for information when it is run. These commands can be used to set the directory path inside a macro when:

- importing files
- opening models
- running macros

The following commands are available:

PATH DELETE	deletes a single path
PATH DELETEALL	deletes all pre-defined paths to directories
PATH ADD BACK	creates a new path to a directory
PATH LIST	lists the paths (in the Command window)
PATH QUIT	quits the path commands

The following example shows how to run several macros from within another macro. The macros are stored in C:\Documents and Settings\xxx\My Documents.

```
PATH DELETEALL
PATH ADD BACK 'C:\Documents and Settings\xxx\My
Documents'
PATH LIST

MACRO RUN 'test1.mac'
MACRO RUN 'test2.mac'
MACRO RUN 'test3.mac'
```

Running a macro in another macro

You can embed an existing, tested, macro inside a new macro. This saves time on testing and repeating commands.

The command to run a macro from within a macro is:

```
MACRO RUN pathname_of_macro
```

If the name of a macro file contains spaces, the name must be included in double quotes. For example,
```
macro run "name of file.mac"
```

When you initiate a macro from a running macro, you can also pass values into the macro. The command to do this is described in Entering values during macro initiation (see page 14).

You can also pass expressions as arguments in the command line to run a macro from another macro. The result of the expression must be real.

```
macro run create_block.mac $length ($Length/2)
(2*$Length)
```

If one of the arguments in the command is a variable or an expression, or you have a negative number, you must take care with the use of brackets.

If you run the macro with the arguments

```
10 ($bob) -1
```

10 will be allocated to `$length`

`($bob)` –**1** will be evaluated and assigned to the second argument. This leaves nothing to be assigned to the third variable. So, only two sides of the block will have lengths assigned to them.

To allocate all three arguments, the correct use of brackets should be:

```
10 ($bob) (-1)
```

To make certain of the correct use of brackets, you can use brackets around the individual arguments at run time.

Exporting variables from a macro

You can export variables from a running macro. The command is:

```
EXPORT $variable_name
```

If the macro is running from within another macro, a variable of that name is either modified or created.

The following example shows how to pass values into a macro and export from one macro to another.

Macro1 has the following code in it:

```
LET $a = 50
LET $b = 100
LET $c = 200

MACRO RUN Macro2.mac $a $b $c

PRINT $a
PRINT $b
PRINT $c

PRINT $d
PRINT $e
PRINT $f
```

Macro2 has the following code:

```
ARGS{
INT a
INT b
INT c
}

LET $d = $a / 2
LET $e = $b / 2
LET $f = $c / 2

EXPORT $d
```

```
EXPORT $e
EXPORT $f
```

The result as shown in the Command window would be:

50

100

200

25

50

100

You also see the following warning:

Warning variable created

This means that the three variables that were exported from **Macro2** have been created in **Macro1** so that they can be printed.

Exporting file names

You can use the following macro command to pad out file export names in macros.

```
PADLEADING
```

The example below fills the variable *padded* with *00010*. This creates a string of width *5* containing the given string value *$s* where the variable *s = 10* padded with leading *0*s.

```
string padded = PADLEADING($s; 5; '0')
```

This macro command cannot be used with print commands.

Stepping from within a macro

You can step commands in a macro while the macro is running.

To enable stepping mode from within a macro, use the command:

```
EXECUTE STEP
```

To disable stepping mode, use:

```
EXECUTE RUN
```

When the command `EXECUTE STEP` is processed, the commands that follow it are stepped until the macro finishes or the command `EXECUTE RUN` is reached.

Pausing a macro

A pause temporarily stops a running macro. There are two types of pauses you can add to a macro:

- a pause that lasts a predefined number of seconds
- a pause that waits for the user to press a button to continue the macro.

If you pause a macro for a set number of seconds, after this period of time, the macro continues automatically. This command is useful when macros run too quickly for you to see what is happening to your model. By pausing the macro for a few seconds, you can see how the macro is operating on your model.

The command is:

`EXECUTE PAUSE integer`

where *integer* is the number of seconds you wish to pause the macro.

Use the `INPUT FREE` command to pause a macro indefinitely and display a dialog:
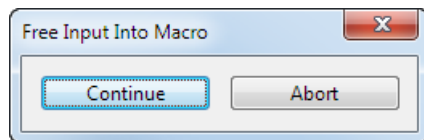
Click **Continue** to continue running the macro.

Click **Abort** to terminate the macro.

While the macro is paused, you can make changes to your model and then continue running the macro.

Ending a macro

A macro ends in the following cases:

- when it reaches its last command.
- when it executes a `RETURN` command.

Useful curve commands

To add a curve at a keypoint:

```
ADD CURVE fred AT KEYPOINT 2
```

If the keypoint does not exist, nothing happens

To add a composite curve:

```
ADD COMPCURVE fred AT COMPOSITE 3 KEYPOINT 5
```

If the keypoint does not exist, nothing happens

To make a span of a curve invisible,

```
SPAN_INVISIBLE span_number/point_index curve_id
DISPLAY REBUILD
```

To make a span of a curve visible,

```
SPAN_VISIBLE span_number/point_index curve_id
DISPLAY REBUILD
```

Use the following commands to control the display of the bad trimming dialog when exporting:

`EXPORTOPTS IGNOREBADTRIMON` supresses the dialog.

`EXPORTOPTS IGNOREBADTRIMOFF` displays the message dialog

Skipping command lines

In addition to stepping commands, you can also skip blocks of commands. This is done using the `SKIP` command. The following command causes *17* lines to be skipped:

```
SKIP 17
```

Helix macro tutorial

This example describes how to generate a macro to create a helix.
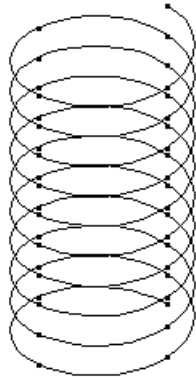
We suggest you complete this example before attempting to create your own macros.

While creating the helix macro, you edit a macro file to make changes to it. You can edit your own file or run a stored file. The stored files are in the following folder:

C:\Program Files\Autodesk\PowerShapexxxxx\file\examples\Macro_Writing

where xxxxx is the version number of PowerShape and C is the disk on which PowerShape is installed.

Recording and viewing the helix macro

We will record a macro to create the first turn of the helix. By recording the macro, you can find the commands to use in your macro. When you have the basic commands, you can enhance your macro.

1 Open the model in which you want to create the helix

2 Select Home tab > Macro panel > Record. The **Record Macro** dialog is displayed.

3 Browse to the folder, where you want to save the macro file.

4 In the **File name** box, type

   helix_turn

5 Click **Save** to close the dialog and start creating the macro.

6 Select Wireframe tab > Create panel > Curve > Bezier. This ensures that the **Curve** option is selected when you run the macro.

7 Type in the coordinates of the points of the curve in the graphics window:

10 0 0

-10 10 1

-10 -10 1

10 -10 1

10 10 1

This creates a spiral shape.
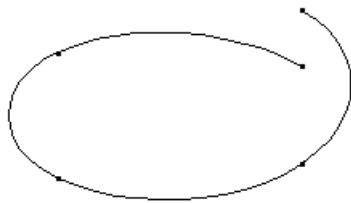
8 On the Quick Access toolbar, click to exit curve creation.

9 Select Home tab > Macro panel > Record to stop recording.

10 Open the macro in a text editor, such as **Notepad**. The macro should look as follows:

The first command tells the software to enter curve creation mode:

CREATE CURVE

The second command selects the Curve option from Wireframe tab > Create panel > Curve:

THROUGH

The third command inputs the co-ordinates of the points on the curve:

10 0 0

-10 10 1

-10 -10 1

10 -10 1

10 10 1

The final command exits curve creation mode and goes back to selection mode:

Select

If you want to use the helix macro to create threads in your models, a more appropriate macro to use is **helix.mac**, available in:

C:\Program Files\Autodesk\PowerShapexxxxx\file\examples\Macro_Writing

where xxxxx is the version number of PowerShape and C is the disk on which PowerShape is installed.

Running the macro

You can run a macro many times to perform the same task. This saves you time, because you do not have to enter each command individually in the task.

To run your macro file:

1   Delete the curve in your model.

2   Select Home tab > Macro panel > Macro > Run. The **Run Macro** dialog is displayed.

3   Select your macro file.

4   Click **Open**.

Editing the macro

This example shows how to edit the macro to create a helix with radius **50** and the distance between each turn (pitch) **20.**

1   Open your macro file in a text editor.

2   Edit the coordinates in your macro to:

    50 0 0

    -50 50 5

    -50 -50 5

    50 -50 5

    50 50 5

3   Save the file.

4   Select Home tab > Macro panel > Macro > Run to display the **Run Macro** dialog.

5   Select your macro file.

6   Click **Open** to create the helix.

Adding variables

You may want to create a helix using different values. This topic shows you how change the values in the macro to use variables.

The macro then creates a helix using the following variables:

- radius, which is set to *10*
- pitch (the length between each turn) which is set to *4*

You can:

- Edit your macro file
- Open and examine the file *helix_variable.mac* in the folder:

  C:\Program Files\Autodesk\PowerShapexxxxx\file\examples\Macro_Writing

  where xxxxx is the version number of PowerShape and C is the disk on which PowerShape is installed.

The changes are given in black text.

LET $radius = 10

LET $pitch = 4

LET $neg_radius = -$radius

LET $zheight = $pitch / 4

create curve

THROUGH

$radius 0 0

$neg_radius $radius $zheight

$neg_radius $neg_radius $zheight

$radius $neg_radius $zheight

$radius $radius $zheight

Select

The **LET** commands assign values. For example, the following command assigns 10 to *variable* $radius.

 LET $radius = 10

 For each coordinate, the value is replaced by a single variable. For example:

-50 50 5

becomes:

$neg_radius $radius $zheight

This makes it easier to change the values. Instead of changing all the coordinates each time you want to create a different size helix, you can now assign new values to the variables.

There are different variables for the negative and positive radius. The coordinate of each point in the curve is of the form:

x_value y_value z_value

where the values are either numbers or single variables. If you want to use expressions for positions in your macro, you must use the following:

POSITION

X expression_for_x

Y expression_for_y

Z expression_for_z

ACCEPT

where each expression is a valid expression in PowerShape's macro language.

To run the macro that includes variables:

1 Select Home tab > Macro panel > Macro > Run. The **Run Macro** dialog is displayed.

2 Select your macro file or *helix_variable.mac*. For further details see, Running the macro (see page 54).

3 Click **Open** to create the helix.

To change the values of the radius and pitch, open the macro file and edit the values in the macro. This saves time changing all the coordinate values.

For further details, see Using expressions in macros (see page 28).

Adding a loop

We want the helix to turn 10 times. To do this, we add a **while** loop.

You can:

▪ Edit your macro file

▪ Open and examine the file *helix_variable.mac* in the folder:

C:\Program Files\Autodesk\PowerShapexxxxx\file\examples\Macro_Writing

where xxxxx is the version number of PowerShape and C is the disk on which PowerShape is installed.

The changes are given in black text.

LET $radius = 10

LET $pitch = 4

LET $numturn = 10

LET $neg_radius = -$radius

LET $zheight = $pitch / 4

create curve

`THROUGH`

$radius 0 0

WHILE $numturn {

LET numturn = $numturn - 1

$neg_radius $radius $zheight

$neg_radius $neg_radius $zheight

$radius $neg_radius $zheight

$radius $radius $zheight

}

Select

The variable `numturn` indicates how many times the helix turns. The following command assigns a value to this variable:

- **LET $numturn = 10**

The value of `numturn` is also the condition of the **while** loop. You can read the **while** loop commands as:

- *While* `numturn` *does not equal zero, perform the commands in the brackets, { }.*

When the last bracket is reached, PowerShape checks numturn:

- If numturn does not equal zero, then the commands in the brackets {} are performed again.

- If numturn equal zero then the commands below the last bracket are carried out.

To run the macro that includes a loop:

1 Select Home tab > Macro panel > Macro > Run. The **Run Macro** dialog is displayed.

2 Select your macro file or *helix_loop.mac*. For further details see, Running the macro (see page 54).

3 Click **Open.**

The helix turns 10 times:
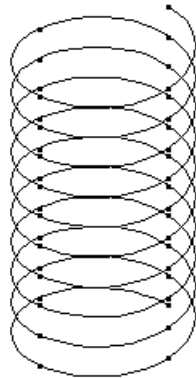
You can change the value of numturn in the command:

LET $numturn = 10

to make the helix turn a different number of times.

Adding comments

You can add comments to your macro to remind you what each command does. Two slashes // are added at the start of a line to show it is a comment. You can also use blank lines to separate blocks of commands.

For example, when you add:

// Calculating values for the co-ordinates

The two slashes // indicate the line contains a comment. When you run a macro, PowerShape ignores any comment lines, so it behaves in the same way with or without comments added. The comments can remind you of what a block of commands does.

You can either:

- Edit your macro file

- Open and examine the file *helix_variable.mac* in the folder:

  C:\Program Files\Autodesk\PowerShapexxxxx\file\examples\Macro_Writing

  where xxxxx is the version number of PowerShape and C is the disk on which PowerShape is installed.

The changes are given in black text.

// This macro creates a helix

// Written by: John Doe

// Values to change the size of the helix

```
LET $radius = 10
LET $pitch = 4
LET $numturn = 10


// Calculating values for the co-ordinates
LET $neg_radius = -$radius
LET $zheight = $pitch / 4


// Creating the helix's curve
create curve
THROUGH

  // The first co-ordinate
  $radius 0 0


  // Using a loop to input the
  // co-ordinates for each turn
  WHILE $numturn {
    LET numturn = $numturn - 1
    $neg_radius $radius $zheight
    $neg_radius $neg_radius $zheight
    $radius $neg_radius $zheight
    $radius $radius $zheight

  }
// Exiting curve creation mode
Select
```

To run the macro that includes comments:

1 Select Home tab > Macro panel > Macro > Run. The **Run Macro** dialog is displayed.

2 Select your macro file or *helix_comments.mac.* For further details see, Running the macro (see page 54).

3 Click **Open**.

   The same helix is created as described in Adding a loop (see page 56).

Interacting with the user

If you don't want to edit the macro each time you need to create a helix with a different size, you can display dialogs to request the values.

You can either:

- Edit your macro file

- Open and examine the file *helix_variable.mac* in the folder:

  C:\Program Files\Autodesk\PowerShapexxxxx\file\examples\Macro_Writing

  where xxxxx is the version number of PowerShape and C is the disk on which PowerShape is installed.

To edit your macro file:

1 Comment out the following commands in your macro.

   // LET $radius = 10

   // LET $pitch = 4

   // LET $numturn = 10

   These commands will be replaced. However, you can leave the commands in your macros as comments, in case you want to use them again.

2 Add the following commands before the commands which are given in Step 1.

   // Displays dialogs to input values

   INPUT NUMBER 'Radius of helix' $radius

   INPUT NUMBER 'Pitch (per turn)' $pitch

   INPUT NUMBER 'Number of turns' $numturn

The **INPUT NUMBER** command tells the user to input a number.

When the macro is run, the command:

Input NUMBER 'Radius of helix' $radius

displays the dialog shown below:
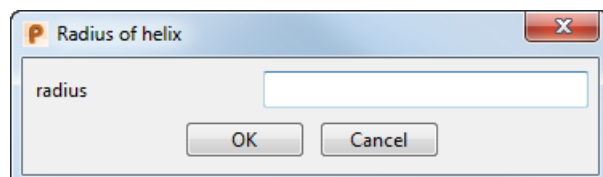
The string '**Radius of helix**' is the title of the dialog. When the user enters a value, it is assigned to the variable *$radius*. The name of the variable is on the left of the data box on the dialog.

To run the interactive macro:

1   Select Home tab > Macro panel > Macro > Run. The **Run Macro** dialog is displayed.

2   Select your macro file or *helix_interact.mac.*

3   Click **Open**. The **Radius of helix** dialog is displayed.

4   Enter a value and click **OK**. The **Pitch** dialog is displayed.

5   Enter a value and click **OK**. The **Number of turns** dialog is displayed.

6   Enter a value and click **OK**. The values are inserted in the macro and the helix is drawn using the values.

Changing the origin of the helix

In this example, the origin of the helix is the origin of the current workspace. We want to use any position as the origin.

We will add code so that the user can click a point on the screen to define the origin.

You can either:

- Edit your macro file

- Open and examine the file *helix_variable.mac* in the folder:

  C:\Program Files\Autodesk\PowerShapexxxxx\file\examples\Macro_Writing

  where xxxxx is the version number of PowerShape and C is the disk on which PowerShape is installed.

The changes are given in black text.

// This macro creates a helix

// Written by: John Doe

// Displays dialogs to input values

INPUT POINT 'Position of centre' $cenpos

INPUT NUMBER 'Radius of helix' $radius

INPUT NUMBER 'Pitch (per turn)' $pitch

INPUT NUMBER 'Number of turns' $numturn

```
// Values to change the size of the helix
// LET $radius = 10
// LET $pitch = 4
// LET $numturn = 10


// Calculating values for the co-ordinates
LET $neg_radius = -$radius
LET $zheight = $pitch / 4


// Creating the helix's curve
create curve
THROUGH

  // The first co-ordinate
    // $radius 0 0
  LET start_x = $radius + $cenpos_x
  LET start_y = $cenpos_y
  LET start_z = $cenpos_z
  $start_x $start_y $start_z

    // Using a loop to input the
  // co-ordinates for each turn
  WHILE $numturn {
   LET numturn = $numturn - 1
   $neg_radius $radius $zheight
   $neg_radius $neg_radius $zheight
   $radius $neg_radius $zheight
   $radius $radius $zheight
  }

// Exiting curve creation mode
Select
```

Further information on changing the origin of the helix

The command:

INPUT POINT 'Position of centre' $cenpos

displays the following dialog:
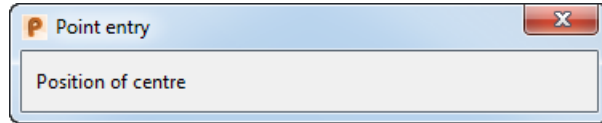
The dialog remains displayed on the screen until the user enters a point.

The point data is entered into the variable *$cenpos*. You can obtain the x co-ordinate of the point using the variable *$cenpos_x.* Similarly, the y and z co-ordinates can be obtained.

The following commands enter the first point of the helix relative to the input position.

LET start_x = $radius + $cenpos_x

LET start_y = $cenpos_y

LET start_z = $cenpos_z

$start_x $start_y $start_z

To run the macro that changes the origin of the helix:

1   Select Home tab > Macro panel > Macro > Run. The **Run Macro** dialog is displayed.

2   Select your macro file or *helix_origin.mac*.

3   Click **Open**. The **Point entry** dialog appears asking for you to input a position.

4   Click a point on the screen. The three dialogs are displayed as described in interacting with the user (see page 60).

5   Enter values in each dialog and click **OK**. The helix is drawn on the screen.

Creating a helix around a cylinder

The helix is now constructed relative to a user-defined point. This topic describes how to extend the macro so the helix is constructed around an existing primitive cylinder (surface).

When the macro is running, the user selects the cylinder. The user is then asked:

- The number of turns to the helix
- The length of the pitch

The helix is drawn around the cylinder.

The macro also:

- Lets the user select the cylinder.
- Creates a temporary workplane at the workplane of the cylinder. The temporary workplane gives the centre of the helix and the orientation of the workplane.

You can either:

- Edit your macro file
- Open and examine the file *helix_variable.mac* in the folder:

  C:\Program Files\Autodesk\PowerShapexxxxx\file\examples\Macro_Writing

  where xxxxx is the version number of PowerShape and C is the disk on which PowerShape is installed.

The changes are given in black text.

// This macro creates a helix

// Written by: John Doe

// Clear the selection list

SELECT CLEARLIST

// Selecting a cylinder

INPUT SELECTION 'Select a cylinder'

LET cyl = selection.object[0]

// Displays dialogs to input values

// INPUT POINT 'Position of centre' $cenpos

// INPUT NUMBER 'Radius of helix' $radius

INPUT NUMBER 'Pitch (per turn)' $pitch

INPUT NUMBER 'Number of turns' $numturn

// Values to change the size of the helix

// LET $radius = 10

// LET $pitch = 4

// LET $numturn = 10

```
//Creating a temporary workplane
CREATE WORKPLANE
$cyl.origin.x $cyl.origin.y $cyl.origin.z


// Modifying the workplane


MODIFY
NAME tmpwkhelix
XAXIS DIRECTION
X $cyl.xaxis.x
Y $cyl.xaxis.y
Z $cyl.xaxis.z
ACCEPT
YAXIS DIRECTION
X $cyl.yaxis.x
Y $cyl.yaxis.y
Z $cyl.yaxis.z
ACCEPT
ZAXIS DIRECTION
X $cyl.zaxis.x
Y $cyl.zaxis.y
Z $cyl.zaxis.z
ACCEPT
ACCEPT


    // Calculating values for the co-ordinates
LET $radius = abs($cyl.lat[1].point[1].x)
LET $neg_radius = -$radius
LET $zheight = $pitch / 4


// Creating the helix's curve
create curve
THROUGH
```

// The first co-ordinate

$radius 0 0

// LET start_x = $radius + $cenpos_x

// LET start_y = $cenpos_y

// LET start_z = $cenpos_z

// $start_x $start_y $start_z

// Using a loop to input the

// co-ordinates for each turn

WHILE $numturn {

  LET numturn = $numturn - 1

  $neg_radius $radius $zheight

  $neg_radius $neg_radius $zheight

  $radius $neg_radius $zheight

  $radius $radius $zheight

}

// Exiting curve creation mode

// and deleting the temporary

// workplane

Select

SELECT CLEARLIST

SELECT ADD WORKPLANE 'tmpwkhelix'

DELETE

Before the cylinder is selected, we clear the selection list using the following command.

SELECT CLEARLIST

The command

INPUT SELECTION 'Select a cylinder'

displays the following dialog:

This dialog tells the user to select objects.

When the user clicks **OK**, your macro can get the details of what is selected by accessing the 'selection' object.

The following command assigns the first object in the selection to the variable *cyl*.

LET cyl = selection.object[0]

selection.object[0] is the first object in the selection. This object is assigned to variable *cyl*.

To find out more information about the selected object, you can use either:

- **selection.object[number].syntax**

- **cyl.syntax**

where syntax is the syntax associated with the selected object. For further details on the list of syntax for each object, see PowerShape object information (see page 143).

When you write macros, we advise you to assign the selected objects you want to use later in your macro to other variables. If the selection changes, you will obviously lose your selection.

For further details, see: Creating a workplane at the origin of the cylinder (see page 67)

Creating a workplane at the origin of the cylinder

The following command creates a workplane at the origin of the cylinder.

CREATE WORKPLANE

$cyl.origin.x $cyl.origin.y $cyl.origin.z

The variable *$cyl* is the primitive cylinder. We have used the syntax of the primitive cylinder to find out its origin.

The commands below edit:

- the name of the workplane

- the direction of each axis of the workplane to match the axis on the instrumentation of the primitive.

MODIFY

NAME tmpwkhelix

XAXIS DIRECTION

X $cyl.xaxis.x

Y $cyl.xaxis.y

Z $cyl.xaxis.z

ACCEPT

YAXIS DIRECTION

X $cyl.yaxis.x

Y $cyl.yaxis.y

Z $cyl.yaxis.z

ACCEPT

ZAXIS DIRECTION

X $cyl.zaxis.x

Y $cyl.zaxis.y

Z $cyl.zaxis.z

ACCEPT

ACCEPT

The commands to use in your macro may not be obvious. You may need to:

1 Record a macro

2 Open the macro in a text editor

3 Copy the commands in your macro.

For example, to create and edit a workplane, record a macro to create a workplane, and then edit the properties you want to use in your macro.

The following command:

LET $radius = abs($cyl.lat[1].point[1].x)

uses the x co-ordinate of point 1 of lateral 1 of the cylinder to define the radius.

The command below uses the origin of the workplane to define the start point of the helix.

$radius 0 0

The following three lines clear the selection, then select and delete the workplane.

```
SELECT CLEARLIST
SELECT ADD WORKPLANE 'tmpwkhelix'
DELETE
```

Adding user selection of the cylinder to the macro

You can either:

- Edit your macro file.

- Open and examine the file *helix_variable.mac* in the folder:

  C:\Program
  Files\Autodesk\PowerShapexxxxx\file\examples\Macro_Writing

  where xxxxx is the version number of PowerShape and C is the
  disk on which PowerShape is installed.

The changes are given in black text.

```
// This macro creates a helix

// Written by: John Doe


// Clear the selection list
SELECT CLEARLIST


// Selecting a cylinder
INPUT SELECTION 'Select a cylinder'
LET cyl = selection.object[0]


// Displays dialogs to input values
// INPUT POINT 'Position of centre' $cenpos
// INPUT NUMBER 'Radius of helix' $radius
INPUT NUMBER 'Pitch (per turn)' $pitch
INPUT NUMBER 'Number of turns' $numturn


// Values to change the size of the helix
// LET $radius = 10
// LET $pitch = 4
// LET $numturn = 10
```

```
//Creating a temporary workplane
CREATE WORKPLANE
$cyl.origin.x $cyl.origin.y $cyl.origin.z

// Modifying the workplane

MODIFY
NAME tmpwkhelix
XAXIS DIRECTION
X $cyl.xaxis.x
Y $cyl.xaxis.y
Z $cyl.xaxis.z
ACCEPT
YAXIS DIRECTION
X $cyl.yaxis.x
Y $cyl.yaxis.y
Z $cyl.yaxis.z
ACCEPT
ZAXIS DIRECTION
X $cyl.zaxis.x
Y $cyl.zaxis.y
Z $cyl.zaxis.z
ACCEPT
ACCEPT

    // Calculating values for the co-ordinates
LET $radius = abs($cyl.lat[1].point[1].x)
LET $neg_radius = -$radius
LET $zheight = $pitch / 4


// Creating the helix's curve
create curve
THROUGH
```

```
// The first co-ordinate

$radius 0 0
//  LET start_x = $radius + $cenpos_x
//  LET start_y = $cenpos_y
//  LET start_z = $cenpos_z
//  $start_x $start_y $start_z


// Using a loop to input the
// co-ordinates for each turn
WHILE $numturn {
  LET numturn = $numturn - 1
  $neg_radius $radius $zheight
  $neg_radius $neg_radius $zheight
  $radius $neg_radius $zheight
  $radius $radius $zheight
}


// Exiting curve creation mode
// and deleting the temporary
// workplane


Select
SELECT CLEARLIST
SELECT ADD WORKPLANE 'tmpwkhelix'
DELETE
```

Before the cylinder is selected, clear the selection list using the following command.

SELECT CLEARLIST

The command

INPUT SELECTION 'Select a cylinder'

displays the following dialog:

This dialog tells the user to select objects.

When the user clicks **OK**, your macro can get the details of what is selected by accessing the 'selection' object.

The following command assigns the first object in the selection to the variable *cyl*.

LET cyl = selection.object[0]

selection.object[0] is the first object in the selection. This object is assigned to variable *cyl*.

To find out more information about the selected object, you can use either:

- **selection.object[number].syntax**

- **cyl.syntax**

where syntax is the syntax associated with the selected object. For further details on the list of syntax for each object, see PowerShape object information (see page 143).

When you write macros, we advise you to assign the selected objects you want to use later in your macro to other variables. If the selection changes, you will lose your selection.

For further details, see: Creating a workplane at the origin of the cylinder (see page 67)

Run your macro that creates a helix around a cylinder

To create a helix around a cylinder:

1 Create a primitive cylinder (surface).

2 Select Home tab > Macro panel > Macro > Run. The **Run Macro** dialog is displayed.

3 Select your macro file or *helix_cyl.mac*.

4 Click **Open**. The **Select a cylinder** dialog is displayed.

5 Select the primitive cylinder.

6 Click **Accept**.

   Two dialogs are displayed, asking for the *pitch* and the *number of turns*.

7 Enter values in each dialog and click **Accept**.

   The helix is drawn around the cylinder.

Testing input data

Many macros fail because the input data is wrong. To make sure that the correct data is input, you can test the data. If the wrong data is entered, prompt the user to input the data again.

The macro checks:

▪ if a single object is selected

▪ if the single object is a surface

▪ if the surface is a cylinder

If none of the above are true, the user is prompted that a single cylinder must be selected and then given an option to exit the macro. If the user decides to continue, they are asked to select a cylinder again. The macro also checks if the helix is smaller or larger than the cylinder.

Run the macro to check if the tests work:

1 Create different objects in your model to test your macro. Make sure you have a primitive cylinder.

2 Select Home tab > Macro panel > Macro > Run. The **Run Macro** dialog is displayed.

3 Select your macro file or *helix_test.mac*.

4 Click **Open**. The **Select a cylinder** dialog is displayed asking for you to select a cylinder.

5 Select a couple of objects.

6 Click **OK.** The following message is displayed:
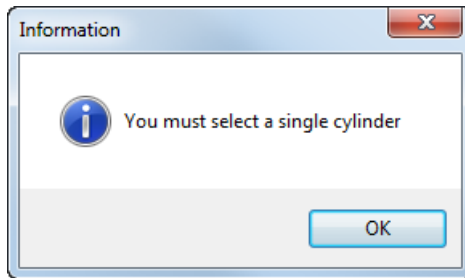
7 Click **OK**. The **Query** dialog is displayed:
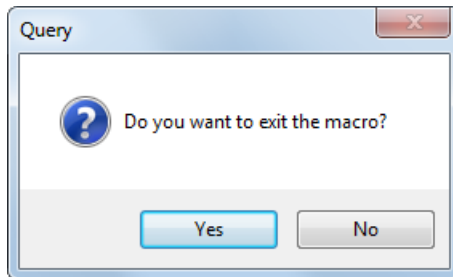
If you click **Yes**, the macro exits. If you click **No**, the **Select a cylinder** dialog appears.

8 Click **Yes** to exit the macro.

9 Run the macro again.

10 Select a couple of objects.

11 This time, when the **Query** dialog asks you whether to exit the macro, click **No**.

12 The **Select a cylinder** dialog appears. Select a cylinder.

13 Click **OK**.

14 The two dialogs appear as described earlier asking for the pitch and the number of turns. Enter values in each dialog and click **OK.** The helix is created around the cylinder.

- If the helix is larger than the cylinder the following message appears.
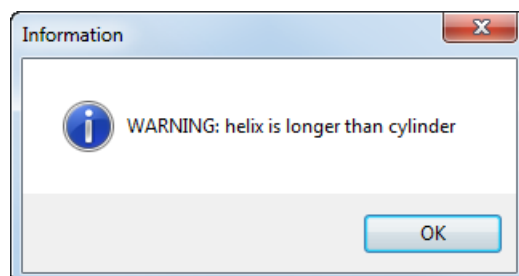
- If the helix is smaller, the following message appears.
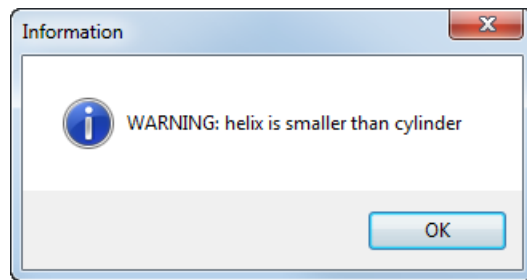
- If the helix fits the cylinder, no message is displayed.

15 Run the macro again and input different values for the helix to test all the options.

Adding tests to your macro

You can either edit your macro file or open and examine the file *helix_test.mac* in the folder:

C:\Program Files\Autodesk\PowerShapexxxxx\file\examples\Macro_Writing

where xxxxx is the version number of PowerShape and C is the disk on which PowerShape is installed.

The changes are given in black text.

// This macro creates a helix

// Written by: John Doe

// Asking the user to select a cylinder

// and then checking that the selection

// contains only a cylinder

LET $no_cyl = 1

WHILE $no_cyl {

  // Clear the selection list

  SELECT CLEARLIST

  // Selecting a cylinder

  INPUT SELECTION 'Select a cylinder'

  // Testing if a single object is selected

  LET $single = selection.number == 1

```
IF $single {
  // Testing if the single object is
  // a surface
```

The strings *Surface* and *Cylinder* must use the correct capitalisation.

```
  LET $surf = selection.type[0] == 'Surface'
  IF $surf {
  // Testing if the surface is a cylinder
    LET $no_cyl=!(selection.object[0].type == 'Cylinder')
  }
}

IF $no_cyl {
  PRINT ERROR 'You must select a single cylinder'
  INPUT QUERY 'Do you want to exit the macro?' $prompt
 IF $prompt {
   RETURN
 }
 }
}

LET cyl = selection.object[0]


// Displays dialogs to input values
// INPUT POINT 'Position of centre' $cenpos
// INPUT NUMBER 'Radius of helix' $radius
INPUT NUMBER 'Pitch (per turn)' $pitch
INPUT NUMBER 'Number of turns' $numturn


// Values to change the size of the helix
// LET $radius = 10
```

```
// LET $pitch = 4
// LET $numturn = 10


//Creating a temporary workplane
CREATE WORKPLANE
$cyl.origin.x $cyl.origin.y $cyl.origin.z


// Modifying the workplane

MODIFY
NAME tmpwkhelix
XAXIS DIRECTION
X $cyl.xaxis.x
Y $cyl.xaxis.y
Z $cyl.xaxis.z
ACCEPT
YAXIS DIRECTION
X $cyl.yaxis.x
Y $cyl.yaxis.y
Z $cyl.yaxis.z
ACCEPT
ZAXIS DIRECTION
X $cyl.zaxis.x
Y $cyl.zaxis.y
Z $cyl.zaxis.z
ACCEPT
ACCEPT


// Checking the size of the helix and warning
// the user if too small or too big
LET $helix_height = $pitch * $numturn
LET $length = abs($cyl.long[1].point[2].z)
```

```
Let $big = ($helix_height > $length)
IF $big {
PRINT ERROR 'WARNING: helix is longer than cylinder'
}
Let $small = ($helix_height < $length)
IF $small {
PRINT ERROR 'WARNING: helix is smaller than cylinder'
}


// Calculating values for the co-ordinates
LET $radius = abs($cyl.lat[1].point[1].x)
LET $neg_radius = -$radius
LET $zheight = $pitch / 4


// Creating the helix's curve
create curve
THROUGH

  // The first co-ordinate
  $radius 0 0
  //  LET start_x = $radius + $cenpos_x
  //  LET start_y = $cenpos_y
  //  LET start_z = $cenpos_z
  //  $start_x $start_y $start_z

  // Using a loop to input the
  // co-ordinates for each turn
  WHILE $numturn {
    LET numturn = $numturn - 1
    $neg_radius $radius $zheight
    $neg_radius $neg_radius $zheight
    $radius $neg_radius $zheight
```

```
      $radius $radius $zheight

   }


// Exiting curve creation mode
// and deleting the temporary
// workplane


Select
SELECT CLEARLIST
SELECT ADD WORKPLANE 'tmpwkhelix'
DELETE
```
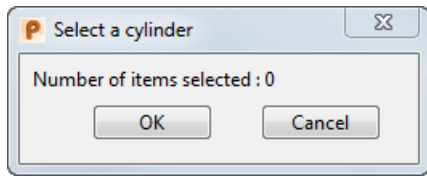
More information on adding tests to your macro

Two tests are added to:

- check if a single object is selected and that it is a cylinder
- check if the helix is smaller or larger than the cylinder

The tests used the **IF** command to check if the data is valid. With any test, you must decide what to do if the data is not valid.

The macro will fail if a cylinder is not selected as the first object. When selecting objects, we cannot always guarantee which is the first object. We have restricted users to selecting a single cylinder.

- The following command assigns a value of *1* to the variable *no_cyl*. This is the condition of the loop and shows that no single cylinder is selected.

  LET $no_cyl = 1

- The **While** loop continues to perform its commands while no cylinder is selected.

  WHILE $no_cyl {

  Carry out commands within the brackets

  }

- In the loop, the following clear the selection list and ask the user to select a cylinder.

  SELECT CLEARLIST

  INPUT SELECTION 'Select a cylinder'

- Test to see if the selection only contains a single object. In the following command, **selection.number** is the number of items selected.

LET $single = selection.number == 1

The following statement:

selection.number == 1

checks if the left and right sides are equal. In our case, we want to know if *1* object is selected. If this is true, then **$single** becomes *1*. Otherwise **$single** becomes zero.

- The following checks the value of **$single.** If the value is *1*, then the commands within the brackets are carried out.

    IF $single {

    Carry out commands within the brackets

        }

    If the value is *0*, then the commands after the closing bracket are carried out.

- These are the commands in brackets:

    LET $surf = selection.type[0] == 'Surface'

    IF $surf {

    LET $no_cyl=!(selection.object[0].type == 'Cylinder')

    }

    They check if the single object is a surface and whether that surface is a primitive cylinder. If the object is a primitive cylinder, then the variable **$no_cyl** becomes *0*.

- When we have tested the selection and still do not have a single cylinder selected, we want to tell the user that a single cylinder must be selected and ask whether to exit the macro.

    This command checks if **$no_cyl** is *1* and then displays two dialogs.

    IF $no_cyl {

- The following command displays one of the dialogs.

    PRINT ERROR 'You must select a single cylinder'
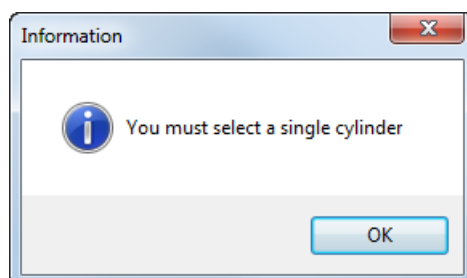
This tells you what is wrong. As soon as the user clicks **OK**, the following command is carried out.

INPUT QUERY 'Do you want to exit the macro?' $prompt

This displays the following dialog.
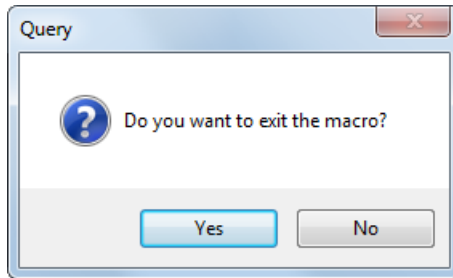
If the user clicks **Yes**, the variable **$prompt** becomes *1*. If the user clicks **No**, the variable becomes *0.*

If **$prompt** is *1*, then the command **RETURN** is carried out. This command exits the macro.

IF $prompt {

RETURN

}

- The second test warns the user if the helix is longer or smaller than the cylinder. The commands below test the size of the helix against the length of the cylinder and display warnings where necessary.

LET $helix_height = $pitch * $numturn

LET $length = abs($cyl.long[1].point[2].z)

Let $big = ($helix_height > $length)

IF $big {

PRINT ERROR 'WARNING: helix is longer than cylinder'

}

LET $small = ($helix_height < $length)

IF $small {

PRINT ERROR 'WARNING: helix is smaller than cylinder'

}

Examples of macros

Blanking

Using these macros to blank items.

```
// Blanking all curves
QUICK QUICKSELECTWIRE
DISPLAY BLANKSELECTED
//
// Blanking all surfaces
QUICK QUICKSELECTSURF
DISPLAY BLANKSELECTED
```

Calculate the volume of each solid in the selection

Use this macro to calculate the volume of each solid in the selection and print the total volume of all the solids.

```
// This selects all the solids in the model
//
FILTERBUTTON FilterItems
SelectType solid
All
ACCEPT
//
REAL s_total = 0
PRINT 'Start total = '$s_total
//
LET numturn = selection.number
//
WHILE $numturn {
  LET $numturn = $numturn - 1
  REAL s_vol = selection.object[$numturn].volume
  LET s_name = selection.object[$numturn].name
  PRINT 'Volume of solid '$s_name ' = '$s_vol
  REAL s_total = ($s_total + $s_vol)
}
//
SELECT EVERYTHING PARTIALBOX
SELECT clearlist
//
PRINT 'Total volume of selected solids = '$s_total
```

Close all models

Use this macro to close all open models.

```
LET n = window.number
LET w = $n > 0
WHILE $w {
  FILE CLOSE SELECTED YES
  LET w = window.number
}
```

Create a curve from a selection of points

Use this macro to create a curve from a selection of points.

```
// This example uses lists and vectors
// This only works correctly if there are no
// duplicate points. The curve is also created
// in the order the points are taken from
// the selection list and this is only
// really controlled by the number order they
// are created in. Need a model with points in it


// select all the points in the model
FILTERBUTTON FilterItems

SelectType  Point

InvertType

InvertType

All

accept


// Quit if we have no points selected
LET numpts = selection.number

LET e = ($numpts==0)

IF $e {

  PRINT 'No points are selected.'

   return

}
```

```
// Create a list of points
LIST all_points = { }
LET i = 0
LET carry_on = ($i < $numpts)
WHILE $carry_on {
  LET point_obj = SELECTION.OBJECT[$i]
  VECTOR pt = $point_obj.POSITION
   LIST_ADD $all_points END $pt
   LET i = $i + 1
   LET carry_on = ($i < $numpts)
}


// Create a curve that goes through all the points
CREATE CURVE THROUGH
LET i = 1
LET carry_on = ($i <= $numpts)
WHILE $carry_on {
  VECTOR pt = $all_points[$i]
  REAL x = $pt[1]
   REAL y = $pt[2]
   REAL z = $pt[3]
   STRING command = concatenate('abs ';  $x; ' '; $y; ' ';
$z)
   EXECUTE COMMAND $command

   LET i = $i + 1
   LET carry_on = ($i <= $numpts)
}

SELECT
EVERYTHING PARTIALBOX
```

Create a tapered helix

Use this macro to create a tapered helix.

```
// This macro creates a tapered helix for either an
external or internal thread.
//
// Ask the user to select a workplane and then check that
the selection contains only a workplane,
// the workplane is then made activate.
// The Helix will be created about this workplane, so Z
must be aligned at the centre of the screw
//
// Use a while loop to make the correct selection
LET $no_wkp = 1
WHILE $no_wkp {

  // Clear the selection list
  select clearlist

  // Selecting a workplane
  INPUT SELECTION 'Select a workplane'

  // Testing if a single object is selected
  LET $single = selection.number == 1
  IF $single {
    // Test if the single object is a workplane
    LET $seltype = selection.type[0] == 'Workplane'
    IF $seltype {
          // If the selection is correct activate the
workplane and carry on creating the curves
             pri 'Selection correct'
             Modify ACTIVATE Accept
             let $no_wkp = $no_wkp - 1
       }
    } ELSE {
       // Else ask to exit the macro or make a new selection
```

```
        INPUT QUERY 'Do you want to exit the macro?'
$prompt

        IF $prompt {

            // If YES exit the macro

            Print 'Exiting the macro'

        RETURN

            } ELSE {

            // Try selecting again

            select clearlist

            print 'Trying selecting again'

                INPUT SELECTION 'Select a workplane'

            }

    }

}


// Prompt the user to input the values for the number of
turns, radius and height

// Query whether the thread is internal of external

input number 'Number of turns (whole number)' $hn

input number 'Radius of the helix' $hr

input number 'Height of the helix' $hh

input query  'Is this an external thread?' $yesno


real $hz1 = ($hh / $hn)

real $hz2 = ($hh - ($hh / $hn))



if $yesno {

    // if the thread is external create this curve

    create curve helix

    0

    height ($hh / $hn)

    turns 1

    same off

    radius2 ($hr - 1)
```

```
radius1 ($hr)

accept

string $c1 = selection.name[0]

create curve helix

0 0 $hz1

height ($hh - (2*($hh / $hn)))

turns ($hn - 2)

same on

radius1 ($hr)

accept

create curve helix

0 0 $hz2

height ($hh / $hn)

turns 1

same off

radius1 ($hr - 1)

radius2 ($hr)

accept

} else {

    // if the thread is internal create this curve

    create curve helix

    0

    height ($hh / $hn)

    turns 1

    same off

    radius2 ($hr + 1)

    radius1 ($hr)

    accept

    string $c1 = selection.name[0]

    create curve helix

    0 0 $hz1

    height ($hh - (2*($hh / $hn)))

    turns ($hn - 2)
```

```
            same on

            radius1 ($hr)

            accept

            create curve helix

            0 0 $hz2

            height ($hh / $hn)

            turns 1

            same off

            radius1 ($hr + 1)

            radius2 ($hr)

            accept


    }


    // Create a composite curve from the three separate
    curves

    select clearlist

    create curve compcurve

    add curve $c1

    save

    checkquit
```

Create geometry

Use this macro to create geometry to be used in the macro.

```
// this creates the geometry to be used in the macro
// Two intersecting planes are created and then a curve
is created from the intersection.
PRINCIPALPLANE XY
create surface PLANE
0
PRINCIPALPLANE ZX
create surface Plane
PLANE
0
SelectAll
create curve INTERSECT
ACCEPT
//
// set the name to be used for the curve
```

```
STRING new_name = 'fred'
//
// find out how many items were created
LET c_obj = created.number
PRINT 'Number of created items ' $c_obj
//
// the WHILE loop checks that a composite curve was
created and renames the composite curve
//
WHILE $c_obj {
//
  LET $c_obj = $c_obj - 1
//
  LET n = created.object[$c_obj].name
  LET t = created.type[$c_obj]
  IF $t == 'Composite Curve' {
    LET $t = 'Compcurve'
  }
//
  SELECT clearlist
//
  LET com = concatenate('add '; $t;' "'; $n; '"')
  EXECUTE COMMAND $com
  PRINT $com
//
  RENAME
  VAR_NAME $new_name
  ACCEPT
//
}
```

Create normal workplane for each point on a curve

The following example creates a normal workplane for each point on a curve:

```
// This macro assumes you have already created the curve
in the model
// A dialog is raised to select the curve you want to
use.
// Does not work for composite curves
//
// Selecting a curve
INPUT SELECTION 'Select a curve'
//
// find out the name of the curve
LET name = selection.name[0]
PRINT $name
//
// find out the number of points in the curve
LET numturn = curve[$name].number
```

```
PRINT $numturn
select clearlist
//
// create a point at each keypoint of the curve
WHILE $numturn {
  select clearlist
  create workplane NormalSingle
  Position
  KEYPOINT
  add Curve $name
  NUMBEREDPOINT
  KEYPTNUMBER $numturn
  APPLY
  cancel
//
    LET numturn = $numturn - 1
}
//
select
```

Create text in a macro

Use this macro to create text in the macro.

When LIVETEXT is on, this macro will not work; you cannot enter live text using a variable.

```
// How to create text using a variable in a macro
// Livetext on does not work
//
//
TOOLS PREFERENCES
UNITPREFS
TEXTPREFS
TEXT LIVETEXT OFF
ACCEPT
//
STRING fred = 'wibble'
LET MYTEXT = 'fred'
// INPUT TEXT 'Enter some text' $fred
//
CREATE TEXT TEXT HORIZONTAL YES
0 0 0
ScrolledText $fred
Accept
TEXT FONT Arial
TEXT HEIGHT 0.3
TEXT PITCH 0.1
SELECT
select clearlist
//
```

```
TOOLS PREFERENCES
UNITPREFS
TEXTPREFS
TEXT LIVETEXT ON
ACCEPT
//
LET com = concatenate(''''; ($fred); '''')
p $fred
CREATE TEXT TEXT HORIZONTAL YES
20 0 0
EXECUTE COMMAND $com
SELECT
```

Deactivate all solids in a model

Use this macro to deactivate all solids in a model.

```
// Need some solids in the model
// Get the name of the currently active solid (this will
return "There is no active solid" if there isn't an
active solid)
//
STRING active_solid_name = SOLID.ACTIVE
// Deactivate the active solid
LET e = SOLID[$active_solid_name].EXISTS
IF $e {
  SELECT CLEARLIST
  ADD SOLID $active_solid_name
  MODIFY MODIFY DEACTIVATE ACCEPT
  SELECT CLEARLIST
}
```

Deleting pcurves

Use this macro to delete pcurves.

```
toolbar tools edit
toolbar tools fixing
TRIMREGIONEDIT
//
// The following command was added
//
EDITPCURVE
//
ADD_ALL_CURVES
DELETE
TOOLBAR TREDIT LOWER SELECT
SELECT CLEARLIST
```

DO - WHILE loop macro

This macro uses a `DO-WHILE` loop to create a point and ask a question.

```
// Need a model open for this to work
//
DO {
  PRINT 'looping'
  create point
  0 0 0
  select
  // ask a question to get the 1 or 0 for the exit of the
  loop
  INPUT QUERY 'Do you want to create another hole?' $fred
} WHILE $fred
  PRINT 'finishing'
  RETURN
```

Dynamic sectioning

Use this macro to create a dynamic section.

```
VIEW CLIPPLANES RAISE
VIEW CLIPPLANES EDGES ON
```

Exporting multiple images

Use this macro to export images to a file.

```
// Need to have a 3D object in the model and need to
change the macro to select that object for it to be
rotated
//
// Here is the powershape macro that made the frames:
// The incremental rotation per frame.
INT inc = 60
//
// The maximum angle through which to rotate.
INT max_angle = 360
//
INT frame_number = 0
//
LET true = 1
WHILE $true {
  //
  // Make the filename for this frame.
  LET frame_number = $frame_number + 1
  //
  // Make a STRING containing the frame number.
  STRING frame_name = inttostring($frame_number)
```

```
//
// Pad this name with leading zeros to ensure the names
collate correctly.
STRING padded_name = padleading($frame_name; 5; '0')
//
// Make the complete filename.
STRING filename = concatenate('e:xxxx\PRINT\f';
$padded_name; '.png')
//
// Print to the file.
print tofile replace $filename
//
// Have we finished?
LET angle = $inc * ($frame_number - 1)
LET finish = ($angle > $max_angle)
IF $finish {
  RETURN
}
PRINT "Angle = " $angle
//
// Rotate the target object.
select add solid '1'
edit rotate
angle $inc
apply
dismiss
select
select clearlist
}
//
// This macro creates a number of .png files, one per
frame.
// It may be more convenient to create .jpg files.
// You can now turn these frames into a movie.
```

Export using variables

Use this macro to export to a dgk file using variables.

```
// need to have a model open with some items in it to
export

// Other conversions
//==================================
// inttoreal
// inttostring
// realtoint
// realtostring
// stringtoint
// stringtoreal
//==================================
```

```
//
// set path to export to
//
LET path = 'e:\xxxx\'
//
// Set the value of the INT
INT numturn = 10
//
WHILE $numturn {
  // Convert the INT to a STRING
  STRING fred = inttostring($numturn)
  //
  // Do the export using concatenate
  selectall
  LET com2 = concatenate('file export '; $path; $fred;
  '.dgk')
  PRINT $com2
  EXECUTE COMMAND $com2
  //
  LET numturn = $numturn - 1
//
}
```

Importing components from an .xt file

Use this macro to import components from an .xt file.

```
// The macro will work if you open a New model then
Import the xt file.
// It assumes that there are no previously named levels.
// It also assumes that the objects imported are
components.
//
// Select all the imported components
//================================
//
selectall
//
// store the number of components
//=================================
//
LET numturn = selection.number
//
// Start the loop
//==================================
//
WHILE $numturn {
  //
  // Set some variables
  //=============================
  LET s_com = $numturn - 1
```

```
LET $l_name = selection.name[$s_com]
//
// Set the start number of 501 for the levels
//================================
LET lev_num = 500 + $numturn
//
// renames the level with the name of the component
//================================
LET com = concatenate('LEVEL RENAME '; $lev_num;' ';
$l_name)
EXECUTE COMMAND $com
//
//clear the selection so one component can be added to
a level
//================================
Select clearlist
add Component $l_name
//
// adds the selection to the renamed level
//================================
LET com = concatenate('LEVEL POPUP RAISE '; $lev_num)
EXECUTE COMMAND $com
Level Popup AddSelection
//
// select everything again
//================================
selectall
//
// reset the loop number
//================================
LET numturn = $numturn - 1
}
```

Move points on a curve

Use this macro to move points on a curve.

```
// This relies on the compcurve being created with
// an even number of points in a vertical line.
// The point of the tooth should be the even number.
// The move gradually gets bigger.
//
select clearlist
//
// Selecting the composite
INPUT SELECTION 'Select a composite curve'
LET c_name = selection.object[0].name
select clearlist
INPUT NUMBER 'Enter distance to move point by' $Distance
add compcurve $c_name
//
```

```
// number of points in the curve
LET c_num = compcurve[$c_name].point.number
//
// set distance m to move the point
REAL m = 0
WHILE $c_num {
  // add the curve
  add compcurve $c_name
  //
  //select point on curve
  select_points $c_num
  end_select
  //
  // move the point
  $m 0 0
  //
  // clear the selection
  select clearlist
  //
  // set the new distance value of m
  LET m = $m - $Distance
  //
  // set the new point number
  // even numbers and top of tooth is every two points
  LET c_num = $c_num - 2
}
```

Select and add object

Use this macro to add the selected object.

```
// adds the first item in the selection
//
SELECTALL
//
LET n = selection.name[0]
LET t = selection.type[0]
//
select clearlist
//
LET com = concatenate('add '; $t;' "'; $n; '"')
EXECUTE COMMAND $com
PRINT $com
```

Offset surface curves by different distances

Use this macro to offset surface curves by different distances.

```
// Need to have a powersurface in an open model
//
LET $no_pow = 1
WHILE $no_pow {
  // Clear the selection list
  SELECT CLEARLIST
  // Selecting a powersurface
  INPUT SELECTION 'Select a single Powersurface'
  // Testing IF a single object is selected
  LET $single = selection.number == 1
  IF $single {
    // Testing IF the single object is a surface.
    // The strings Surface and Powersurface must use the
    correct capitalisation.
    LET $surf = selection.type[0] == 'Surface'
    IF $surf {
      // Testing IF the surface is a Powersurface
      LET $no_pow =! (selection.object[0].type ==
      'Powersurface')
    }
  }
  IF $no_pow {
    PRINT ERROR 'You must select a single powersurface'
    INPUT QUERY 'Do you want to exit the macro?' $prompt
    IF $prompt {
      RETURN
    }
  }
}
//
LET s_name = selection.object[0].name
//
select clearlist
//
INPUT NUMBER 'Enter overall distance to offset furthest
surface curve by' $Distance
//
// number of laterals in the surface
LET s_num = surface[$s_name].nlats
//
select clearlist
//
WHILE $s_num {
  // add the surface
  add surface $s_name
  //
```

```
// select a curve on a surface
// have to use the concatenate and EXECUTE COMMAND to
piece together the add lateral command
LET sel_curve = concatenate('select_lats '; $s_num)
EXECUTE COMMAND $sel_curve
//
// move the point
toolbar tools edit
EDIT SUBEDITS ON
edit offset
distance $Distance
select
//
// clear the selection
select clearlist
//
// set the new Distance value to be
LET Distance = $Distance - ($Distance / $s_num)
//
// set the new surface curve number
LET s_num = $s_num - 1
}
```

Open psmodels from a directory list

Use this macro to open psmodels from a directory list.

```
// Use directory['pathname'].files['pattern']

// to open all psmodels from a known directory


// Get list of models in a known directory

let model_list =
directory['E:\homes\clb\xxxx'].files['*.psmodel']


// Set the number of psmodels in the directory

let num_models = LENGTH($model_list)


// Create a while loop to open the psmodels

LET i = 1

LET carry_on = ($i <=  $num_models)

WHILE $carry_on {


  // Find the name of the psmodel

  let model_name = $model_list[$i]
```

```
     print $model_name


     // Construct command to open the psmodel
     string command = concatenate('name '; $model_name)
     print $command


     // Open the psmodel
     FILE OPEN
     EXECUTE COMMAND $command
     ACCESS READWRITE
     ACCEPT


     // reset the number to loop to the next psmodel
     LET i = $i + 1
     LET carry_on = ($i <=  $num_models)


   }
```

Open x_t from a directory list

Use this macro to open all files of type x_t from a known directory.

```
// Use directory['pathname'].files['pattern']
// to import all files of type x_t from a known directory
// Each file is imported into its own psmodel


// Get list of models in directory
let model_list =
directory['E:\homes\clb\xxxx'].files['*.x_t']


// Set number of files in the directory
let num_models = LENGTH($model_list)


// Create a while loop to import all the files
LET i = 1
LET carry_on = ($i <=  $num_models)
WHILE $carry_on {
```

```
    // open a psmodel to import the file into
    // This line can be commented out if all files
    // are required in the same psmodel
    FILE NEW

    // Find the name of the file
    let model_name = $model_list[$i]
    print $model_name

    // Construct command to open the file
    string command = concatenate('file import ';
$model_name)
    print $command

    //Import the file
    EXECUTE COMMAND $command

    // reset the number to loop to the next file
    LET i = $i + 1
    LET carry_on = ($i <=  $num_models)


}
```

Using LOOP to print the length of lines to a file

Use this option to print the lengths of lines to a file. The name and location of the file is specified at run-time.

```
args{
STRING filename
}
//
// in the command window enter a line like
// macro run E:\testdata\test_macros\loop-to-PRINT-
length-of-lines-to-a-file.mac 'E:xxxx\fred.txt'
// need to have a model open with some lines in it
//
// -----------------------------------------
// Open txt outfile to hold the report.
```

```
// ----------------------------------------
LET use_dialog = $filename == 'dialog'
IF $use_dialog {
    file outfile open Dialog
    Title Create a graphics report file
    FileTypes txt File (.txt)|*.txt
    Raise
} ELSE {
    // This must be an absolute filename.
    file outfile open replace $filename
}
//
// Open the file to PRINT to
LET filename = outfile.name
//
// PRINT the name of the file in the file
PRINT 'This file is ' $filename ''
//----------------------------------------
// Find the length of the lines
//----------------------------------------
FILTERBUTTON FilterItems
SelectType  Line
All
accept
//
LET numturn = selection.number
WHILE $numturn {
    LET s_line = $numturn - 1
    LET l_name = selection.object[$s_line].name
    LET l_len = line[$l_name].length
    PRINT 'Length of line '$l_name ' is '$l_len
    LET numturn = $numturn - 1
}

EVERYTHING PARTIALBOX
select clearlist
// ----------------------------------------
// Close the file you are printing to
file outfile close
```

Using SWITCH

Use this macro to use SWITCH to define a variable which is
compared against a list of possible values.

```
// you need some objects in the model and some selected
// IF you have two objects selected it will DO case 2 and
the default
STYLE LOWERFORM
LET e = selection.number
PRINT $e
```

```
//
STYLE RAISEFORM
SWITCH $e {
//
  case 2
  PRINT 'selection is 2'
  Style Name Blue
  //
  case 3
  PRINT 'selection is 3'
  //
  case 4
  PRINT 'selection is 4'
  Style Width 0.7
  create arc full
  0 0 0
  select
  //
  default
  PRINT 'default case'
  Style Pattern Dotted
  Select clearlist
  STYLE LOWERFORM
  //
}
PRINT 'you are at the end of the switch'
```

Using WHILE loop to create point at centre of arc

Use this macro to create a point at the centre of an arc.

```
// need a model with some arcs in it
FILTERBUTTON FilterItems
SelectType arc
All
accept
//
LET numturn = selection.number
//
WHILE $numturn {
  LET $numturn = $numturn - 1
  LET $l_name = selection.name[$numturn]
  //
  LET s_cenx = selection.object[$numturn].centre.x
  LET s_ceny = selection.object[$numturn].centre.y
  LET s_cenz = selection.object[$numturn].centre.z
  //
  select clearlist
  //
  Create point
  $s_cenx $s_ceny $s_cenz
```

```
        select
        //
        select clearlist
        //
        FILTERBUTTON FilterItems
        SelectType arc
        All
        accept
    }
    //
    select clearlist
    EVERYTHING PARTIALBOX
```

HTML application tutorial

This tutorial shows you how to write an application using Hypertext Mark-up Language (HTML) to create the following helix.
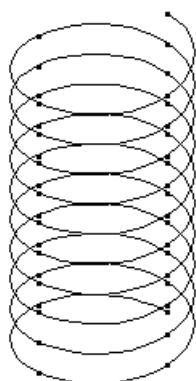
It should be possible to work through this tutorial without any prior knowledge of HTML. Detailed explanations of the HTML codes are not given; they can be found in any book on HTML.

When creating applications using HTML files, you may need to record macros to find the commands. It is therefore advisable to complete the Macro tutorial (see page 52) before working through the HTML tutorial.

Opening a new text file

To create a new text file to store the HTML codes:

1   Create a new file in a text editor, such as Notepad.

2   Add the following to the text file:

<HTML>

<HEAD>

</HEAD>

<BODY>

</BODY>

</HTML>

3   Save the file as helix.htm.

This file now contains the basic layout of the HTML file in two sections:

HEAD — Contains descriptive information about the HTML file as well as other information such as style rules or scripts.

BODY — The basic HTML commands to define the controls.

Adding controls to the application

To add controls in the HTML file:

1   Add code to the **BODY** section so that it looks as follows:

   <BODY>

   <h1>Helix creation</h1>

   <FORM NAME=helix>

   Radius <INPUT TYPE=text NAME=radius VALUE="10" > <p>

   Pitch <INPUT TYPE=text NAME=pitch VALUE="4" > <p>

   Turns <INPUT TYPE=text NAME=turns VALUE="10" > <p>

   <INPUT TYPE=button VALUE=" Apply " ><p>

   </FORM>

   </BODY>

2   Save the file.

The **FORM** object lets you to add controls that input data. It is defined as follows:

<FORM NAME=helix>

</FORM>

The **INPUT** object lets you add controls inside the form. The code has added two types of control:

▪   **Text box**

   <INPUT TYPE=text NAME=radius VALUE="10" >

This code contains a variable called **VALUE**. This puts a default value in the text box.

- **Button**

  <INPUT TYPE=button VALUE=" Apply " >

Displaying the HTML file in PowerShape

Open the file in Internet Explorer inside PowerShape to see what the html page looks like:

1 Start PowerShape.

2 Double-click the Command box in the status bar to open the Command window.

3 In the Command window, type:

   browser explorer {path}helix.htm

   where **{path}** is the location of *helix.htm*

   You should see the following:
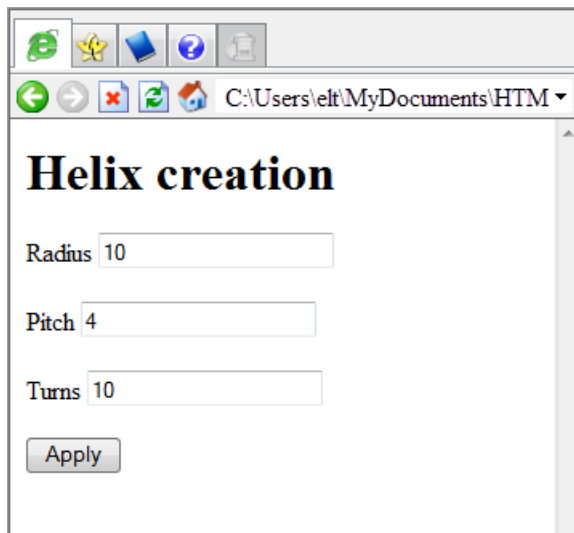
You can change the values in the text boxes and click the **Apply** button, but, as yet, this application does nothing in PowerShape.

Connecting to PowerShape using VBscripts

You can use VBscripts to write the code that allows you to communicate with PowerShape. You can also use other script languages such as Javascript (see page 124).

1 Add code to the **HEAD** section so that it looks like this:

   <HEAD>

   <script language="VBscript">

```
// Connect to the PowerShape

set pshape = Window.external


</script>

</HEAD>
```

2 Save the file.

The line with the two slashes // is a comment. The script is enclosed in the following lines of code:

- **<script language="VBscript">**

- **</script>**

The language used by the script is given in the first line.

The following command connects PowerShape using the object called *pshape:*

- **set pshape = Window.external**

Interacting with PowerShape

To make the dialog work with PowerShape:

- Add the commands that communicate with PowerShape to create a simple helix (see page 52).

- Add a procedure (see page 107).

- Link the procedure to the **Apply** button (see page 111).

Adding the Apply_click() procedure

PowerShape understands the commands used in macros. The best way to work out the commands to use is by recording a macro.

You are strongly recommended to complete the Helix macro tutorial (see page 52) before creating your own HTML applications.

The following are commands from the macro in the Helix macro tutorial:

LET $radius = 10

LET $pitch = 4

LET $numturn = 10

LET $neg_radius = -$radius

LET $zheight = $pitch / 4

create curve

THROUGH

$radius 0 0

WHILE $numturn {

LET numturn = $numturn - 1

$neg_radius $radius $zheight

$neg_radius $neg_radius $zheight

$radius $neg_radius $zheight

$radius $radius $zheight

   }

Select

The following steps show you how to convert these commands into vbscript commands:

1 In the **script** section, add the procedure called **Apply_click()** as shown below.

<script language="VBscript">

// Connect to PowerShape

set pshape = Window.external

Sub Apply_click()

//Calculating values for the coordinates

neg_rad = - document.helix.radius.value

zheight = document.helix.pitch.value /4

//Creating the helix's curve

pshape.Exec "Create curve"

pshape.Exec "through"

//First coordinates of the curve

pshape.Exec "abs " & document.helix.radius.value & " 0 0"

```
//Using a loop to input the coordinates from each turn

Counter = document.helix.turns.value

Do Until Counter = 0

Counter = Counter - 1

pshape.Exec neg_rad & " " & document.helix.radius.value & " " &
zheight

pshape.Exec neg_rad & " " & neg_rad & " " & zheight

pshape.Exec document.helix.radius.value & " " & neg_rad & " " &
zheight

pshape.Exec document.helix.radius.value & " " &
document.helix.radius.value & " " & zheight

Loop


//Exiting curve creation mode

pshape.Exec "Select"


End Sub


</script>
```

2 Save the file.

More information on the Apply_Click() procedure

The following commands are in the macro:

```
LET $radius = 10

LET $pitch = 4

LET $numturn = 10
```

In the HTML file, we have already assigned values to the *radius*, *pitch* and the *number of turns* when we created their text boxes.

- We assigned values to the variables *neg_radius* and *zheight* as in the macro commands.

  The following commands are in the macro:

  ```
  LET $neg_radius = -$radius

  LET $zheight = $pitch / 4
  ```

  In the HTML file, we use the values from the text boxes of the radius and the pitch. So for *neg_radius:*

  ```
  neg_rad = - document.helix.radius.value
  ```

This assigns the negative value of the radius to the variable *neg_rad*.

- The command:

  document.helix.radius.value

  defines the elements in the HTML file from which the string is extracted. The code `value` extracts the numeric value of the string in the textbox called *radius*. There are two other elements, *document* and *helix*:

  document denotes the current page;

  helix is the name of the form which contains the text box.

- Similarly, a value is assigned to the variable *zheight:*

  zheight = document.helix.pitch.value /4

- For the following macro commands, we use the **pshape.Exec** method to replace some of the code:

  create curve

  THROUGH

  $radius 0 0

  WHILE $numturn {

  LET numturn = $numturn - 1

  $neg_radius $radius $zheight

  $neg_radius $neg_radius $zheight

  $radius $neg_radius $zheight

  $radius $radius $zheight

  }

  Select

  So, the **create curve** command line has become:

  pshape.Exec "create curve"

- The **pshape.Exec** method uses strings to communicate with PowerShape.

  $radius 0 0

  has now been replaced by:

  pshape.Exec document.helix.radius.value & " 0 0"

  The **&** joins the strings on each side of it.

  So:

  document.helix.radius.value & " 0 0"

is a single string containing the contents of the Radius text box and two zeros. This is equivalent to the macro command:

$radius 0 0

- The **while** loop in the macro has been replaced by **Do Until Loop**. Both loops operate in a similar way.

- The following have been replaced by the **pshape.Exec** command and variables containing strings.

$neg_radius $radius $zheight

$neg_radius $neg_radius $zheight

$radius $neg_radius $zheight

$radius $radius $zheight

The strings are combined using **&** and **" "** characters.

So, for example:

$neg_radius $radius $zheight

becomes:

pshape.Exec neg_rad & " " & document.helix.radius.value & " " & zheight

Linking the procedure to the Apply button

To link the procedure to the **Apply** button:

1 Add **onClick=Apply_click()** to the input object for the **Apply** button as follows:

<INPUT TYPE=button value=" Apply " onClick=Apply_click() >

2 Save the file.

*In the string below, the **onClick** command defines the action when you click the **Apply** button. In this case, it calls the procedure **Apply_click()**, that was added in the script:*

<INPUT TYPE=button value=" Apply " onClick=Apply_click() >

Testing your application

To run your application:

1 Right-click in the Browser window in PowerShape to display a context menu.

2 Select **Refresh** from the context menu to install the latest *helix.htm* file in the browser.

3 Click the **Apply** button in the Browser window to create a helix using the default values.

4 Change the values in the three text boxes.

5 Click **Apply** again to create a helix using the new values.

Adding a Quit button to exit the HTML application

You can add a **Quit** button to the form that opens a HTML file when it is selected.

The following command adds a button with label **Quit** on the HTML page:

<INPUT TYPE=button VALUE="Quit" onClick="document.location = 'http://www.autodesk.com'" ><p>

When you click the **Quit** button, the action is defined by the following:

onClick="document.location = 'http://www.autodesk.com'"

This opens the Autodesk home page, providing you have internet access from your computer. If you don't have internet access, change the address to any HTML file you can access.

To add a **Quit** button on the same line as the **Apply** button:

1 Remove **<p>** from the following line in the HTML file:

<INPUT TYPE=button value=" Apply " onClick=Apply_click() ><p>

2 After this line, insert the following:

<INPUT TYPE=button VALUE="Quit" onClick="document.location = 'http://www.autodesk.com'" ><p>

3 Save the file.

To test the **Quit** button:

1 Right-click in the Browser window and select **Refresh** from the context menu.

2 Click the **Quit** button in the Browser window to display the Autodesk home page.

3 To go back to the helix application, right-click in the Browser window and select **Back** from the context menu.

Entering helix origin positions

You can change the application to allow you to enter an origin position for the helix by typing a value or clicking a position on the screen.

There are two stages to this:

1 Changing the interface (see page 113).

2 Adding the code (see page 114).

Changing the interface

To change the interface to enable helix origin positions to be entered:

1 Add the following code before the code for the **Apply** button in the HTML file:

\<hr\>

Input origin of the helix\<p\>

\<INPUT TYPE=button VALUE=" Click Point " onClick=point_click()\>

\<INPUT TYPE=button VALUE=" Read Point " onClick=point_read()\>

\<p\>

X \<INPUT TYPE=text NAME=x_text VALUE="0"\> \<p\>

Y \<INPUT TYPE=text NAME=y_text VALUE="0"\> \<p\>

Z \<INPUT TYPE=text NAME=z_text VALUE="0"\> \<p\>

\<hr\>

2 Save the HTML file.

The INPUT command was used previously to create buttons and text boxes. Now you have added two more buttons and three additional text boxes. The \<hr\> code inserts a horizontal line on the page.

Adding the code

You can enter the position for the origin in one of the following ways:

- Click the **Click point** button and enter a position in PowerShape. Then click the **Read point** button to read the coordinates and display them in the **X**, **Y** and **Z** text boxes.

- Enter the coordinates directly into the **X**, **Y** and **Z** text boxes.

Adding the following script to the HTML file provides this functionality.

1 Before the end of the script command **</script>**, add the following procedures.

```
Sub point_click()


//Send command to ask for user point input

pshape.Exec "INPUT POINT 'Click origin' $pos"


End Sub


Sub point_read()


//Extract the position input from the variable $pos

document.helix.x_text.value = pshape.Evaluate("$pos_x")

document.helix.y_text.value = pshape.Evaluate("$pos_y")

document.helix.z_text.value = pshape.Evaluate("$pos_z")


End Sub
```

2 Save the HTML file.

In the first procedure, the code allows you to click points on the screen. Remember the following command from the macro tutorial:

- **INPUT POINT 'Position of centre' $cenpos**

This is used in the application as follows:

- **pshape.Exec "INPUT POINT 'Click origin' $pos"**

where **pshape.Exec** sends the command from the vbscript to PowerShape.

In the second procedure, the next commands are of the form:

- **document.helix.x_text.value = pshape.Evaluate("$pos_x")**

where **pshape.Evaluate** extracts values from objects in PowerShape, in this case, the x coordinate of the input position *$pos*. The value of the coordinate is then entered into the **X** text box using the code: **document.helix.x_text.value**

Updating the Apply_Click procedure

Update the **Apply_Click** procedure to use the values from the X, Y and Z text boxes:

1 Find the following code in the **Apply_Click** procedure:

   //First coordinates of the curve

   pshape.Exec "abs " & document.helix.radius.value & " 0 0"

2 Change it to:

   //First coordinates of the curve

   //pshape.Exec "abs " & document.helix.radius.value & " 0 0"

   start_x =(document.helix.radius.value + 0)+(document.helix.x_text.value + 0)

   pshape.Exec "abs " & start_x & " " & document.helix.y_text.value & " " & document.helix.z_text.value

3 Save the HTML file.

You will notice that we added a zero to some of the variables, for example:

document.helix.rad_text.value

This variable is a string, which represents a number. By adding the zero to the variable, the string is converted into a number and used in the expression.

Instead of removing the following command, we have turned it into a comment by placing // in front of it:

//pshape.Exec "abs " & document.helix.radius.value & " 0 0"

This lets you to use the command again later.

Testing your application again

You are now ready to test your application. Complete the following tests:

- Define the origin of the helix by entering values for X, Y and Z:

  1 Right-click in the Browser window and select **Refresh** from the context menu.

  2 Enter some values for **X**, **Y** and **Z** to define the origin of the helix.

  3 Change the **Radius**, **Pitch** and **Number of turn** values if you want.

  4 Click **Apply**. A helix is created with its origin at the X, Y, Z position that you entered.

- Define the origin of the helix using the mouse:

  1 Change the **Radius**, **Pitch** and **Number of turn** values if you want.

  2 Click the **Click Point** button.

  3 Click a position in the graphics window.

  4 Click the **Read Point** button. This enters the position coordinates into the **X**, **Y** and **Z** text boxes.

  5 Press **Apply**. A helix is created with its origin at the point you selected.

Selecting objects

To extend the application so that it can create a helix around a selected cylinder, you need to add another button to the interface:

1 Add the following code before the **Apply** button in the HTML file:

   Create helix around a cylinder\<p>

   \<INPUT TYPE=button VALUE="Select Cylinder" onClick=cyl_select() >

   \<hr>

2 Save the HTML file.

Boolean variable called cylinder

In some commands, you need to know if a cylinder is selected. You can use a Boolean variable called *cylinder* to indicate if a cylinder is selected. When the program is run, the cylinder variable is set to *false*. When you select a cylinder and use it in the HTML application, the cylinder variable is set to *true*.

To add the Boolean variable called cylinder:

1 At the start of the script, find the following lines:

   // Connect to PowerShape

   set pshape = Window.external

2 After these lines, add the following:

   //No cylinder selected

   cylinder = false

   This sets the cylinder variable to false as soon as you display the HTML file.

3 Save the HTML file.

Adding code for the cyl_select() procedure

The user selects a cylinder and then clicks the **Select cylinder** button.

To add the **cyl_select** procedure called by this button:

1   Before the end of the script command </script>, add the following lines.

```
Sub cyl_select()


//Check if a single cylinder is selected

If pshape.Evaluate("selection.number") = "1" Then

If pshape.Evaluate("selection.object[0].type") = "Cylinder" Then

//Cylinder selected

cylinder = True

End If

End If


If cylinder = False Then

//Tell user that 1 cylinder must be selected

//and exit the procedure

MsgBox ("1 cylinder must be selected!")

Exit Sub

End If


pshape.Exec "Let cyl = selection.object[0]"

//Extract the origin of the cylinder and put in X, Y, and Z boxes

document.helix.x_text.value = pshape.Evaluate("$cyl.origin.x")

document.helix.y_text.value = pshape.Evaluate("$cyl.origin.y")

document.helix.z_text.value = pshape.Evaluate("$cyl.origin.z")

//Extract the radius of the cylinder

document.helix.radius.value = pshape.Evaluate("$cyl.radius")


End Sub
```

2   Save the HTML file.

- The first part of the procedure uses the **pshape.Evaluate** command to check if a single cylinder is selected. This command extracts information from PowerShape. For example, the following extracts the number of objects selected:

  pshape.Evaluate("selection.number")

  If a single cylinder is selected, the cylinder variable is set to *true.* This indicates that a cylinder is selected.

  If a single cylinder is not selected, a message box appears telling the user and the procedure is terminated. The following command terminates the procedure:

  Exit Sub

- The following command assigns the name and identity of the first object in the selection to the variable *cyl* in PowerShape:

  pshape.Exec "Let cyl = selection.object[0]"

- The next set of commands extract the coordinate values from the origin of the cylinder and put the values in the **X**, **Y** and **Z** boxes on the form:

  - **document.helix.x_text.value=pshape.Evaluate("$cyl.origin.x")**

  - **document.helix.y_text.value=pshape.Evaluate("$cyl.origin.y")**

  - **document.helix.z_text.value=pshape.Evaluate("$cyl.origin.z")**

- The command below extracts the radius of the cylinder and enters the value in the **Radius** box on the form:

  document.helix.radius.value = pshape.Evaluate("$cyl.radius")

Temporary workplane

To create the helix in the right direction along the cylinder, you can use a temporary workplane.

In the **Apply_click** procedure, the commands can be updated to:

- Create a temporary workplane (see page 120)

- Input the first point of the helix relative to the temporary workplane (see page 122)

- Delete the temporary workplane (see page 122)

Creating a workplane

To edit the HTML to create a workplane:

1 At the beginning of the **Apply_click** procedure, add the following:

If cylinder = True Then

//create a workplane and modify it

pshape.Exec "create workplane" & vbCrLf _

& "$cyl.origin.x $cyl.origin.y $cyl.origin.z" & vbCrLf _

& "MODIFY" & vbCrLf _

& "NAME tmpwkhelix" & vbCrLf _

& "XAXIS DIRECTION" & vbCrLf _

& "X $cyl.xaxis.x" & vbCrLf _

& "Y $cyl.xaxis.y" & vbCrLf _

& "Z $cyl.xaxis.z" & vbCrLf _

& "ACCEPT" & vbCrLf _

& "YAXIS DIRECTION" & vbCrLf _

& "X $cyl.yaxis.x" & vbCrLf _

& "Y $cyl.yaxis.y" & vbCrLf _

& "Z $cyl.yaxis.z" & vbCrLf _

& "ACCEPT" & vbCrLf _

& "ZAXIS DIRECTION" & vbCrLf _

& "X $cyl.zaxis.x" & vbCrLf _

& "Y $cyl.zaxis.y" & vbCrLf _

& "Z $cyl.zaxis.z" & vbCrLf _

& "ACCEPT" & vbCrLf _

& "ACCEPT"

End If

2 Save the HTML file.

You can then check if the cylinder variable is *true*. This variable is only true if a cylinder is selected and the **Select cylinder** button is clicked. If the cylinder variable is *true*, a workplane is created using the following PowerShape commands from the macro tutorial:

//Creating a temporary workplane

CREATE WORKPLANE

$cyl.origin.x $cyl.origin.y $cyl.origin.z

// Modifying the workplane

MODIFY

NAME tmpwkhelix

XAXIS DIRECTION

X $cyl.xaxis.x

Y $cyl.xaxis.y

Z $cyl.xaxis.z

ACCEPT

YAXIS DIRECTION

X $cyl.yaxis.x

Y $cyl.yaxis.y

Z $cyl.yaxis.z

ACCEPT

ZAXIS DIRECTION

X $cyl.zaxis.x

Y $cyl.zaxis.y

Z $cyl.zaxis.z

ACCEPT

ACCEPT

*When executing PowerShape commands, we use the **pshape.Execute** command. If you have many **pshape.Execute** commands to send, using a single command saves time communicating with PowerShape. In this example, there is only one **pshape.Execute.** To send extra lines of commands with the single **pshape.Execute**, you can use the following syntax:*

pshape.Exec "command line 1" & vbCrLf _

& "command line 2" & vbCrLf _

& "command line 3" & vbCrLf _

& "command line 4"

You cannot include any comments between the lines in the above syntax.

First point relative to workplane

Edit the HTML such that the first coordinate of the helix depends on whether a cylinder is selected:

1 In the **Apply_click** procedure, find the following code for the first coordinate:

//First coordinates of the curve

//pshape.Exec "abs " & document.helix.radius.value & " 0 0"

start_x =(document.helix.radius.value + 0)+(document.helix.x_text.value + 0)

pshape.Exec "abs " & start_x & " " & document.helix.y_text.value & " " & document.helix.z_text.value

2 Change the code to the following:

//First coordinates of the curve

If cylinder = True Then

pshape.Exec "abs " & document.helix.radius.value & " 0 0"

Else

start_x =(document.helix.radius.value + 0)+(document.helix.x_text.value + 0)

pshape.Exec "abs " & start_x & " " & document.helix.y_text.value & " " & document.helix.z_text.value

End If

3 Save the HTML file.

> *If you have selected a cylinder, the helix must start at the coordinates in relation to the temporary workplane. Otherwise, the coordinates must be relative to the coordinates in the* **X, Y** *and* **Z** *boxes.*

Deleting the workplane

Add commands to the **Apply_Click** procedure that will delete the temporary workplane:

1 Find the following code in the **Apply_Click** procedure:

//Exiting curve creation mode

pshape.Exec "Select"

2 Add the following lines after the code:

//Delete the temporary workplane

If cylinder = True Then

```
pshape.Exec "select clearlist"

pshape.Exec "select add workplane 'tmpwkhelix'"

pshape.Exec "delete"

cylinder = False

End If
```

3   Save the HTML file.

When the helix is created, the temporary workplane is deleted and the cylinder variable changes to false. This indicates no cylinder is selected.

We have used the PowerShape commands from the macro tutorial.

Testing the new code

You are now ready to test your application:

1   Save your HTML file.

2   Right-click in the Browser window and select **Refresh** from the context menu.

3   Create a cylinder surface.

4   In PowerShape, select the cylinder.

5   Click the **Select Cylinder** button on the **Helix creation** form.

The **Radius** and the **X**, **Y** and **Z** boxes now contain values from the cylinder.

6   Change the **Pitch** and **Number of turn** values if you want.

7   Click **Apply**. A helix is created around the cylinder.

You have successfully created an application using HTML.

You could further enhance the application by adding, for example:

- tests to check the input data.

- ✓ and ✗ icons to indicate if a cylinder is selected or not.

Example using Javascript

You can use other script languages instead of vbscript.

The final code of the helix example is given below in Javascript:

```
<HTML>
<HEAD>
<script language="javascript">

// Connect to PowerShape
var pshape = window.external;

//No cylinder selected
cylinder = false

function Apply_click()
{
if (cylinder == true)
{
//create a workplane and modify it
pshape.Exec ("create workplane");
pshape.Exec ("$cyl.origin.x $cyl.origin.y $cyl.origin.z");
pshape.Exec ("MODIFY");
pshape.Exec ("NAME tmpwkhelix");
pshape.Exec ("XAXIS DIRECTION");
pshape.Exec ("X $cyl.xaxis.x");
pshape.Exec ("Y $cyl.xaxis.y");
pshape.Exec ("Z $cyl.xaxis.z");
pshape.Exec ("ACCEPT");
pshape.Exec ("YAXIS DIRECTION");
pshape.Exec ("X $cyl.yaxis.x");
pshape.Exec ("Y $cyl.yaxis.y");
pshape.Exec ("Z $cyl.yaxis.z");
```

```
pshape.Exec ("ACCEPT");
pshape.Exec ("ZAXIS DIRECTION");
pshape.Exec ("X $cyl.zaxis.x");
pshape.Exec ("Y $cyl.zaxis.y");
pshape.Exec ("Z $cyl.zaxis.z");
pshape.Exec ("ACCEPT");
pshape.Exec ("ACCEPT")
} //end if



//Calculating values for the coordinates
neg_rad = - document.helix.radius.value;
zheight = document.helix.pitch.value /4;


//Creating the helix's curve
pshape.Exec ("Create curve");
pshape.Exec ("through");


//First coordinates of the curve
if (cylinder == true)
{
pshape.Exec ("abs " + document.helix.radius.value + " 0 0");
} //end if
else
{
start_x =  parseFloat(document.helix.radius.value) +
parseFloat(document.helix.x_text.value);
pshape.Exec ("abs " + start_x + " " + document.helix.y_text.value + " " +
document.helix.z_text.value);
} //end else



//Using a loop to input the coordinates from each turn
Counter = document.helix.turns.value;
```

```
while (Counter > 0)
{
Counter = Counter - 1;
pshape.Exec (neg_rad + " " + document.helix.radius.value + " " +
zheight);
pshape.Exec (neg_rad + " " + neg_rad + " " + zheight);
pshape.Exec (document.helix.radius.value + " " + neg_rad + " " +
zheight);
pshape.Exec (document.helix.radius.value + " " +
document.helix.radius.value + " " + zheight)
} //end while


//Exiting curve creation mode
pshape.Exec ("Select");



//Delete the temporary workplane
if (cylinder == true) {
pshape.Exec ("select clearlist");
pshape.Exec ("select add workplane 'tmpwkhelix'");
pshape.Exec ("delete");
cylinder = false
} //end if


} //end of function apply_click


function point_click()
{
//Send command to ask for user point input
pshape.Exec ("INPUT POINT 'Click origin' $pos")
} // end of function point_click
```

```
function point_read()
{
//Extract the position input from the PowerShape
//variable $pos
document.helix.x_text.value = pshape.Evaluate("$pos_x");
document.helix.y_text.value = pshape.Evaluate("$pos_y");
document.helix.z_text.value = pshape.Evaluate("$pos_z")
} // end of function point_read


function cyl_select()
{
//Check if a single cylinder is selected
if (pshape.Evaluate("selection.number") == "1") {
if (pshape.Evaluate("selection.object[0].type") == "Cylinder")
//Cylinder selected
cylinder = true
}

if (cylinder == false)
{
//Tell user that 1 cylinder must be selected
//and exit the procedure
window.alert ("1 cylinder must be selected!");
return
} //end if

pshape.Exec ("Let cyl = selection.object[0]");
//Extract the origin of the cylinder and put in X, Y, and Z boxes
document.helix.x_text.value = pshape.Evaluate("$cyl.origin.x");
document.helix.y_text.value = pshape.Evaluate("$cyl.origin.y");
document.helix.z_text.value = pshape.Evaluate("$cyl.origin.z");
//Extract the radius of the cylinder
```

```
document.helix.radius.value = pshape.Evaluate("$cyl.radius")

} // end of function cyl_select

</script>

</HEAD>

<BODY>

<h1>Helix creation</h1>

<FORM NAME=helix >

Radius <INPUT TYPE=text NAME=radius VALUE="10" > <p>
Pitch <INPUT TYPE=text NAME=pitch VALUE="4" > <p>
Turns <INPUT TYPE=text NAME=turns VALUE="10" > <p>

<hr>
Input origin of the helix<p>
<INPUT TYPE=button VALUE=" Click Point " onClick="point_click();"
>
<INPUT TYPE=button VALUE=" Read Point " onClick="point_read();"
>

<p>
X <INPUT TYPE=text NAME=x_text VALUE="0"> <p>
Y <INPUT TYPE=text NAME=y_text VALUE="0"> <p>
Z <INPUT TYPE=text NAME=z_text VALUE="0"> <p>

<hr>

Create helix around a cylinder<p>
<INPUT TYPE=button value="Select Cylinder" onClick="cyl_select();" >
```

```
<hr>

<INPUT TYPE=button VALUE=" Apply " onClick="Apply_click();" >
<INPUT TYPE=button VALUE="Quit" onClick="document.location =
'http://www.autodesk.com'" ><p>

</FORM>

</BODY>

</HTML>
```

Creating OLE applications

You can use the PowerShape OLE server to create applications which communicate with PowerShape.

There are two types of OLE applications:

- HTML-based

  An HTML-based application consists of HTML pages and runs in the Browser window in PowerShape. It also communicates commands with PowerShape.

  You can write HTML pages using various HTML or text editors. In the HTML page, you can add scripts using languages such as vbscript and javascript. The OLE commands in the scripts allow you to communicate with PowerShape.

  In our examples for HTML-based applications, we use vbscript. You can download documentation on vbscript from: http://www.microsoft.com.

  The *HTML application tutorial* introduces you to creating HTML-based applications using vbscripts.

- add-in

  An add-in application is a program that enables you to customize and extend the functionality of PowerShape. The OLE commands in the programs allow you to communicate with PowerShape.

  You can write add-in applications using programming languages, such as Microsoft Visual Basic and Microsoft Visual C++. The OLE commands in the programs enable the add-ins to communicate with PowerShape.

These applications allow you to:

- perform commonly used operations
- create easy-to-use interfaces

Both types of applications use the same OLE commands. The following sections use HTML examples.

A help file is provided for the OLE COM functions at:

C:\Program Files\Autodesk\PowerShapexxxxx\file\ole\help.chm.

Connecting to PowerShape using HTML

Before you can use the PowerShape OLE server, you must connect to a PowerShape session.

You can connect to an existing PowerShape session, but how you connect to PowerShape will depend on whether your application is HTML-based or an add-in.

For HTML-based applications, use:

set pshape = window.external

In vbscript, this would look as follows:

<script language="vbscript" >

set pshape = window.external

...

...

...

</script>

For add-in applications, use:

Set pshape = Getobject(,"PowerShape.Application")

Both methods create the object *pshape,* which is connected to an existing PowerShape session.

With these methods, when you close the add-in applications, PowerShape remains open.

Sending commands to PowerShape

The following method sends commands to the connected PowerShape session:

pshape.Exec Command

where *Command* is a string expression containing a macro (see page 7) command to run in PowerShape.

For example:

When the command button **cmdCreateLine** is clicked, a single line is produced between the coordinates entered in four text boxes *txtX1, txtY1, txtX2, txtY2*.

'When the command button is clicked....

Sub cmdCreateLine_Click()

```
'Set PowerShape into single line mode
pshape.Exec "CREATE LINE SINGLE"
'Enter the origin of the line
pshape.Exec txtX1.Text & " " & txtY1.Text
'Enter the incremental move required
pshape.Exec (txtX2.Text - txtX1.Text) & _
" " & (txtY2.Text - txtY1.Text)
'Set PowerShape back to select mode
pshape.Exec "SELECT"
End Sub
```

You can split a command into two lines by using an underscore character "_" as a separator. For example, the following commands:

```
pshape.Exec (txtX2.Text - txtX1.Text) & _
" " & (txtY2.Text - txtY1.Text)
```

are the same as the command:

```
pshape.Exec (txtX2.Text - txtX1.Text) & " " & (txtY2.Text - txtY1.Text)
```

Getting information from PowerShape

If you can print the value of something in PowerShape, you can also extract its value using the Evaluate command. The server returns a VARIANT variable, which means the result can be a number, a string, or a vector (an array of numbers).

To use the Evaluate method, the syntax is:

```
V = pshape.Evaluate(value_string)
```

where *value_string* is a string containing the object you require the information on.

For example, you can use the following to extract the number of selected objects:

```
V = pshape.Evaluate("selection.number")
```

For a list of strings for each object, see PowerShape object information (see page 143).

Getting information about a model

Use the **pshape.activedocument** method to get information about an open model.

This method is assigned to an object using the following commands:

- **Dim psmodel As Object**

- **Set psmodel = pshape.activedocument**

When you set this object, it is associated with the current active model. You can use this object to check whether the model is active or editable using the following properties:

- **psmodel.active**

- **psmodel.editable**

If the model associated with *psmodel* is active, then the active property will return true, otherwise it will return false. Similarly, if the model is editable, then the editable property will return true, otherwise it will return false.

> *All PowerShape commands automatically operate on the active model and some commands fail if Editable is false.*

For example:

You can restrict the commands in your add-in application to one model. The active document method allows you to observe a model, you can then check if the model is active.

```vbscript
<script language = "vbscript">
set pshape = window.external
Set psmodel = pshape.activedocument

Private Sub Apply_Click()
  If psmodel.active Then
    If psmodel.editable Then
      MsgBox ("Model editable!")
    Else
      MsgBox ("Model not editable!")
    End If
  Else
    MsgBox ("Original model not active!")
  End If

End Sub

</script>
```

Showing and hiding the PowerShape window

To show the PowerShape window:

pshape.Visible = True

To hide the PowerShape window:

pshape.Visible = False

Controlling the PowerShape window

You can control the PowerShape window using the **WindowState** property. It enables you to specify the following states:

- minimise
- maximise
- normalised
- bring to foreground

pshape.WindowState = value

You can input *value* as a number from the following table:

Value	Description
1	This is the state when you can resize and position the window.
2	Maximise window.
4	Minimise window to the taskbar.
8	Bring window to the foreground.

For example:

The following minimizes the PowerShape window, performs some commands and then maximizes the window again.

<script language="VBscript">

Set pshape = window.external

// This minimise the PowerShape window,

// carries out some commands, and

// maximises the window again

Sub Minimise_Click()

pshape.windowstate = 1

//Carry out some commands

...

...

...

pshape.windowstate = 2

End Sub

</script>

How do I find the version number of PowerShape?

The **pshape.Version** property returns a string containing the version number of PowerShape that your application is currently connected to:

If you are not connected to PowerShape, an error is returned.

How do I know if PowerShape is busy?

The **pshape.busy** property checks if the connected PowerShape session is busy:

If PowerShape is busy, this property returns **True**, otherwise it returns **False**.

PowerShape is registered as *busy* when:

▪ the **Import** or **Export** dialogs are open

▪ the **Print** dialog is open

▪ any PowerShape dialog is displayed

This property is useful when waiting for a user to input a position in an add-in application. For an example, see Add-in example using Visual Basic (see page 135).

Add-in example using Visual Basic

This example waits for a point input in PowerShape after clicking a button called **cmdIndicate**. It then extracts its coordinates into three text boxes: *txtX, txtY, and txtZ.*

<script language="VBscript">

Set pshape = window.external

Private Sub cmdIndicate_Click()

'Send command to ask for user point input

'While waiting for point input, PowerShape

'will be registered as Busy

pshape.Exec "INPUT POINT 'Click Origin' $pos"

'Wait until point has been input

Do

Loop Until pshape.Busy = False

'Extract the position input from the PowerShape

'variable $pos which was used

txtX.Text = pshape.Evaluate("$pos_x")

txtY.Text = pshape.Evaluate("$pos_y")

txtZ.Text = pshape.Evaluate("$pos_Z")

End Sub

</script>

If the **do…loop** is not included, the program does not wait until the point is entered. This would result in trying to extract values that are not set. Try removing this loop to see what happens.

Show and hide dialogs, or suspend graphics during commands

To access some functions, such as changing the name of an arc, you need to display the **Arc** dialog. However, when the application uses the OLE server, you want to access the functions within it, you do not normally want to display the dialog.

The following commands control the display of the user interface and dialogs, and control graphics refreshes, as operations are carried out:

- Use **ShowForms** property to hide and display the dialogs when sending OLE commands.

  pshape.ShowForms = False

  turns off the PowerShape interface updates until the state of the property is changed.

  pshape.ShowForms = True

  restarts the display of dialogs.

- Use the following commands for one-off control of display of toolbars:

  FORMUPDATE

  updates the interface to current state. The state of **ShowForms** is unchanged.

  FORMUPDATE ON

  restarts the display of dialogs (same as *ShowForms=True*).

  FORMUPDATE OFF

stops the display of dialogs (same as *ShowForms = False*).

- Use the following commands for one-off control of the display of the dialogs. The state of **ShowForms** is unchanged:

DIALOG ON

displays the dialog.

DIALOG OFF

hides the dialog.

- Use the following commands to suspend graphics:

REFRESH OFF

suspends graphics while operations are carried out, until **REFRESH ON** is executed

REFRESH ON

restarts the graphics

REFRESH ON FORCE

overrides stacked **REFRESH OFF** commands, to force graphics to restart

How do I exit PowerShape using my application?

Use the **pshape.exit** command to exit the PowerShape session you are connected to.

No confirmation dialog is displayed before PowerShape quits.

Entering positions using OLE application

You can click positions in PowerShape and then read the position data into your application.

Use the **INPUT POINT** command from the PowerShape macro language in a **pshape.Exec** command.

For example,

pshape.Exec "INPUT POINT 'Click origin' $pos"

When the position is clicked, its coordinates are assigned to the following variables: *$pos_x, $pos_y and $pos_z*.

You can access these variables using **pshape.Evaluate** command.

Example using vbscript:

Sub point_click()

//Send command to ask for user point input

pshape.Exec "INPUT POINT 'Click origin' $pos"

End Sub

Sub point_read()

//Extract the position input from the PowerShape

//variable $pos

document.helix.x_text.value = pshape.Evaluate("$pos_x")

document.helix.y_text.value = pshape.Evaluate("$pos_y")

document.helix.z_text.value = pshape.Evaluate("$pos_z")

End Sub

You cannot combine these commands into one procedure. If you do, the following happens when you use the application.

- When the user clicks the position, the application continues to the next command line without receiving the *$pos* data from PowerShape.

- If you pause the application using the **pshape.busy** property, it will loop.

Selecting objects

This topic describes how to use selected objects in your application.

You can use the following methods to select objects for use:

- Before it is run.

  As soon as the application is run, you can use the *selection* object information to interrogate the selection and then operate on the selection.

- While it is running.

  You need some method of telling the application that the objects are selected. One way is to add a button to the application. When you have selected the required objects, click the button to complete the selection. You can then use the *selection* object information (see page 143) to interrogate the selection and operate on the selection.

For example:

Private Sub Cmd_cyl_Click()

'Check if a single cylinder is selected

If pshape.Evaluate("selection.number") = "1" Then

If pshape.Evaluate("selection.object[0].type") = "Cylinder" Then

```
'Cylinder selected
cylinder = True
End If
Else
'Tell user that 1 cylinder must be selected
'and exit the procedure
MsgBox ("1 cylinder must be selected!")
Exit Sub
End If
pshape.Exec "Let cyl = selection.object[0]"
'Extract the origin of the cylinder and put in X, Y, and Z boxes
Txt_x.Text = pshape.Evaluate("$cyl.origin.x")
Txt_y.Text = pshape.Evaluate("$cyl.origin.y")
Txt_z.Text = pshape.Evaluate("$cyl.origin.z")
End Sub
```

Tips and tricks

Each command in an application communicates with PowerShape using the Windows interpreter. Therefore, running each command results in a short delay. If you have many PowerShape commands, this delay can last a few seconds.

To minimize the delay where you have a block of PowerShape commands, use a single execute command. Each line of commands must be separated by a special character.

You can type the **pshape.Exec** command as follows:

```
pshape.Exec "command line 1" & vbCrLf _
& "command line 2" & vbCrLf _
& "command line 3" & vbCrLf _
& "command line 4" & vbCrLf _
& "command line 5" & vbCrLf _
& "command line 6"
```

Another way to reduce the time taken is to create a macro containing the block of commands. You can then run the macro in an **Exec** command.

Running a HTML-based application

To run an HTML-based application in the PowerShape browser window:

1 Start PowerShape.

2 Double-click the Command box in the status bar to open the Command window.

3 In the Command window, type:

browser explorer {path_of_html_file}

where **{path_of_html_file}** is the path to the file. This displays the HTML file in the browser window in PowerShape.

You can then use the HTML-based application in PowerShape.

Running an add-in application

When you have created or downloaded an add-in application, you can run it inside or outside PowerShape. When you run your application, it begins executing its commands.

To run your application outside PowerShape, do one of the following:

▪ Use the **Run** command from the **Start** menu.

▪ Double click your application's icon in **Windows Explorer**.

You can also add shortcuts to your application. For further details see the Microsoft Windows Help.

To run your application inside PowerShape, you can use the Add-in Manager to create a link to your application. This enables you to run your application from within PowerShape.

For more information, see Working with add-in applications in PowerShape (see page 140).

Working with add-in applications in PowerShape

To add an add-in application:

1 Select Home tab > Add-Ins panel > Add-ins > Manage. The **Add-In Manager** dialog is displayed.

2 Click the **Add** button on the dialog.

A new item, called **Add-in**, is added to the list. This item is highlighted, ready for you to change its name.

3 In the **Title** box, type the name of the item. This name appears in the Home tab > Add-Ins panel > Add-ins menu.

4 In the **Command** box, type the path where the application is stored. Alternatively, click the **Browse** ⬚ button to display the **Open** dialog and search for your application .

5 In the **Arguments** box, type any start-up parameters for the application.

6 In the **Start in** box, type the path where you want your application to run.

7 Click **Apply** to save your changes. This adds the item to the list of available macros.

8 Repeat steps 2 - 8 to add more applications.

9 Click **Cancel** to close the dialog.

Add-in applications are only available to the person who added them.

To run an add-in application:

1 Click Home tab > Add-Ins panel > Add-ins.

2 Select the application from the menu.

To change the name of an Add-in:

1 Select Home tab > Add-Ins panel > Add-ins > Manage. The **Add-In Manager** dialog is displayed.

2 Select the Add-in application in the **Menu contents** list.

3 Edit the **Title**.

4 Click **Apply** to change the name.

5 Click **Cancel** to close the dialog.

To re-order the Add-in applications:

1 Select Home tab > Add-Ins panel > Add-ins > Manage. The **Add-In Manager** dialog is displayed.

2 Select the item you want to move.

3 Use the **Move Item Up** ⬆ and **Move Item Down** ⬇ buttons to reposition the selected item.

4 Click **Apply** to change the order.

5 Click **Cancel** to close the dialog.

To remove an Add-in application:

1 Select Home tab > Add-Ins panel > Add-ins > Manage. The **Add-In Manager** dialog is displayed.

2 Select the item you want to delete.

3 Click the **Delete** ✖ button.

4 Click **Apply** to delete the item.

5 Click **Cancel** to close the dialog.

Object information

You can access information about objects using special macro commands. These commands help you identify precisely which feature of an object you wish to retrieve and investigate.

For example, the command to access the start coordinates of a line is:

line[*name*].start

This retrieves the start coordinates [x, y, z] of the line called *name*.

For the x coordinate of the start position of this line, the syntax is:

line[*name*].start.x

In the syntax, *name* appears (in italics) as **object[*name*]**. This is the name of the object on the left of the square bracket [ ].

Sometimes, *name* appears more than once as:

object1[*name*].object2[*name*].

name of object 1 does not necessarily equal *name* of object 2.

> *PowerShape allocates a unique identity number to each object. You can substitute the* name *of an object for its unique identity number. For example, you can use either:*
>
> **line[id 75].start.x,** *where 75 is the unique identity number of the line.*
>
> *or*
>
> **line[1].start.x,** *where 1 is the name of the line.*

You can find information on the different commands for the following types of objects:

Arc (page 145)	Level (page 178)	Surface (page 207)
Application paths (page 147)	Line (page 178)	Symbol (page 221)
Assembly (page 148)	Mesh (see page 180)	Text (page 223)
Clipboard (page 156)	Mesh Doctor (page 181)	Tolerance (page 225)
Cloud (page 156)	Model (page 182)	Units (page 225)

Arc commands

The following arc commands are available:

Command	Description
arc[*name*].exists	*1* if arc exists; *0* otherwise.
arc[*name*].id	unique identity number of the arc in the model.
arc[id *n*].name	name of the arc that has the given identity number.
arc[*name*].start	coordinates [x, y, z] of the start position of the arc.
arc[*name*].start.x	x coordinate of the start position of the arc.
arc[*name*].start.y	y coordinate of the start position of the arc.
arc[*name*].start.z	z coordinate of the start position of the arc.
arc[*name*].end	coordinates [x, y, z] of the end position of the arc.
arc[*name*].end.x	x coordinate of the end position of the arc.
arc[*name*].end.y	y coordinate of the end position of the arc.
arc[*name*].end.z	z coordinate of the end position of the arc.
arc[*name*].mid	coordinates [x, y, z] of the mid position of the arc.
arc[*name*].mid.x	x coordinate of the mid position of the arc.
arc[*name*].mid.y	y coordinate of the mid position of the arc.
arc[*name*].mid.z	z coordinate of the mid position of the arc.
arc[*name*].radius	radius value of the arc.
arc[*name*].centre	coordinates [x, y, z] of the centre position of the arc.
arc[*name*].centre.x	x coordinate of the centre position of the arc.

arc[*name*].centre.y	y coordinate of the centre position of the arc.
arc[*name*].centre.z	z coordinate of the centre position of the arc.
arc[*name*].length	length of the circumference of the arc.
arc[*name*].centre_mark	the centre mark type. For each type of centre marker, the standard number is given below:

- *0* for none
- *1* for dot
- *2* for cross

arc[*name*].start_angle	start angle of the arc.
arc[*name*].end_angle	end angle of the arc.
arc[*name*].span_angle	span angle of the arc.
arc[*name*].style.colour	colour number of line style used to draw the arc.
arc[*name*].style.color	color (USA) number of line style used to draw the arc.
arc[*name*].style.gap	gap of line style used to draw the arc.
arc[*name*].style.weight	weight of line style used to draw the arc.
arc[*name*].style.width	width of line style used to draw the arc.
arc[*name*].level	level on which the arc exists.

Application paths command

Command	Description
app.paths	displays path information for some of the directories that PowerShape uses, for example:
	Program : C:\Program Files\Autodesk\PowerShapexxxxx\sys\exec\powershape.exe
	Document : C:\Program Files\Autodesk\PSDocxxxxx\help
	Pre-config macro : C:\Program Files\Autodesk\PowerShapexxxxx\lib\macro\preconfig.mac
	Post-config macro : C:\Program Files\Autodesk\PowerShapexxxxx\lib\macro\postconfig.mac
	Login macro : C:\Program Files\Autodesk\PowerShapexxxxx\lib\macro\login.mac
	Temp : C:\Documents and Settings\xxx\Local Settings\Temp
	Shareddb : C:\Documents and Settings\All Users\Shared Documents\Autodesk\shareddb
	Parts : C:\Documents and Settings\All Users\Shared Documents\Autodesk\parts
	Local config : C:\Documents and Settings\xxx\Application Data\PowerShape\
	Home : C:\Documents and Settings\xxx\Application Data\

Assembly commands

Command	Description
comassembly component "*c_name*" **property set** "*name" "value"*	set/change value of property.
comassembly component "*c_name*" **property remove** "*name*" "*value*"	remove property.
comassembly component "*c_name*" **property remove all**	remove all properties.
comassembly definition *defn_name* **thumbnail_view_dir** *direction*	sets the view for the thumbnail that is displayed in the component library window. where: *defn_*name is the name of the component definition *direction* is a view direction. This may have the following values: **top** **bottom** **right** **left** **front** **back** **iso1** **iso2** **iso3** **iso4**
comassembly component *c_name* **property list**	print list of properties and their values.
component ["*c_name*"].**property**["*name*"].**value**	check value of property.
component ["*c_name*"].**property**["*name*"].**exists**	check if the property is present. Returns 1 if the component exists, 0 if it does not exist.
comassembly definition ["*c_name*"] **property list**	print list of properties of component definition and their values.
comassembly component_defn["*cd_name"*].**property**["*name"*].**exists**	The command returns *1* if the component exists. *0* if it does not exist.

comassembly definitions imported refresh	refreshes imported definitions. The command is only available when a model is open.
COMASSEMBLY DEFINITION "definition name" HIDE_IN_LIBRARY **COMASSEMBLY DEFINITION "definition name" SHOW_IN_LIBRARY**	Hide/display the component definitions in the component library window.

Relationships commands

Command	Description
relationship['"assembly_name" "relation_name"].exists	1 if relationship exists; 0 otherwise.
relationship['"assembly_name" "relation_name"].gen_type	returns the type of the relationship. 0 — plane/plane 1 — point to point 2 — plane/point 3 — point/plane 4 — line/line 5 — line/point 6 — point/line 7 — plane/line 8 — line/plane
relationship['"assembly_name" "relation_name"].add_type	returns additional type of the relationship.
relationship['"assembly_name" "relation_name"].distance	distance value.
relationship['"assembly_name" "relation_name"].alignment	the alignment of the relationship.
relationship['"assembly_name" "relation_name"].attachment_master	master attachment name of the relationship.
relationship['"assembly_name" "relation_name"].attachment_slave	slave attachment name of the relationship.
relationship['"assembly_name" "relation_name"].component_master	master component name of the relationship.
relationship['"assembly_name" "relation_name"].component_slave	slave component name of the relationship
relationship['"assembly_name" "relation_name"].is_broken	1 if relationship is broken; 0 otherwise.
relationship['"assembly_name" "relation_name"].has_distance	1 if the relationship has a distance parameter; 0 otherwise.
relationship['"assembly_name" "relation_name"].tree_name	tree browser name of the relationship.

Attachment commands

Command	Description
attachment[name].exists	1 if exists, 0 otherwise.
attachment[name].point	returns point of given attachment
attachment[name].vector	vector of the given attachment
attachment[name].is_default	1 if true, 0 if false.

External attachments on component definitions commands

comassembly create plane_attachment $*attachment_name* $*posx* $*posy* $*posz* $*vecx* $*vecy* $*vecz* **on definition** $*def_name*

comassembly create plane_attachment $*attachment_name* **$posx** $*posy* $*posz* $*vecx* $*vecy* $*vecz* **on instance** $*inst_name*

comassembly create line_attachment $*attachment_name* **$posx $posy** $*posz* $*vecx* $*vecy* $*vecz* **on definition** $*def_name*

comassembly create line_attachment $*attachment_name* **$posx $posy** $*posz* $*vecx* $*vecy* $*vecz* **on instance** $*inst_name*

comassembly create point_attachment $*attachment_name* $*posx* $*posy* $*posz* **on definition** $*def_name*

comassembly create point_attachment $*attachment_name* $*posx* $*posy* $*posz* **on instance** $*inst_name*

Component commands

Command	Description
component[*name*].min_range_w	minimum range of the component with respect to the world workplane.
component[*name*].max_range_w	maximum range of the component with respect to the world workplane.
component[*name*].min_range	minimum range of the component with respect to the active workplane.
component[*name*].max_range	maximum range of the component with respect to the active workplane.
component[*name*].size	size of the component.
component[*name*].exists	*1* if component exists; *0* otherwise.
component[*name*].level	level value of component.
component[*name*].status	status of component *0* — free state *1* — undefined *2* — fully defined *3* — over-defined *4* — error position

Parameter commands

Command	Description
parameter[name].expression	parameter expression.
parameter[name].dimension	parameter dimension.
parameter[name].dep_items	item(s) dependent on the parameter.
parameter[name].hidden	value of the HIDDEN flag.
parameter[name].expfl	value of the EXPRESSION flag.
parameter[name].main	value of the MAIN flag.
parameter[name].automatic	value of the AUTOMATIC flag.
parameter.number	number of non-hidden and non-automatic parameters in the model. This is the number of entries in the drop down list in the Parameter Editor dialog.

Component definitions commands

Command	Description
component_defn[name].exists	1 if component definition exists, 0 otherwise
component_defn[name].num_components	number of components using the component definition
component_defn[name].is_active	1 if the component definition is an active assembly; 0 otherwise
component_defn[name].is sub_assembly	1 if the component definition is a sub-assembly; 0 otherwise
component_defn[name].num_poi_attachments	number of point attachments.
component_defn[name].num_lin_attachments	number of linear attachments.
component_defn[name].num_pla_attachments	number of plane attachments.
component_defn[name].is_imported	1 if the component definition is imported; 0 otherwise.
component_defn[name].is_model_defn	1 if component definition is a model component definition, 0 otherwise.
component_defn[name].num_solids	returns number of solids.
component_defn[name].num_axis_attachments	number of axis attachments.
component_defn[name].num_attachments	number of attachments.
component_defn[name].is_parametric	1 if component definition is parametric; 0 otherwise.
component_defn ['assembly_name'].cog	returns the centre of gravity of the assembly
component_defn ['component_name'].cog	returns the centre of gravity of the component.
preserve_params on	preserves the global parameters when registering a component definition.

component_defn["name "].attachment["name "].surface...	where ….. can be any property of a surface. For example: **print component_defn["***name***"].attachment["***name***"].surface.name** **print component_defn["***name***"].attachment["***name***"].surface.area**

Power Features commands

Command	Description
component_defn[*assembly_name***].pfsummary.source[***source path***].feature[***feature_name***].target[***target_path***].exists**	returns the stored power features summary data for required source, feature, target.
component_defn[*assembly_name***].pfsummary.source[***source path***].feature[***feature_name***].target[***target_path***].flag**	returns the value of power features summary flag for required source, feature, target.

TU-coordinates commands

Command	Description
comassembly insert attachment linked_by_tu ["*name of defn***"] ["***name of attachment***"] ["***surface's name***"]/surface ID POINT/PLANE t-value u-value**	inserts new attachment linked to surface by tu-coordinate.
component_defn[*name***].attachment[***name***].is_linked_by_tu**	1 if attachment is linked to surface by tu-coordinates; 0 otherwise.
component_defn["*name***"].attachment["***name***"].t**	get t-value stored in attachment.
component_defn["*name***"].attachment["***name***"].u**	get u-value stored in attachment.

Tool Solid command

Command	Description
solid['*ToolSolid*'].hide	1 if the solid is owned by another item and not displayed; 0 if the solid is hidden.

Clipboard command

Command	Description
clipboard.valid	*1* if there is something on the clipboard; *0* otherwise.

Cloud commands

Command	Description
cloud[name].level	returns the level that the specified cloud is on.
cloud[name].exists	1 if the cloud exists, 0 otherwise.
cloud[name].id	unique identity number of the cloud in the model.
cloud[id n].name	name of the cloud that has the given identity number n.
cloud[*name*].style.colour	colour number of line style used to draw the cloud.
cloud[*name*].style.color	color (USA) number of line style used to draw the cloud.
cloud[*name*].style.gap	gap of line style used to draw the cloud.
cloud[*name*].style.weight	weight of line style used to draw the cloud.
cloud[*name*].style.width	width of line style used to draw the cloud.

Composite curve commands

Commands for composite curves can take the following forms:

- **compcurve[*name*]......**
- **composite curve[*name*].....**

To avoid duplication, the format **compcurve[*name*]** is used throughout.

Command	Description
compcurve[*name*].exists	*1* is the composite curve exists; *0* otherwise.
compcurve[*name*].id	unique identity number of the composite curve in the model.
compcurve[id *n*].name	name of the composite curve that has the given identity number.
compcurve[*name*].description	the description of the curve is stored in the database.
compcurve[*name*].closed	*1* if the composite curve is closed; *0* otherwise.
compcurve[*name*].item.number	number of items that make up the composite curve.
compcurve[*name*].length	length of the composite curve.
compcurve[*name*].length_between(*a; b*)	length along the composite curve between key points a and b.
compcurve[*name*].area	area of the composite curve. If the composite curve is: - closed and planar, the area is the enclosed area. - open, it is closed with a straight line for the area measurement. - non-planar, PowerShape tries to construct a plane from the first few items. If this fails, the current principal plane is used. The composite curve is projected onto the plane and the area is measured from the projected curve.
compcurve[*name*]	fillets the composite curve, where *name* is the name of the composite curve.
compcurve[*name*].level	level on which the composite curve exists.

Points in composite curve commands

Command	Description
compcurve[*name*].point.number	number of points in the composite curve.
compcurve[*name*].point[*number*]	coordinates [x, y, z] of the composite curve's point.
compcurve[*name*].point[*number*].x	x coordinate of the composite curve's point.
compcurve[*name*].point[*number*].y	y coordinate of the composite curve's point.
compcurve[*name*].point[*number*].z	z coordinate of the composite curve's point.
compcurve[*name*].point[*number*].entry_tangent	unit vector of the tangent direction entering the point.
compcurve[*name*].point[*number*].entry_tangent.x	x value of the unit vector that defines the tangent direction entering the point.
compcurve[*name*].point[*number*].entry_tangent.y	y value of the unit vector that defines the tangent direction entering the point.
compcurve[*name*].point[*number*].entry_tangent.z	z value of the unit vector that defines the tangent direction entering the point.
compcurve[*name*].point[*number*].exit_tangent	unit vector of the tangent direction leaving the point.
compcurve[*name*].point[*number*].exit_tangent.x	x value of the unit vector that defines the tangent direction leaving the point.
compcurve[*name*].point[*number*].exit_tangent.y	y value of the unit vector that defines the tangent direction leaving the point.
compcurve[*name*].point[*number*].exit_tangent.z	z value of the unit vector that defines the tangent direction leaving the point.
compcurve[*name*].point[*number*].entry_tangent.azimuth	azimuth angle of the tangent entering the point.
compcurve[*name*].point[*number*].entry_tangent.elevation	elevation angle of the tangent entering the point.
compcurve[*name*].point[*number*].exit_tangent.azimuth	azimuth angle of the tangent leaving the point.

Command	Description
compcurve[*name*].point[*number*].exit_tangent.elevation	elevation angle of the tangent leaving the point.
compcurve[*name*].point[*number*].entry_magnitude	magnitude entering the point.
compcurve[*name*].point[*number*].exit_magnitude	magnitude leaving the point.

Bounding box around composite curve commands

Command	Description
compcurve[*name*].size	size of the bounding box around the composite curve.
compcurve[*name*].size.x	size in the x direction of the bounding box around the composite curve.
compcurve[*name*].size.y	size in the y direction of the bounding box around the composite curve.
compcurve[*name*].size.z	size in the z direction of the bounding box around the composite curve.
compcurve[*name*].min_range	minimum coordinates of the bounding box around the composite curve.
compcurve[*name*].min_range.x	x coordinate of the minimum coordinates of the bounding box around the composite curve.
compcurve[*name*].min_range.y	y coordinate of the minimum coordinates of the bounding box around the composite curve.
compcurve[*name*].min_range.z	z coordinate of the minimum coordinates of the bounding box around the composite curve.
compcurve[*name*].max_range	maximum coordinates of the bounding box around the composite curve.
compcurve[*name*].max_range.x	x coordinate of the maximum coordinates of the bounding box around the composite curve.

Command	Description
compcurve[*name*].max_range.y	y coordinate of the maximum coordinates of the bounding box around the composite curve.
compcurve[*name*].max_range.z	z coordinate of the maximum coordinates of the bounding box around the composite curve.

Centre of gravity of composite curve commands

Command	Description
compcurve[*name*].cog	coordinates [x, y, z] of the centre of gravity of the composite curve.
compcurve[*name*].cog.x	x coordinate of the centre of gravity of the composite curve.
compcurve[*name*].cog.y	y coordinate of the centre of gravity of the composite curve.
compcurve[*name*].cog.z	z coordinate of the centre of gravity of the composite curve.

Style of composite curve commands

Command	Description
compcurve[*name*].style.colour	colour number of line style used to draw the composite curve.
compcurve[*name*].style.color	color (USA) number of line style used to draw the composite curve.
compcurve[*name*].style.gap	gap of line style used to draw the composite curve.
compcurve[*name*].style.weight	weight of line style used to draw the composite curve.
compcurve[*name*].style.width	width of line style used to draw the composite curve.

Created commands

Use this group of commands to query which objects were created as a result of the last operation. These objects are accessed from the creation list.

Command	Description
created.exists	*1* if at least one item is in the creation list; *0* otherwise.
created.number	number of items in the creation list.
created.clearlist	clears the creation list.
created.object[*number***]**	object type and its name in the creation list. For example, *Line[4]*, *Arc[1]*. If *n* items are created, then *number* is the item's number in the creation list.
created.object[*number***].***syntax*	object information as specified by the *syntax* for object created.object[*number*]. The *syntax* you can use is given under each type of object. For example, if **created.object[1]** is *Line[2]*, then you can specify the *syntax* as any syntax after **Line[name]**. For further details see Line (see page 178) . For the x coordinate of the start of the line, you can use **created.object[1].start.x** where **start.x** is the syntax.

created.type[*number***]**	type of an object in the creation list. For example, *Line*, *Arc*.
	If *n* objects are created, then *number* is the item's number in the creation list. *number* is from *0* to *(n-1)*.
	When you compare the type of an object with a text string, you must use the correct capitalisation. For example, to check that created.type[0] is a composite curve, you must use:
	created.type[0] == 'Composite Curve'
	and not:
	created.type[0] == 'Composite curve'
	created.type[0] == 'composite curve'
created.name[*number***]**	name of an item in the creation list.
	If *n* items are created, then *number* is the item's number in the creation list.
	In all cases, *number* is from 0 to (n-1).

Curve commands

Command	Description
curve[*name***].exists**	*1* if curve exists; *0* otherwise.
curve[*name***].id**	unique identity number of the curve in the model.
curve[id ***n*].name**	name of the curve that has the given identity number.
curve[*name***].description**	the description of the curve is stored in the database.
curve[*name***].type**	checks the curve and returns one of the following strings:
	▪ *Bézier*
	▪ *Bspline*
curve[*name***].number**	number of points in the curve.
curve[*name***].closed**	*1* if the curve is closed; *0* otherwise.

curve[*name*].start	start coordinates [x, y, z] of the curve.
curve[*name*].start.x	x coordinate of the start of the curve.
curve[*name*].start.y	y coordinate of the start of the curve.
curve[*name*].start.z	z coordinate of the start of the curve.
curve[*name*].end	end coordinates [x, y, z] of the curve.
curve[*name*].end.x	x coordinate of the end of the curve.
curve[*name*].end.y	y coordinate of the end of the curve.
curve[*name*].end.z	z coordinate of the end of the curve.
curve[*name*].length	length of the curve.
curve[*name*].length_between(*a; b*)	length along the curve between key points a and b.
curve[*name*].area	area of the curve. If the curve is: ■ closed and planar, the area is the enclosed area. ■ open, it is closed with a straight line for the area measurement. ■ non-planar, the curve is projected onto the current principal plane and the area is measured from the projected curve.
curve[*name*].size	size of the bounding box around the curve.
curve[*name*].size.x	size in the x direction of the bounding box around the curve.
curve[*name*].size.y	size in the y direction of the bounding box around the curve.
curve[*name*].size.z	size in the z direction of the bounding box around the curve.
curve[*name*].min_range	minimum coordinates of the bounding box around the curve.
curve[*name*].min_range.x	x coordinate of the minimum coordinates of the bounding box around the curve.

curve[*name*].min_range.y	y coordinate of the minimum coordinates of the bounding box around the curve.
curve[*name*].min_range.z	z coordinate of the minimum coordinates of the bounding box around the curve.
curve[*name*].max_range	maximum coordinates of the bounding box around the curve.
curve[*name*].max_range.x	x coordinate of the maximum coordinates of the bounding box around the curve.
curve[*name*].max_range.y	y coordinate of the maximum coordinates of the bounding box around the curve.
curve[*name*].max_range.z	z coordinate of the maximum coordinates of the bounding box around the curve.
curve[*name*].cog	coordinates [x, y, z] of the centre of gravity of the curve.
curve[*name*].cog.x	x coordinate of the centre of gravity of the curve.
curve[*name*].cog.y	y coordinate of the centre of gravity of the curve.
curve[*name*].cog.z	z coordinate of the centre of gravity of the curve.
curve[*name*].style.colour	colour number of line style used to draw the curve.
curve[*name*].style.color	color (USA) number of line style used to draw the curve.
curve[*name*].style.gap	gap of line style used to draw the curve.
curve[*name*].style.weight	weight of line style used to draw the curve.
curve[*name*].style.width	width of line style used to draw the curve.
curve[*name*].level	level on which the curve exists.

Points in a curve commands

Command	Description
curve[*name*].point[*number*]	coordinates [x, y, z] of the point.
curve[*name*].point[*number*].x	x coordinate of the point.
curve[*name*].point[*number*].y	y coordinate of the point.
curve[*name*].point[*number*].z	z coordinate of the point.
curve[*name*].point[*number*].selected	*1* if the point is selected; *0* otherwise.
curve[*name*].point[*number*].dependent	*1* if the point is dependent; *0* otherwise.
curve[*name*].point[*number*].entry_tangent	unit vector of the tangent direction entering the point.
curve[*name*].point[*number*].entry_tangent.x	x value of the unit vector which defines the tangent direction entering the point.
curve[*name*].point[*number*].entry_tangent.y	y value of the unit vector which defines the tangent direction entering the point.
curve[*name*].point[*number*].entry_tangent.z	z value of the unit vector which defines the tangent direction entering the point.
curve[*name*].point[*number*].exit_tangent	unit vector of the tangent direction leaving the point.
curve[*name*].point[*number*].exit_tangent.x	x value of the unit vector which defines the tangent direction leaving the point.
curve[*name*].point[*number*].exit_tangent.y	y value of the unit vector which defines the tangent direction leaving the point.
curve[*name*].point[*number*].exit_tangent.z	z value of the unit vector which defines the tangent direction leaving the point.
curve.selected.points	Returns the number of currently selected points on a wireframe curve (an INT).
compcurve.selected.points	Returns the number of currently selected points on a wireframe composite curve (an INT).

Command	Description
curve[*name*].point[*number*].entry_tang ent.azimuth	azimuth angle of the tangent entering the point.
curve[*name*].point[*number*].entry_tang ent.elevation	elevation angle of the tangent entering the point.
curve[*name*].point[*number*].exit_tange nt.azimuth	azimuth angle of the tangent leaving the point.
curve[*name*].point[*number*].exit_tange nt.elevation	elevation angle of the tangent leaving the point.
curve[*name*].point[*number*].entry_mag nitude	magnitude entering the curve's point.
curve[*name*].point[*number*].exit_magni tude	magnitude leaving the curve's point.

Dimension commands

The following groups of dimension commands are available:

Command	Description
dimension[*name*].exists	*1* if dimension exists; *0* otherwise.
dimension[*name*].id	unique identity number of the dimension in the model.
dimension[id *n*].name	name of the dimension that has the given identity number.
dimension[*name*].value	value of dimension.
dimension[*name*].diameter	*1* if the dimension measures a diameter; *0* otherwise.
dimension[*name*].leader.style	style name of the leader of the dimension
dimension[*name*].leader.trim	*1* if the option **Trim leader to text** is on; *0* otherwise. The **Trim leader to text** option trims the leader to the position of the dimension annotation.

dimension[*name*].leader.keep	*1* if the option **Internal leader on small dimensions** is on; *0* otherwise.
	When you have a dimension with leaders placed on either side of the dimension, the **Internal leader on small dimensions** option adds a line so that no gap exists between the arrows of the leader.
dimension[*name*].leader.marksize	size of the mark on the leader of the dimension.
dimension[*name*].leader.marktype	standard number indicating the type of marker. For each type of marker, the standard number is given below.
	▪ Dot — *1*
	▪ Slash — *10*
	▪ Cross — *5*
	▪ Filled circle — *11*
	▪ Circle — *4*
	▪ Filled arrow — *9*
	▪ Arrow — *8*
dimension[*name*].annotation.style	style name of the annotation of the dimension.
dimension[*name*].annotation.height	height of the annotation.
dimension[*name*].annotation.embed	*1* if the annotation is embedded; *0* otherwise.
dimension[*name*].annotation.horizontal	*1* if the annotation is set to horizontal; *0* otherwise.
dimension[*name*].annotation.proportional	*1* if the annotation is set to proportional; *0* otherwise.
dimension[*name*].annotation.italic	*1* if the annotation is set to italic; *0* otherwise.
dimension[*name*].annotation.gap	gap between the text and the leader.
dimension[*name*].annotation.fraction	*1* if decimal part of the dimension is set to a fraction; *0* otherwise.
dimension[*name*].annotation.denom	denominator of the fraction.

dimension[*name*].annotation.angleformat	number to indicate the type of angle format. For each type of angle format, the number is given below: ▪ *1* — Decimal ▪ *2* — Degrees ▪ *3* — Degrees - Minutes ▪ *4* — Degrees - Minutes - Seconds
dimension[*name*].annotation.decimal	number of decimal places of the dimension.
dimension[*name*].witness.style	style name of the witness line of the dimension.
dimension[*name*].tolerance.style	style name of the tolerance of the dimension.
dimension[*name*].tolerance.value1	value 1 of the tolerance range.
dimension[*name*].tolerance.value2	value 2 of the tolerance range.
dimension[*name*].tolerance.height	height of the tolerance text.
dimension[*name*].tolerance.alignment	number to indicate the type of tolerance alignment. For each type of tolerance alignment, the number is given below: ▪ 1 — alignment 75.00 +0.15 -0.05 ▪ 2 — alignment 75.00 ±0.10 ▪ 3 — alignment 75.15 74.95
dimension[*name*].tolerance.decimal	number of decimal places of the tolerance.
dimension[*name*].style.colour	colour number of line style used to draw the dimension.
dimension[*name*].style.color	color (USA) number of line style used to draw the dimension.
dimension[*name*].style.gap	gap of line style used to draw the dimension.
dimension[*name*].style.weight	weight of line style used to draw the dimension.
dimension[*name*].style.width	width of line style used to draw the dimension.
dimension[*name*].level	level on which the dimension exists.

There are also commands to retrieve information on the position of the dimension (see page 169).

Position of the dimension commands

A dimension is specified by its text position and various other positions, depending on the dimension type. The text position (*position.text*) is at the centre of the text. There are three other possible positions: *position.one*, *position.two* and *position.three*.

A linear dimension is specified as shown below. It has a text position (*position.text*), *position.one* and *position.two*
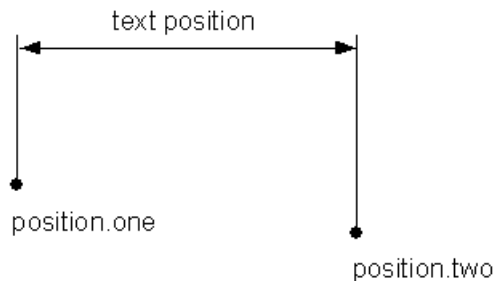
An angular dimension has a text position (*position.text*), *position.one*, *position.two* and *position.three*.
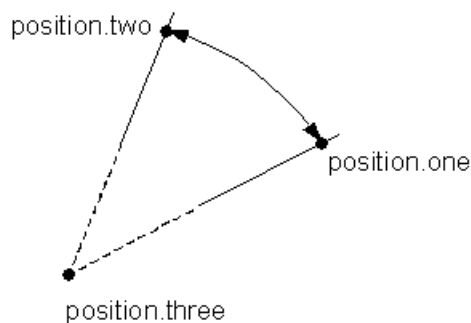
A radial dimension has a text position (*position.text*), *position.one*, *position.two* and *position.three*.
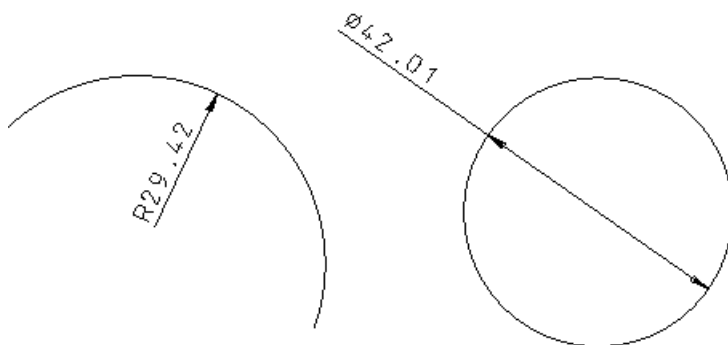
Command	Description
dimension[*name*].position.text	coordinates [x, y, z] of the position of the text of the dimension.
dimension[*name*].position.one	coordinates [x, y, z] of position.one of the dimension.

dimension[_name_**].position.two**	coordinates [x, y, z] of position.two of the dimension.
dimension[_name_**].position.three**	coordinates [x, y, z] of position.three of the dimension.

Drawing commands

The following drawing commands are available:

Command	Description
drawing[_name_**].exists**	1 if drawing exists; 0 otherwise.
drawing[_name_**].description**	description of the drawing.
drawing.number	the number of drawings.
drawing[_name_**].id**	unique identity number of the drawing.
drawing[id _n_**].name**	name of the drawing that has the given identity number.
drawing.name[_index_**]**	returns a drawing name where _index_ is greater than 0 and less than or equal to the number of drawings.
drawing[_name_**].width**	width of the drawing.
drawing[_name_**].height**	height of the drawing.
drawing[_name_**].template_model**	name of the model containing the template_drawing used by the drawing.
drawing[_name_**].template_drawing**	name of the template drawing used by the drawing.
drawing[_name_**].tmpl_model_invalid**	_1_ if the model, containing the template drawing used by the drawing, exists in the database; _0_ otherwise.
drawing[_name_**].tmpl_drawing_invalid**	_1_ if the template drawing, used by the drawing, exists; _0_ otherwise.
drawing[_name_**].views**	number of views on the drawing.
drawing[_name_**].view.name[**_N_**]**	name of the _N_th view on the drawing, where 0 < N <= number of views.
drawing[_name_**].no_of_items**	number of objects on the drawing.
drawing[name].view[view_name].needs updating	_1_ if the view needs updating; _0_ otherwise.

Drawing view commands

Drawing view commands are only available in conjunction with Drawing commands as indicated:

Command	Description
drawing[*name***].view[***name***].xmin_extent**	x coordinate of the minimum extent of the view on the drawing.
drawing[*name***].view[***name***].xmax_extent**	x coordinate of the maximum extent of the view on the drawing.
drawing[*name***].view[***name***].ymin_extent**	y coordinate of the minimum extent of the view on the drawing.
drawing[*name***].view[***name***].ymax_extent**	y coordinate of the maximum extent of the view on the drawing.
drawing[*name***].view[***name***].scale**	scale of the view.
drawing[*name***].view[***name***].origin**	coordinates [x, y, z] of the origin of the view.
drawing[*name***].view[***name***].no_of_items**	number of objects in the view.
drawing[*name***].view[***name***].transform[***number***]**	the elements of the rotation matrix and the translation vector of the view in relation to the model space. The value of number determines the elements: ▪ 0, 1, 2 specifies the elements of the first row of the rotation matrix. ▪ 4, 5, 6 specifies the elements of the second row of the rotation matrix. ▪ 7, 8, 9 specifies the elements of the third row of the rotation matrix. ▪ 12, 13, 14 specifies elements of the translation vector.
drawing[*drawing_name***].view[***view_name***].drawing_to_view[***x ; y ; z***]** **drawing[***drawing_name***].view[***view_name***].drawing_to_world[***x ; y ; z***]** **drawing[***drawing_name***].view[***view_name***].view_to_drawing[***x ; y ; z***]** **drawing[***drawing_name***].view[***view_name***].world_to_drawing[***x ; y ; z***]**	Use these variables to convert between drawing, view and world space.

drawing[*drawing_name***].view[***view_name*** **e].drawing_to_view[***x ; y ; z***].x**	Use X/Y/Z modifiers with the above variables to return the x-ordinate of the converted point.

Electrode commands

The following electrode commands are available:

Command	Description
electrode.list **electrode.list.all**	list of all electrode names.
electrode.list.originals	list of electrode names, excluding copies.
electrode.list.copies	list of electrode names of electrode copies.
electrode[*name***].datum**	coordinates [x, y, z] of the origin of the electrode's datum.
electrode[*name***].datum.x**	x coordinate of the origin of the electrode's datum.
electrode[*name***].datum.y**	y coordinate of the origin of the electrode's datum.
electrode[*name***].datum.z**	z coordinate of the origin of the electrode's datum.
electrode[*name***].blank.name**	name of the electrode's blank.
electrode[*name***].blank.rectangular**	*1* if the blank is rectangular; *0* if it is circular.
electrode[*name***].blank.length**	length of the electrode's blank.
electrode[*name***].blank.width**	width of the electrode's blank.
electrode[*name***].blank.diameter**	diameter of the electrode's blank.
electrode[*name***].blank.height**	height of the electrode's blank.
electrode[*name***].blank.material**	material of the electrode's blank.
electrode[*name***].holder.catalogue**	name of holder catalogue.
electrode[*name***].holder.base**	name of base holder.
electrode[*name***].holder.edm**	name of additional EDM holder.
electrode[*name***].holder.machining**	name of additional Machining holder.
electrode[*name***].holder.<base|machinin g|edm>.items** **electrode[***name***].holder.<base|machinin g|edm>.item.number**	number of items that make up the base, machining or edm holder.

electrode[*name*].holder.<base|machining|edm>.item(n) **electrode[*name*].holder.<base|machining|edm>.item(n).name**	name of *n*th item that makes up base, machining or edm holder.
electrode[*name*].holder.<base|machining|edm>.item(n).id	ID of nth item that makes up base, machining or edm holder.
electrode[*name*].holder.<base|machining|edm>.item(n).type	type of *n*th item that makes up base, machining or edm holder (Solid or Symbol).
electrode[*name*].burn_region.surfaces	Returns the number of surfaces in an electrode's burn region.
electrode[*name*].burn_region.attached	Checks if a new burn region has been attached to an electrode. Returns 1 for electrodes that are part of a multi-impression burn.
electrode[*name*].burn_region.surface[*n*]	Zero-indexed access to the surfaces in an electrodes burn region. Normal surface attributes can be accessed, for example: **electrode[*name*].burn_region.surface[*0*].id.**
electrode[*name*].quantity.rough	the number of roughers in the electrode family.
electrode[*name*].quantity.semi	the number of semi-finishers in the electrode family.
electrode[*name*].quantity.finish	the number of finishers in the electrode family.
electrode[*name*].undersize.rough	the undersize of the rougher (in the current units).
electrode[*name*].undersize.semi	the undersize of the semi-finisher (in the current units).
electrode[*name*].undersize.finish	the undersize of the finisher (in the current units).
electrode[*name*].frame.exists	returns *1* if the electrode has a frame; *0* otherwise
electrode[*name*].frame.length	returns the length of electrode frame.
electrode[*name*].frame.width	returns the width of electrode frame.
electrode[*name*].frame.height	returns the height of electrode frame.

electrode[*name*].frame.has_chamfer	returns *1* if the electrode frame has a chamfer; *0* otherwise.
electrode[*name*].frame.chamfer_size	returns the size of chamfer on the electrode frame.

There are also General (see page 174) Electrode commands.

> *In some electrode commands, you can specify the name of the electrode. For example: **electrode[*name*].exists.***
>
> *In these commands, you can also enter* index n*, where n is the nth electrode created in the model.*
>
> *For example, the following expression checks if the first electrode exists: **electrode[*index 1*].exists.***

General Electrode commands

Command	Description
electrode.number	number of electrodes in the model.
electrode[*name*].exists	*1* if the electrode exists; *0* otherwise.
electrode[*name*].id	identity number of the electrode in the model.
electrode[id *n*].name	name of the electrode that has the given identity number.
electrode[*name*].level	level on which the electrode exists.
electrode[*name*].rotation	the rotation of the electrode from the workplane of the electrode.
electrode[*name*].sparkgap	spark gap of the electrode.
electrode[*name*].burn_depth	distance in z from the bottom of the electrode to the top of the burn region.
electrode[*name*].surface_finish	the surface finish selected on the electrode family page of the wizard for that electrode.
electrode[*projected_area*]	area of the burn region as projected onto the XY plane.

electrode[*name*].solid.solid_attributes	attributes of the solid depending on the value of *solid_attributes*. For example: **electrode[*name*].solid.volume** volume of the solid. For a complete list of attributes, see Solid commands (see page 197).
electrode[*name*].base_height	height of the base of the electrode can be defined using the variable.
electrode[*name*].active_solid	the solid that the electrode was extracted from.
electrode[*name*].active_workplane	the workplane that was active when the electrode was created. (These are only available for electrodes that are extracted, not those that are copied).
electrode[*name*].fillins **electrode[*name*].fillin.number**	number of fill-in surfaces associated with this electrode.
electrode[*name*].fillin(*n*) **electrode[*name*].fillin(*n*).name**	name of *n*th fillin surface for this electrode (n starts at 1).
electrode[*name*].fillin(*n*).id	ID of *n*th fillin surface for this electrode.
electrode[*name*].details(1)	the first additional description field for the electrode. By default this is the "Job No." entry.
electrode[*name*].details(2)	the second additional description field for the electrode. By default this is the "Works Order" entry.
electrode[*name*].details(3)	the third additional description field for the electrode. By default this is the "Description" entry.
electrode.number.all	number of all electrodes.
electrode.number.originals	number of electrodes, excluding copies.
electrode.number.copies	number of electrode copies.
electrode[*name*].is_copy	*1* if an electrode is a copy. *0* if not a copy.
electrode[*name*].parent	the name of parent if the electrode is a copy.
electrode[*name*].copies	the number of copies of this electrode.

electrode[*name***].angle.a**	angle of rotation of the extraction vector in XY.
electrode[*name***].angle.b**	angle from the vertical.
electrode[*name***].angle.c**	rotation around the vector defined by **a** and **b**.
electrode[*name***].burn_vector**	vector representing the extraction direction.
electrode[*name***].vector_clearance**	distance the electrode is cleared from the part along the burn vector before it is moved in Z.

File commands

Command	Description
file move file "*pathname_from***" "***pathname_to***"**	move a file to another location.
file copy file "*pathname_from***" "***pathname_to***"**	copy a file to another location.
file move dir "*pathname_from***] [***pathname_to***"**	move a directory to another location.
file copy dir "*pathname_from***" "***pathname_to***"**	copy a directory to another location.
file create dir "*pathname***"**	create a new directory.
file[*name***].exists**	*1* if file exists; *0* otherwise.
file[*name***].readable**	*1* if file is readable; *0* otherwise.
file[*name***].writeable**	*1* if file is writeable; *0* otherwise.
file[*name***].size**	returns file size in bytes.
file[*name***].mode**	*0* if file does not exists *1* if file *2* if directory.
directory[*name***].exists**	*1* if directory exists; *0* otherwise.
directory[*name***].readable**	*1* if directory is readable; *0* otherwise.
directory[*name***].writeable**	*1* if directory is writeable; 0 otherwise.
directory[*name***].mode**	*0* is directory does not exists *1* if file *2* if directory.
directory['pathname'].files['pattern']	returns a list of files in a directory.

Hatch commands

The following hatch commands are available:

Command	Description
hatch[*name*].exists	*1* if drawing exists; *0* otherwise.
hatch[*name*].id	unique identity number of the hatch in the model.
hatch[id *n*].name	name of the hatch that has the given identity number.
hatch[*name*].cross	*1* if hatch is crossed; *0* otherwise.
hatch[*name*].fill	*1* if hatch is filled; *0* otherwise.
hatch[*name*].angle	first angle of hatch.
hatch[*name*].angle1	first angle of hatch.
hatch[*name*].angle2	second angle of hatch.
hatch[*name*].spacing	spacing of hatch.
hatch[*name*].boundaries	number of boundaries enclosing the hatch.
hatch[*name*].style.colour	colour number of line style used to draw the hatch.
hatch[*name*].style.color	color (USA) number of line style used to draw the hatch.
hatch[*name*].style.gap	gap of line style used to draw the hatch.
hatch[*name*].style.weight	weight of line style used to draw the hatch.
hatch[*name*].style.width	width of line style used to draw the hatch.
hatch[*name*].level	level on which the hatch exists.

Language command

Command	Description
print language.summary	the language and system locale settings being used by PowerShape.

Level commands

Command	Description
level.number	the number of used levels.
level[*number***].used**	*1* if the level is used; *0* otherwise.
level[id *n*].name	name of the level that has the given identity number.
level[*number***].active**	*1* if the level is on; *0* otherwise.
level.filtered.number	number of filtered levels.
level.filtered[n].index	level number for the nth filtered level, where n is an integer between 0 to (level.filtered.number)-1.
level.filtered.used	*1* if the **used** filter is set; *0* otherwise.
level.filtered.named	*1* if the **named** filter is set; *0* otherwise.
level.filtered.on	*1* if the **on** filter is set; *0* otherwise.

Line commands

The following line commands are available:

Command	Description
line[*name***].start**	start coordinates [x, y, z] of the line.
line[*name***].start.x**	x coordinate of the start of the line.
line[*name***].start.y**	y coordinate of the start of the line.
line[*name***].start.z**	z coordinate of the start of the line.
line[*name***].end**	end coordinates [x, y, z] of the line.
line[*name***].end.x**	x coordinate of the end of the line.
line[*name***].end.y**	y coordinate of the end of the line.
line[*name***].end.z**	z coordinate of the end of the line.
line[*name***].exists**	*1* if line exists; *0* otherwise.
line[*name***].id**	unique identity number of the line in the model.
line[id *n*].name	name of the line that has the given identity number.
line[*name***].length**	length of the line.
line[*name***].style.colour**	colour number of line style used to draw the line.

line[*name***].style.color**	color (USA) number of line style used to draw the line.
line[*name***].style.gap**	gap of line style used to draw the line.
line[*name***].style.width**	width of line style used to draw the line.
line[*name***].level**	level on which the line exists.
item[*name***].style.pattern.start_mark** **item[***name***].style.pattern.end_mark**	name of the mark type at the start or end of the leader line item.

There are also commands to retrieve information on the angles of a line (see page 179).

Angles of a line commands

Use the following variables to find the apparent and elevation angles of a line (these are the same values shown on the line editing form).

The commands return an absolute value, with the angle in the current units: degrees or radians.

Command	Description
line[*xxx***].apparent**	returns the apparent angle of the line using the current working plane of the currently active workspace
line[*xxx***].elevation**	returns the angle of elevation that the line makes using the current principal plane of the currently active workspace.

Optionally, you can specify which principal plane to use:

- **line[***xxx***].apparent.xy**
- **line[***xxx***].apparent.yz**
- **line[***xxx***].apparent.zx**
- **line[***xxx***].elevation.xy**
- **line[***xxx***].elevation.yz**
- **line[***xxx***].elevation.zx**

Mesh commands

Command	Description
mesh[*name*].level	returns the level that the specified mesh is on.
mesh[*name*].exists	1 if the mesh exists, 0 otherwise.
mesh[*name*].id	unique identity number of the mesh in the model.
mesh[id *n*].name	name of the mesh that has the given identity number n.
mesh[*name*].style.colour	colour number of line style used to draw the mesh.
mesh[*name*].style.color	color (USA) number of line style used to draw the mesh.
mesh[*name*].style.gap	gap of line style used to draw the mesh.
mesh[*name*].style.weight	weight of line style used to draw the mesh.
mesh[*name*].style.width	width of line style used to draw the mesh.
mesh[*name*].area	area of mesh.
mesh[*name*].volume	volume of mesh.

Mesh Doctor commands

mesh_doctor can be used instead of **meshdoctor** for the following commands.

Command	Description
meshdoctor.stage	prints the current stage: 1,2,3 or 4.
meshdoctor.[*fault group***].[***fault.state***].number**	returns the total number of faults of the given group, in the given state.
meshdoctor.[*fault group***].[***fault.state***].num_selected**	returns the total number of faults of the given group, in the given state, that are currently selected.
meshdoctor.[*fault group***].[***fault.state***][***n***].[***info***]**	reports information on the nth fault of the given group, in the given state.
meshdoctor.[*fault***].available_fixes.number**	returns the total number of available fixes for the given fault type.
meshdoctor.[*fault***].available_fixes[***n***].description**	returns a description of the nth fix available for the given fault type.
meshdoctor.[*fault***].available_fixes[***n***].name**	returns the name of the nth fix available for the given fault type.
meshdoctor.[*fault***].[***fault.state***].number**	returns the total number of faults of the given type, that are in the given state.
meshdoctor.[*fault***].[***fault.state***].num_selected**	returns the number of selected faults of the given type, that are in the given state.
meshdoctor.[*fault***].[***fault.state***][***n***].[***info***]**	reports information on the nth fault of the given type, that is in the given state.
meshdoctor.[*tolerance.name***].tolerance**	returns the tolerance value.
meshdoctor [*tolerance.name***] tolerance [***number***]**	sets the specified tolerance to the value given by *number*. This command can be called also outside the **Mesh Doctor** if a mesh is selected.

[*fault group***]** can be: *all* (all faults), or *topological* (anything that is not a gap, partial gap, hole or intersection).

[*fault.state*] can be: *current_faults* (the currently detected faults), *fixed_faults* (faults fixed since the wizard was started), or s*elected_faults* (the currently selected faults).

[*info*] can be: *selected* (*1* if the item is selected; *0* if not), *visible* (*1* if the item is visible; *0* if not), *details* (name of the item), *fixes.number* (total number of possible fixes for this fault), *fixes*[*n*].*name* (name of nth fix for this fault), or fixes[*n*].*description* (description of the nth fix for this fault).

[*fault*] can be: *duplicate_node/vertex* (two nodes or vertices closer than tolerance), *impossible_edge* (an edge with more than 2 triangles attached), *reversed_edge* (an edge with triangles attached that have different orientations), *gap* (a narrow hole, or a hole with a narrow section), *hole*, or *intersection* (overlapping triangles).

[*tolerance.name*] can be: *gap_width* (the maximum width for a narrow hole to be considered a gap), *split_edge* (a number >=0 and <1 that is used by the algorithm that classifies narrow holes into gaps or holes), *zero_area* (triangles whose area is smaller than this value are considered a fault), *hole_fill* (the grid spacing to use when filling a hole), or *vertex_distance* (vertices that are closer than this value are considered duplicate).

Model commands

The following groups of model commands are available:

Command	Description
model.selected	name of the selected model.
model[*name***].selected**	*1* if the named model is selected; *0* otherwise.
model[*name***].exists**	*1* if the named model exists; *0* otherwise.
model[*name***].id**	unique identity number of the model.
model[id *n*].name	name of the model that has the given identity number.
model[*name***].open**	*1* if the named model is open; *0* otherwise.

model.lines model.arcs model.curves model.compcurves model.surfaces model.solids model.workplanes model.dimensions model.hatches model.symbols model.texts model.pcurves model.boundaries model.components	number of objects in model
model.filesize	the size (in bytes) of the selected model's database.
model[*name*].filesize	the size (in bytes) of the named model's database. Note, if the model is not open, **model[*name*].filesize** is assigned -1. The collective size of the model's files in its directory will be about 500 bytes larger. The size can be even larger if untruncated files exist. Use File > Info > Compress Model to truncate files as well as reducing the actual database size too.
model[*name*].open.read	*1* if the named model has read access; *0* otherwise.
model[*name*].open.write	*1* if the named model has write access; *0* otherwise.
model.path	pathname of the currently selected model.
model[*name*].path	pathname of the named model For example, **model[mouse].path** returns the pathname C:\Users\Public\Documents\Autodesk\Parts\m142.
model.locked	*1* if the currently selected model is locked; *0* otherwise.
model[*name*].locked	*1* if the named model is locked; *0* otherwise.
model.changed	*1* if the currently selected model has changed; *0* otherwise.

model[*name*].changed	*1* if the named model has changed; *0* otherwise.
model.corrupt	*1* if the currently selected model is corrupted; *0* otherwise.
model[*name*].corrupt	*1* if the named model is corrupted; *0* otherwise.
model.file_doctor.all	number of errors found for general attributes, trimming, arcs and names.
model.file_doctor.gen_attributes	number of errors found for general attributes.
model.file_doctor.deps	number of errors found for dependencies.
model.file_doctor.trimming	number of errors found for trimming.
model.file_doctor.arcs	number of errors found for arcs.
model.file_doctor.names	number of errors found for names.
model.file_doctor.solids	returns the number of errors found by the File Doctor solid checker.
model.file_doctor.orphans	returns the number of errors found by the File Doctor orphaned items checker.
model.version	current model version
model.previous_version	version of model prior to upgrade when the model was opened
model.upgraded	*1* if the model was upgraded on opening, *0* otherwise.

Nesting

The following nesting commands are available:

Command	Description
nestingex.num_parts	number of unique Parts.
nestingex.num_sheets	number of unique Sheets.
nestingex.item[*object name***].exists**	1 if Part or Sheet with the given name exists; 0 if not. For example: **nestingex.item[**"Solid 12"**].exists**
nestingex.item[*object name***].type**	whether the item with the given name is a Part or a Sheet. For example: **nestingex.item[**"Curve 14"**].type**
nestingex.part[*n number***].name**	name of the nth Part. For example: **nestingex.part[**n 4**].name**
nestingex.sheet[*object name***].material_usage**	material usage of the Sheet with the given name. For example: **nestingex.sheet[**"Arc 20"**].material_usage**
nestingex.sheet[*object name***].number**	number of instances of the Sheet with the given name. For example: **nestingex.sheet[**"Curve 24"**].number**
nestingex.part[*n number***].num_colliding**	number of instances of the nth Part which are in collision. For example: **nestingex.part[**n 0**].num_colliding**
nestingex.part[*object name***].instance[***n number***].selected**	whether the nth instance of the Part with the given name is selected. For example: **nestingex.part[**"Solid 5"**].instance[**n 3**].selected**
nestingex.part[*n number***].instance[***n number***].colliding**	whether the nth instance of the nth Part is in collision. For example: **nestingex.part[**n 0**].instance[**n 3**].colliding**

Parameter command

Command	Description
parameter[*name*].value	value of parameter.
parameter[*name*].exists	1 if parameter exists; 0 otherwise.
parameter[*name*].id	unique identity number of the parameter in the model.
parameter[id *n*].name	name of the parameter that has the given identity number.
parameter.number	returns the number of non-hidden and non-automatic parameters in the model. This is the number of entries in the drop down list in the **Parameter Editor** dialog.

Pcurve command

Command	Description
pcurve[*name*].exists	*1* if pcurve exists; *0* otherwise.
pcurve[*name*].number	number of points in the pcurve.
pcurve[*name*].level	level on which the pcurve exists.
pcurve[*name*].closed	*1* if the pcurve is closed; *0* otherwise.
pcurve[*name*].id	unique identity number of the pcurve in the model.
pcurve[id n].name	name of the pcurve that has the given identity number.
pcurve[*name*].edge	*1* if the pcurve is on the edge of a surface; *0* otherwise.
pcurve[*name*].parent.name	name of the surface on which the pcurve lies.
pcurve[*name*].parent.id	unique identification number of the surface on which the pcurve lies.
pcurve[*name*].in_boundary	*1* if the pcurve exists in any boundary; *0* otherwise.
pcurve[*name*].start	coordinates [x, y, z] of the start position in the pcurve.
pcurve[*name*].start.xyz	coordinates [x, y, z] of the start position in the pcurve.

pcurve[*name*].start.x	x coordinate of the start position in the pcurve.
pcurve[*name*].start.y	y coordinate of the start position in the pcurve.
pcurve[*name*].start.z	z coordinate of the start position in the pcurve.
pcurve[*name*].start.tu	tu coordinates [t, u, 0] of the start position in the pcurve.
pcurve[*name*].start.t	t coordinate of the start position in the pcurve.
pcurve[*name*].start.u	u coordinate of the start position in the pcurve.
pcurve[*name*].start.exists	1 if the start coordinates of the pcurve exists; 0 otherwise.
pcurve[*name*].end	coordinates [x, y, z] of the end position in the pcurve.
pcurve[*name*].end.xyz	coordinates [x, y, z] of the end position in the pcurve.
pcurve[*name*].end.x	x coordinate of the end position in the pcurve.
pcurve[*name*].end.y	y coordinate of the end position of the pcurve.
pcurve[*name*].end.z	z coordinate of the end position in the pcurve.
pcurve[*name*].end.tu	tu coordinates [t, u, 0] of the end position in the pcurve.
pcurve[*name*].end.t	t coordinate of the end position in the pcurve.
pcurve[*name*].end.u	u coordinate of the end position in the pcurve.
pcurve[*name*].end.exists	1 if the end coordinates of the pcurve exists; 0 otherwise.
pcurve[*name*].point[*number*]	coordinates [x, y, z] of the pcurve's point.
pcurve[*name*].point[*number*].xyz	coordinates [x, y, z] of the pcurve's point.
pcurve[*name*].point[*number*].x	x coordinate of the pcurve's point.
pcurve[*name*].point[*number*].y	y coordinate of the pcurve's point.
pcurve[*name*].point[*number*].z	z coordinate of the pcurve's point.

Command	Description
pcurve[*name*].point[*number*].tu	tu coordinates [t, u, 0] of the pcurve's point.
pcurve[*name*].point[*number*].t	t coordinate of the pcurve's point.
pcurve[*name*].point[*number*].u	u coordinate of the pcurve's point.
pcurve[*name*].point[*number*].exists	1 if the pcurve's point exists; 0 otherwise.

Point commands

Command	Description
point[*name*].exists	*1* if the point exists; *0* otherwise.
point[*name*].id	unique identity number of the point in the model.
point[id *n*].name	name of the point that has the given identity number.
point[*name*].description	description of the point as stored in the database.
point[*name*].position	coordinates [x, y, z] of the point.
point[*name*].position.x	x coordinate of the point.
point[*name*].position.y	y coordinate of the point.
point[*name*].position.z	z coordinate of the point.
point[*name*].style.colour	colour number of line style used to draw the point.
point[*name*].style.color	color (USA) number of line style used to draw the point.
point[*name*].style.gap	gap of line style used to draw the point.
point[*name*].style.weight	weight of line style used to draw the point.
point[*name*].style.width	width of line style used to draw the point.
point[*name*].level	level on which the point exists.

Printer commands

Command	Description
printer[*name*].exists	*1* if the printer exists; *0* otherwise.
printer[*name*].id	unique identity number of the printer.
printer[id *n*].name	name of the printer that has the given identity number.
printer[*name*].image_string_set	*1* if the image command is set; 0 otherwise.
printer[*name*].image_string	image command for this printer.
printer[*name*].plot_string_set	*1* if the plot command is set; *0* otherwise.
printer[*name*].plot_string	plot command for this printer.
printer[*name*].initialised	1 if the printer is initialised; 0 otherwise.
printer[*name*].num_pens	number of pens stored for this printer.
printer[*name*].pen[*n*].colour	colour number of pen n on this printer.
printer[*name*].pen[*n*].width	width of pen n on this printer.
printer[*name*].pen[*n*].active	*1* if pen n is active; *0* otherwise.

Renderer commands

Command	Description
renderer.has_hardware_triangles	*1* if the hardware supports triangles; *0* otherwise.
renderer.has_depth_cueing	*1* if the hardware supports depth cueing; *0* otherwise.
renderer.has_anti_aliasing	*1* if the hardware supports anti-aliasing; *0* otherwise.

Selection commands

Command	Description
selection.exists	*1* if at least one item is selected; *0* otherwise.
selection.id	unique identity number of the selection in the model.
selection.number **selection.magnitude**	number of selected items.
selection[*name***].description**	description of the selection as stored in the database.
SELECTION.TYPES	returns a list of strings such as { 'Line'; 'Arc'; 'Solid'};one string per selected item.
SELECTION.NAMES	returns a list of strings such as { '1'; '1'; 'fred' };one string per selected item.
selection.object[*number***]**	object type and its name in the selection. For example, *Line[4]*, *Arc[1]*. If *n* items are selected, *number* is the item's number in the selection.
selection.object[*number***].***syntax***	object information as specified by the *syntax* for object selection.object[*number*]. The *syntax* is given under each type of object. For example, if **selection.object[1]** is *Line[2]*, then you can specify the *syntax* as any syntax after Line[*name*]. For further details, see Line (see page 178). For the x coordinate of the start of the line, you can use **selection.object[1].start.x** where *start.x* is the syntax.

selection.type[*number*]	type of an object in the selection. For example, *Line*, *Arc*.
	If *n* objects are selected, *number* is the item's number in the selection.
	If you compare the type of an object with a text string, you must use the correct capitalisation. For example, to check that selection.type[0] is a composite curve, use:
	selection.type[0] == 'Composite Curve'
	not:
	selection.type[0] == 'Composite curve'
	selection.type[0] == 'composite curve'
selection.name[*number*]	name of an item in the selection.
	If *n* items are selected, then *number* is the item's number in the selection.
	In all cases, *number* is from 0 to (n-1).
selection.size	size of the bounding box around the selection.
selection.size.x	size in the x direction of the bounding box around the selection.
selection.size.y	size in the y direction of the bounding box around the selection.
selection.size.z	size in the z direction of the bounding box around the selection.
selection.min_range	minimum coordinates of the bounding box around the selection.
selection.min_range.x	x coordinate of the minimum coordinates of the bounding box around the selection.
selection.min_range.y	y coordinate of the minimum coordinates of the bounding box around the selection.
selection.min_range.z	z coordinate of the minimum coordinates of the bounding box around the selection.
selection.max_range	maximum coordinates of the bounding box around the selection.
selection.max_range.x	x coordinate of the maximum coordinates of the bounding box around the selection.

selection.max_range.y	y coordinate of the maximum coordinates of the bounding box around the selection.
selection.max_range.z	z coordinate of the maximum coordinates of the bounding box around the selection.
selection.min_range_exact	minimum coordinates of the bounding box around the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.min_range_exact.x	x coordinate of the minimum coordinates of the bounding box around the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.min_range_exact.y	y coordinate of the minimum coordinates of the bounding box around the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.min_range_exact.z	z coordinate of the minimum coordinates of the bounding box around the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.max_range_exact	maximum coordinates of the bounding box around the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.max_range_exact.x	x coordinate of the maximum coordinates of the bounding box around the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.

selection.max_range_exact.y	y coordinate of the maximum coordinates of the bounding box around the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.max_range_exact.z	z coordinate of the maximum coordinates of the bounding box around the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.size[*n*]	size of the bounding box around the nth object in the selection.
selection.size[*n*].x	size in the x direction of the bounding box around the nth object in the selection.
selection.size[*n*].y	size in the y direction of the bounding box around the nth object in the selection.
selection.size[*n*].z	size in the z direction of the bounding box around the nth object in the selection.
selection.min_range[*n*]	minimum coordinates of the bounding box around the nth object in the selection.
selection.min_range[*n*].x	x coordinate of the minimum coordinates of the bounding box around the nth object in the selection.
selection.min_range[*n*].y	y coordinate of the minimum coordinates of the bounding box around the nth object in the selection.
selection.min_range[*n*].z	z coordinate of the minimum coordinates of the bounding box around the nth object in the selection.
selection.max_range[*n*]	maximum coordinates of the bounding box around the nth object in the selection.
selection.max_range[*n*].x	x coordinate of the maximum coordinates of the bounding box around the nth object in the selection.

selection.max_range[*n*].y	y coordinate of the maximum coordinates of the bounding box around the nth object in the selection.
selection.max_range[*n*].z	z coordinate of the maximum coordinates of the bounding box around the nth object in the selection.
selection.min_range_exact[*n*]	minimum coordinates of the bounding box around the nth object in the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.min_range_exact[*n*].x	x coordinate of the minimum coordinates of the bounding box around the nth object in the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.min_range_exact[*n*].y	y coordinate of the minimum coordinates of the bounding box around the nth object in the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.min_range_exact[*n*].z	z coordinate of the minimum coordinates of the bounding box around the nth object in the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.max_range_exact[*n*]	maximum coordinates of the bounding box around the nth object in the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.max_range_exact[*n*].x	x coordinate of the maximum coordinates of the bounding box around the nth object in the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.

selection.max_range_exact[*n*].y	y coordinate of the maximum coordinates of the bounding box around the nth object in the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
selection.max_range_exact[*n*].z	z coordinate of the maximum coordinates of the bounding box around the nth object in the selection. The bounding box ignores the centre of arcs and only takes into account the trimmed region of surfaces.
surface.selected.curves	Returns the number of currently selected surface curves (an **INT**).
surface.selected.points	Returns the number of currently selected surface curve points (an **INT**).

Selection positions commands

Currently, the selection position is only calculated when there is only one object in the selection. Therefore, the number in brackets [ ] is always zero.

selection.key_point[0]
the number of the selected keypoint in a surface or curve.

For a curve, if the keypoint is the nth point, **selection.key_point[0]** is *n*.

For a surface, the following surface describes how the numbers are identified.
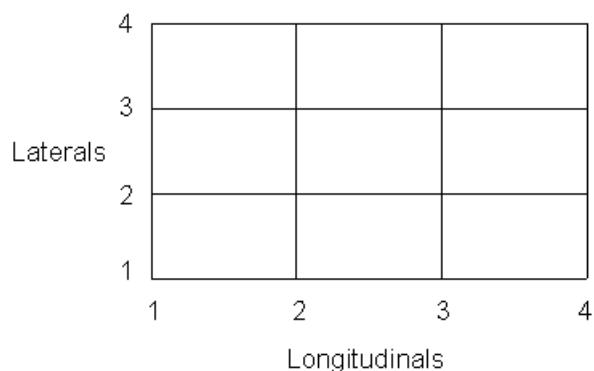
The keypoints are numbered consecutively across the laterals as shown below.
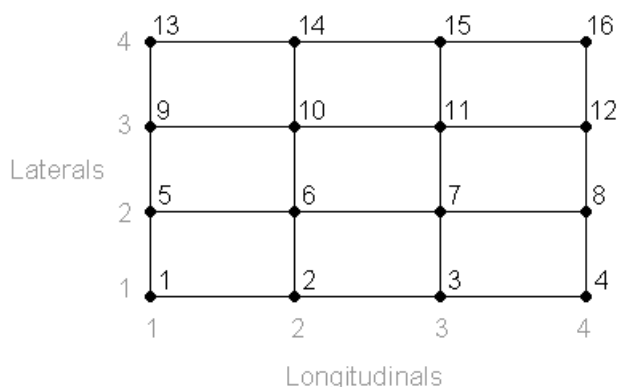
If a spine point is selected, then selection.key_point[0] is the number of points in the surface plus its number in the spine. For example, if a surface has 16 points and the third spine point is selected, then **selection.key_point[0]** is *19*.

selection.nearest_end[0]
the number of the end position nearest the position of selection in a line or arc, where *1* is the start point and *2* is the end point.

selection.composite_item[0]
the number of the object selected in a composite curve. If the third object in the composite curve is selected, then **selection.composite_item[0]** is *3*.

Shareddb command

Command	Description
shareddb.path	pathname of the shared database being used, for example, C:\Users\Public\Documents\shareddb.

Sketcher command

Command	Description
sketch	*1* if Sketcher is on; *0* otherwise.

Solid commands

The following solid commands are available:

Command	Description
solid[*name*].closest_face(*x; y; z*)	returns a string representing the name of the closest face of a solid to a given point. Enter the point in current units and absolute coordinates.
solid[*name*].exists	*1* if the solid exists; *0* otherwise.
XXXX[*entity_name*].owner	returns the Owner string
XXXX[*entity_name*].owner.id	returns the Owner ID
XXXX[*entity_name*].owner.name	returns the Owner Name
XXXX[*entity_name*].owner.type	returns the Owner Type, where **XXXX** is a solid.
solid_active	retrieves the id of the active solid.
solid.active	returns the name of the currently active solid.
solid[*name*].id	unique identity number of the solid in the model.
solid[id *n*].name	name of solid that has the given identity number.

solid[*name*].child[*n*]	returns a string representing the name of the numbered feature below the named solid.
	For example, **print solid[*Fred*].child[*3*]** returns the name of the 3rd feature from the top of the solid named Fred.
feature[*name*].child[*n*]	returns a string representing the name of the feature's numbered sub-branch, such as a Boolean feature.
solid[N].parasolid	returns *1* if the solid is a parasolid; *0* otherwise.
solid[N].v8	returns *1* if the solid is a version 8 solid; *0* otherwise.
solid[*name*].active	*1* if the solid is active; *0* otherwise.
solid[*name*].ghost	*1* for a ghost solid; *0* for a normal solid.
solid[*name*].type	checks the solid and retrieves one of the following strings:
	▪ *Plane*
	▪ *Block*
	▪ *Sphere*
	▪ *Cylinder*
	▪ *Cone*
	▪ *Torus*
	▪ *Extrusion*
	▪ *GeneralSolid*
	▪ *Revolution*
solid[*name*].surfaces	number of surfaces in the solid.
solid[*name*].surface[*number*]	name of the surface in the solid.
solid[N].surface[M].id	returns the id number of the **M**th surface of solid **N**, or the representation number if a parasolid solid.
solid[N].surface[M].name	returns the name of the **M**th surface of solid **N**. This is the same as **solid[N].surface[M]**.
solid[*name*].min_size	minimum coordinates of the bounding box around the solid.
solid[*name*].max_size	maximum coordinates of the bounding box around the solid.
solid[*name*].origin	origin of the solid.

solid[*name*].origin.x	x coordinate of the origin of the solid.
solid[*name*].origin.y	y coordinate of the origin of the solid.
solid[*name*].origin.z	z coordinate of the origin of the solid.
solid[*name*].radius	radius of a cylinder or a sphere.
solid[*name*].length	length of one of the following primitives: block; cylinder; cone; extrusion; plane.
solid[*name*].width	width of a block or a plane.
solid[*name*].diameter	diameter of solid.
solid[*name*].height	height of a block.
solid[*name*].neglength	negative length of an extrusion
solid[*name*].base_radius	radius of a cone on the base of its workplane.
solid[*name*].top_radius	radius of a cone furthest from the base of its workplane.
solid[*name*].major_radius	major radius of a torus.
solid[*name*].minor_radius	minor radius of a torus.
solid[*name*].draft_angle	draft angle of an extrusion.
solid[*name*].angle	angle of primitive revolution
solid[*name*].xaxis **solid[*name*].yaxis** **solid[*name*].zaxis**	The following return the X, Y or Z unit axis vector of the primitive's workplane. The vector is defined in relation to the currently active workplane:
solid[*name*].xaxis.x **solid[*name*].xaxis.y** **solid[*name*].xaxis.z** **solid[*name*].yaxis.x** **solid[*name*].yaxis.y** **solid[*name*].yaxis.z** **solid[*name*].zaxis.x** **solid[*name*].zaxis.y** **solid[*name*].zaxis.z**	The following return the X, Y or Z entity of the unit axis vector of the primitive's workplane. The vector is defined in relation to the currently active workplane:
solid[*name*].area	surface area of the solid.
solid[*name*].volume	volume of the solid.
solid[*name*].watertight	*1* if the solid is watertight within tolerance; *0* otherwise.
solid[*name*].closed	*1* if the solid is closed; *0* otherwise.

solid[*name*].cog	coordinates [x, y, z] of the centre of gravity of the solid.
solid[*name*].cog.x	x coordinate of the centre of gravity of the solid.
solid[*name*].cog.y	y coordinate of the centre of gravity of the solid.
solid[*name*].cog.z	z coordinate of the centre of gravity of the solid.
solid[*name*].moi	coordinates [x, y, z] of the moment of inertia of the solid.
solid[*name*].moi.x	x coordinate of the moment of inertia of the solid.
solid[*name*].moi.y	y coordinate of the moment of inertia of the solid.
solid[*name*].moi.z	z coordinate of the moment of inertia of the solid.
solid[*name*].nlinks	number of linked half edges of a solid, where a half edge is a segment of a boundary of a face.
solid[*name*].tolerance	tolerance to which the half edges are known to link, where a half edge is a segment of a boundary of a face.
solid[*name*].trimming_valid	*1* if boundaries in the solid are valid; *0* otherwise.
solid[*name*].connected	*1* if the surfaces which define the solid connect together within tolerance; *0* otherwise.
solid[*name*].material.polish	polish value of the material used on the solid.
solid[*name*].material.emission	emission value of the material used on the solid.
solid[*name*].material.transparency	transparency value of the material used on the solid.
solid[*name*].material.reflectance	reflectance value of the material used on the solid.
solid[*name*].material.colour	rgb colour values of the material used on the solid.
solid[*name*].material.name	name of the material used for the solid.
solid[*name*].style.colour	colour number of line style used to draw the solid.

solid[*name*]**.style.color**	color (USA) number of line style used to draw the solid.
solid[*name*]**.style.gap**	gap of line style used to draw the solid.
solid[*name*]**.style.weight**	weight of line style used to draw the solid.
solid[*name*]**.style.width**	width of line style used to draw the solid.
solid[*name*]**.level**	level on which the solid exists.
solid.constraint.exists	*1* if scaling constraint exists; *0* otherwise.
solid.constraint.type	*Fixed Size* or *Fixed Distance* to indicate the type of scaling constraint.
solid.constraint.origin	the coordinates of the scaling constraint plane origin.
solid.constraint.xaxis	vector representing the X axis of the scaling constraint plane.
solid.constraint.yaxis	vector representing the Y axis of the scaling constraint plane.
solid.constraint.zaxis	vector representing the Z axis of the scaling constraint plane.

There are also commands to retrieve information about solid features (see page 201) and for picking faces of a solid (see page 206).

Features commands

Command	Description
solid[*name*]**.feature**[*fname*]**.exists** **feature**[*fname*]**.exists**	*1* if the feature exists; *0* otherwise.
solid[*name*]**.feature**[*fname*]**.id** **feature**[*fname*]**.id**	the integer id of the feature.
solid[*name*]**.feature**[*fname*]**.exists** **feature**[*fname*]**.exists**	*1* if the feature exists; *0* otherwise.
solid[*name*]**.feature**[*fname*]**.name** **feature**[*fname*]**.name**	name of the feature.
solid[*name*]**.feature**[*fname*]**.type** **feature**[*fname*]**.type**	type of feature (for example, " fillet", "boss").
solid[*name*]**.feature**[*fname*]**.suppress ed** **feature**[*fname*]**.suppressed**	*1* if the feature currently suppressed; *0* otherwise.

solid[*name*]**.feature**[*fname*]**.error** **feature**[*fname*]**.error**	*1* if the feature error suppressed; *0* otherwise.
solid[*name*]**.feature**[*fname*]**.surfaces** **feature**[*fname*]**.surfaces**	number of visible surfaces in the feature.
solid[*name*]**.feature**[*fname*]**.length** **feature**[*fname*]**.length**	the length/depth/height of the cut/boss feature.
solid[*name*]**.feature**[*fname*]**.angle** **feature**[*fname*]**.angle**	angle of the cut/boss/bulge feature.
solid[*name*]**.feature**[*fname*]**.radius** **feature**[*fname*]**.radius**	radius of the fillet feature.

In addition, the following groups of feature commands are available:

- Holes (see page 202)
- Pockets and protrusions (see page 203)
- Number of features (see page 204)
- Workplane of feature (see page 204)
- Other feature commands (see page 205)

Holes commands

Command	Description
solid[*name*]**.feature**[*fname*]**.origin**	origin of the hole
solid[*name*]**.feature**[*fname*]**.main_depth**	depth of the hole's main section
solid[*name*]**.feature**[*fname*]**.main_diameter**	diameter of the hole's main section
solid[*name*]**.feature**[*fname*]**.bore_depth**	depth of the hole's bore section (if any)
solid[*name*]**.feature**[*fname*]**.bore_diameter**	diameter of the hole's bore section (if any)
solid[*name*]**.feature**[*fname*]**.sink_diameter**	diameter of the hole's sink section (if any)
solid[*name*]**.feature**[*fname*]**.tap_depth**	depth of the hole's tap section (if any)
solid[*name*]**.feature**[*fname*]**.tap_diameter**	diameter of the hole's tap section (if any)
solid[*name*]**.feature**[*fname*]**.tap_pitch**	pitch of the hole's tap section (if any)

Pockets and protrusions commands

You can use the following commands to determine the dimensions of pockets and protrusions. The commands return the required dimension and take the form:

feature[name].length

The following commands are available for pockets and protrusions:

Command	Description
length	Length of the pocket
width	Width of the pocket
height	Height of the protrusion. This returns the same value as **depth**
depth	Depth of the pocket. This returns the same value as **height**
angle1	Draft angle of top wall
angle2	Draft angle of right wall
angle3	Draft angle of bottom wall
angle4	Draft angle of left wall
radius	Radius of joining fillet
radius1	Radius of top left corner fillet
radius2	Radius of top right corner fillet
radius3	Radius of bottom right corner fillet
radius4	Radius of bottom left corner fillet
radius5	Radius of base fillet of a pocket, or top fillet of a protrusion

You can use the existing hole commands to determine the dimensions of the hole in the corner(s) of the pocket. For example, the following command will return the main diameter of the hole in the corner of the pocket:

print feature[name].main_diameter

Number of features commands

Command	Description
solid[*name***].children** **solid[***name***].features**	number of features on the solid.
solid[*name***].children.all** **solid[***name***].features.all**	number of features on the solid, including sub-branches on the feature tree.
solid[*name***].children.selected** **solid[***name***].features.selected**	number of selected features on the solid.
feature[*name***].children** **feature[***name***].features**	number of features in the sub-branch. It can be used with Boolean and Group features.
feature[*name***].features.all**	number of features, including all sub-branches.

Workplane of feature commands

Command	Description
feature[*name***].xaxis** **feature[***name***].yaxis** **feature[***name***].zaxis**	return the X, Y or Z unit axis vector of the feature's workplane. The vector is relative to the currently active workplane.
feature[*name***].xaxis.x** **feature[***name***].xaxis.y** **feature[***name***].xaxis.z** **feature[***name***].yaxis.x** **feature[***name***].yaxis.y** **feature[***name***].yaxis.z** **feature[***name***].zaxis.x** **feature[***name***].zaxis.y** **feature[***name***].zaxis.z**	return the X, Y or Z entity of the unit axis vector of the feature's workplane. The vector is relative to the currently active workplane.

For example:

print feature[*1***].xaxis**

print feature[*1***].xaxis.y**

Other feature commands

Command	Description
feature[*name*].selected	*1* for a selected feature; *0* otherwise.
feature[*name*].suppressed	*1* for suppressed feature; *0* otherwise.
feature[*name*].error	*1* for an error state for a feature; *0* otherwise.
feature[*name*].exists	*1* if the solid feature exists; *0* otherwise.
feature[*name*].id	unique identity number of the solid feature in the model.
feature[id *n*].name	name of solid feature that has the given identity number.
feature[*name*].type	checks the solid feature and retrieves a string indicating the type of feature.
feature[*name*].surfaces	number of visible surfaces that make up the feature.
feature[*name of feature*].surface[*n*]	the name of the *n*th surface of a solid feature, where *n* is the number of the surface of the solid feature.
feature[*name*].length	length of the feature - applies to cut and boss features only.
feature[*name*].angle	angle of the feature. Applies to cut, boss and bulge features only.
feature[*name*].radius	radius of feature. Applies to fillet feature only.
feature[*feature name*].machine	*1* if feature is to be machined; *0* otherwise.
feature[*feature name*].pre_machined	*1* if feature is pre-machined; 0 otherwise.
feature[*feature name*].existed_at_birth	*1* if feature was present in the original solid (for example, a feature existing in a manufacturer standard moldbase component); *0* if the feature was added later.
feature.constraint.exists	*1* if scaling constraint exists; *0* otherwise.
feature.constraint.type	returns *Fixed Size* or *Fixed Distance* to indicate the type of scaling constraint.

feature.constraint.origin	returns the coordinates of the scaling constraint plane origin.
feature.constraint.xaxis	returns a vector representing the X axis of the scaling constraint plane.
feature.constraint.yaxis	returns a vector representing the Y axis of the scaling constraint plane.
feature.constraint.zaxis	returns a vector representing the Z axis of the scaling constraint plane.

Picking faces of a solid commands

When in face selection mode, you can use commands to pick the faces of a selected solid.

Command	Description
pick face name *<face_name>* **pick face** *<face_name>* **pick face replace name** *<face_name>* **pick face name** *<face_name>*	Use these commands to replace the currently selected faces with named faces. This is the same as using the mouse to select the faces.
pick face add name *<face_name>* **pick face add** *<face_name>*	Use these commands to add the named face to the current selection. This is the same as holding down the SHIFT key and left-clicking.
pick face toggle name *<face_name>* **pick face toggle** *<face_name>*	Use these commands to toggle the named face into/out of the current selection. This is the same as holding down the CTRL key and left-clicking.

<face_name> can be a word, string, integer or variable. The following are valid:

- **pick face** *fred*
- **pick face** *'fred'*
- **pick face** *23*
- **string face_name =** *'fred'*
- **pick face $***face_name*

The commands are also available during the following operations:

- Multiple-face selection modes; if you are in convex face selection mode, several faces are selected, spreading out from the named face.
- **Solid Draft Face**
- **Solid Replace Face**
- **Solid Divide Face**

Surface commands

The following surface commands are available:

Command	Description
surface[*name*].exists	*1* if the surface exists; *0* otherwise.
surface[*name*].id	unique identity number of the surface in the model.
surface[id *n*].name	name of surface that has the given identity number.
surface[*name*].description	description of the surface as stored in the database
surface[1].tangentpoint(1;2;3;4;5;6)	A point on a surface such that if viewed from an outside point, the line joining the two points is tangent to the surface. The first three coordinates are a point outside the surface and the last three coordinates are the initial guess point on the surface.
surface[*name*].direction	unit vector of the reference direction of the surface.
surface[*name*].direction.x	x value of the unit vector of the reference direction of the surface.
surface[*name*].direction.y	y value of the unit vector of the reference direction of the surface.
surface[*name*].direction.z	z value of the unit vector of the reference direction of the surface.
surface[*name*].trimmed	*1* if the surface's local trim flag is set; *0* otherwise.
surface[*name*].min_size	coordinates [x, y, z] of the minimum point of the smallest box that fully encloses the surface.

surface[*name*].min_size.x	x coordinate of the minimum point of the smallest box that fully encloses the surface.
surface[*name*].min_size.y	y coordinate of the minimum point of the smallest box that fully encloses the surface.
surface[*name*].min_size.z	z coordinate of the minimum point of the smallest box that fully encloses the surface.
surface[*name*].max_size	coordinates [x, y, z] of the maximum point of the smallest box that fully encloses the surface.
surface[*name*].max_size.x	x coordinate of the maximum point of the smallest box that fully encloses the surface.
surface[*name*].max_size.y	y coordinate of the maximum point of the smallest box that fully encloses the surface.
surface[*name*].max_size.z	z coordinate of the maximum point of the smallest box that fully encloses the surface.
surface[*name*].type	checks the surface and retrieves one of the following strings:

- *Plane*
- *Block*
- *Sphere*
- *Cylinder*
- *Cone*
- *Torus*
- *Extrusion*
- *Revolution*
- *Powersurface*
- *BCP*
- *NURB*
- *PDGS*

surface[*name*].area	area of the surface.
surface[*name*].diameter	diameter of surface.
surface[*name*].volume	volume of the surface.
surface[*name*].cog	coordinates [x, y, z] of the centre of gravity of the surface.

surface[*name*].cog.x	x coordinate of the centre of gravity of the surface.
surface[*name*].cog.y	y coordinate of the centre of gravity of the surface.
surface[*name*].cog.z	z coordinate of the centre of gravity of the surface.
surface[*name*].evaluate(*t*; *u*).position	coordinates [x, y, z] of the position on the surface defined by the t and u parameters.
surface[*name*].evaluate(*t*; *u*).position.x	x coordinate of the position defined on the surface by the t and u parameters.
surface[*name*].evaluate(*t*; *u*).position.y	y coordinate of the position defined on the surface by the t and u parameters.
surface[*name*].evaluate(*t*; *u*).position.z	z coordinate of the position defined on the surface by the t and u parameters.
surface[name].evaluate(t; u).normal	unit vector of the normal to the surface at the position defined by the t and u parameters.
surface[name].evaluate(t; u).normal.x	x value of the unit vector of the normal to the surface at the position defined by the t and u parameters.
surface[name].evaluate(t; u).normal.y	y value of the unit vector of the normal to the surface at the position defined by the t and u parameters.
surface[name].evaluate(t; u).normal.z	z value of the unit vector of the normal to the surface at the position defined by the t and u parameters.
surface[name].evaluate(t; u).curvature.min	minimum curvature at the position on the surface defined by the t and u parameters.
surface[name].evaluate(t; u).curvature.max	maximum curvature at the position on the surface defined by the t and u parameters.

surface[*name***].near(***x***; ***y***; ***z***)**	t and u parameters on the surface nearest to the coordinates [x, y, z].
	For complex surfaces, you can speed up calculations by supplying approximate t and u values that are close to the coordinates. The approximate values are added in brackets, as shown below:
	surface[*name***].near(***x***; ***y***; *z; guess_t*; *guess_u***)**
	surface[*name***].near(***x***; ***y***; ***z***).t** t parameter on the surface nearest to the coordinates [x, y, z].
	surface[*name***].near(***x***; ***y***; ***z***).u** u parameter on the surface nearest to the coordinates [x, y, z].
XXXX[*entity_name***].owner**	returns the Owner string
XXXX[*entity_name***].owner.id**	returns the Owner ID
XXXX[*entity_name***].owner.name**	returns the Owner Name
XXXX[*entity_name***].owner.type**	returns the Owner Type, where **XXXX** is a surface.
surface[*name***].material.polish**	polish value of the material used on the surface.
surface[*name***].material.emission**	emission value of the material used on the surface.
surface[*name***].material.transparency**	transparency value of the material used on the surface.
surface[*name***].material.reflectance**	reflectance value of the material used on the surface.
surface[*name***].material.colour**	RGB colour values of the material used on the surface.
surface[*name***].material.name**	name of the material used for the surface.
surface[*name***].trimming_valid**	*1* if the trim boundaries on the surface form a valid trim region; *0* otherwise.
surface[*name***].boundaries**	number of boundaries on the surface.
surface[*name***].pcurves**	number of pcurves on the surface.

surface[*name***].pcurve[***number***]**	name of the pcurve on the surface. Each pcurve on the surface has a unique *number*, where *number* ranges from *1* to the value of surface[*name*].pcurves.
surface[*name***].style.colour**	colour number of line style used to draw the surface if it is one of the basic 16 colours or *-1* if it is an RGB colour.
	The following variables exist to check the RGB colour of items
	surface[*name***].style.colour.red**
	surface[*name***].style.colour.green**
	surface[*name***].style.colour.blue**
	surface[*name***].style.colour.rgb**
	surface[*name***].style.colour.r**
	surface[*name***].style.colour.g**
	surface[*name***].style.colour.b**
surface[*name***].style.color**	color (USA) number of line style used to draw the surface.
surface[*name***].style.gap**	gap of line style used to draw the surface.
surface[*name***].style.weight**	weight of line style used to draw the surface.
surface[*name***].style.width**	width of line style used to draw the surface.
surface[*name***].level**	level on which the surface exists.

You can also use the following groups of commands:

- Primitives (see page 212)
- Laterals (see page 214) and longitudinals (see page 216)
- Spines (see page 220)
- Number of selected surface curves/surface curve points (see page 190)

Primitive commands

Surface syntax in this section applies to primitive surfaces (including extrusions). It outputs data about the surface's dimensions and workplane instrumentation.

Command	Description
surface[name].**radius**	radius of a cylinder or a sphere.
surface[*name*].**length**	length of one of the following primitives: block; cylinder; cone; extrusion; plane.
surface[*name*].**width**	width of a block or a plane.
surface[*name*].**height**	height of a block.
surface[*name*].**base_radius**	radius of a cone on the base of its workplane.
surface[*name*].**top_radius**	radius of a cone furthest from the base of its workplane.
surface[*name*].**major_radius**	major radius of a torus.
surface[*name*].**minor_radius**	minor radius of a torus.
surface[*name*].**neglength**	negative length of an extrusion.
surface[*name*].**draft_angle**	draft angle of an extrusion.
surface[*name*].**origin**	coordinates [x, y, z] of the origin of the primitive's workplane instrumentation.
surface[*name*].**origin.x**	x coordinate of the origin of the primitive's workplane instrumentation.
surface[*name*].**origin.y**	y coordinate of the origin of the primitive's workplane instrumentation.
surface[*name*].**origin.z**	z coordinate of the origin of the primitive's workplane instrumentation.
surface[*name*].**xaxis**	unit vector which defines the orientation of the X-axis of the primitive's workplane instrumentation.
surface[*name*].**xaxis.x**	x value of the unit vector which defines the orientation of the X-axis of the primitive's workplane instrumentation.
surface[*name*].**xaxis.y**	y value of the unit vector which defines the orientation of the X-axis of the primitive's workplane instrumentation.
surface[*name*].**xaxis.z**	z value of the unit vector which defines the orientation of the X-axis of the primitive's workplane instrumentation.

surface[*name*].yaxis	unit vector which defines the orientation of the Y-axis of the primitive's workplane instrumentation.
surface[*name*].yaxis.x	x value of the unit vector which defines the orientation of the Y-axis of the primitive's workplane instrumentation.
surface[*name*].yaxis.y	y value of the unit vector which defines the orientation of the Y-axis of the primitive's workplane instrumentation.
surface[*name*].yaxis.z	z value of the unit vector which defines the orientation of the Y-axis of the primitive's workplane instrumentation.
surface[*name*].zaxis	unit vector which defines the orientation of the Z-axis of the primitive's workplane instrumentation.
surface[*name*].zaxis.x	x value of the unit vector which defines the orientation of the Z-axis of the primitive's workplane instrumentation.
surface[*name*].zaxis.y	y value of the unit vector which defines the orientation of the Z-axis of the primitive's workplane instrumentation.
surface[*name*].zaxis.z	z value of the unit vector which defines the orientation of the Z-axis of the primitive's workplane instrumentation.
surface[*name*].xaxis **surface[*name*].yaxis** **surface[*name*].zaxis**	return the X, Y or Z unit axis vector of the primitive's workplane. The vector is relative to the currently active workplane.
surface[*name*].xaxis.x **surface[*name*].xaxis.y** **surface[*name*].xaxis.z** **surface[*name*].yaxis.x** **surface[*name*].yaxis.y** **surface[*name*].yaxis.z** **surface[*name*].zaxis.x** **surface[*name*].zaxis.y** **surface[*name*].zaxis.z**	return the X, Y or Z entity of the unit axis vector of the primitive's workplane. The vector is relative to the currently active workplane.

Surface lateral commands

Command	Description
surface[*name*].lateral[*number*].start	coordinates [x, y, z] of the start position of the lateral.
surface[*name*].lateral[*number*].start.x	x coordinate of start position of the lateral.
surface[*name*].lateral[*number*].start.y	y coordinate of start position of the lateral.
surface[*name*].lateral[*number*].start.z	z coordinate of start position of the lateral.
surface[*name*].lateral[*number*].end	coordinates [x, y, z] of the end position of the lateral.
surface[*name*].lateral[*number*].end.x	x coordinate of end position of the lateral.
surface[*name*].lateral[*number*].end.y	y coordinate of end position of the lateral.
surface[*name*].lateral[*number*].end.z	z coordinate of end position of the lateral.
surface[*name*].lateral[*number*].number	number of points in the lateral.
surface[*name*].lateral[*number*].length	length of the lateral.
surface[*name*].lateral[*number*].length_between(*a*; *b*)	length along the lateral between lateral points a and b.
surface[*name*].lateral[*number*].exists	1 if lateral exists; 0 otherwise.
surface[*name*].lateral[*number*].id	unique identity number of the lateral.
surface[*name*].lateral[*number*].name	name of the lateral.
surface[*name*].lateral[*number*].point[*number*]	coordinates [x, y, z] of the position of the lateral's point.
surface[*name*].lateral[*number*].point[*number*].x	x coordinate of the position of the lateral's point.
surface[*name*].lateral[*number*].point[*number*].y	y coordinate of the position of the lateral's point.
surface[*name*].lateral[*number*].point[*number*].z	z coordinate of the position of the lateral's point.
surface[*name*].lateral[*number*].point[*number*].entry_magnitude	magnitude entering the lateral's point.
surface[*name*].lateral[*number*].point[*number*].exit_magnitude	magnitude leaving the lateral's point.

surface[name**].lateral[**number**].point[**number**].entry_tangent**	unit vector of the tangent direction entering the lateral's point.
surface[name**].lateral[**number**].point[**number**].entry_tangent.x**	x value of the unit vector which defines the tangent direction entering the lateral's point.
surface[name**].lateral[**number**].point[**number**].entry_tangent.y**	y value of the unit vector which defines the tangent direction entering the lateral's point.
surface[name**].lateral[**number**].point[**number**].entry_tangent.z**	z value of the unit vector which defines the tangent direction entering the lateral's point.
surface[name**].lateral[**number**].point[**number**].exit_tangent**	unit vector of the tangent direction leaving the lateral's point.
surface[name**].lateral[**number**].point[**number**].exit_tangent.x**	x value of the unit vector which defines the tangent direction leaving the lateral's point.
surface[name**].lateral[**number**].point[**number**].exit_tangent.y**	y value of the unit vector which defines the tangent direction leaving the lateral's point.
surface[name**].lateral[**number**].point[**number**].exit_tangent.z**	z value of the unit vector which defines the tangent direction leaving the lateral's point.
surface[name**].lateral[**number**].point[**number**].entry_tangent.azimuth**	azimuth angle of the tangent entering the point.
surface[name**].lateral[**number**].point[**number**].entry_tangent.elevation**	elevation angle of the tangent entering the point.
surface[name**].lateral[**number**].point[**number**].exit_tangent.azimuth**	azimuth angle of the tangent leaving the point.
surface[name**].lateral[**number**].point[**number**].exit_tangent.elevation**	elevation angle of the tangent leaving the point.
surface[name**].lateral[**number**].point[**number**].entry_normal**	unit vector of the normal entering the lateral's point.
surface[name**].lateral[**number**].point[**number**].entry_normal.x**	x value of the unit vector of the normal entering the lateral's point.
surface[name**].lateral[**number**].point[**number**].entry_normal.y**	y value of the unit vector of the normal entering the lateral's point.
surface[name**].lateral[**number**].point[**number**].entry_normal.z**	z value of the unit vector of the normal entering the lateral's point.
surface[name**].lateral[**number**].point[**number**].exit_normal**	unit vector of the normal leaving the lateral's point.

surface[*name***].lateral[***number***].point[***number***].exit_normal.x**	x value of the unit vector of the normal leaving the lateral's point.
surface[*name***].lateral[***number***].point[***number***].exit_normal.y**	y value of the unit vector of the normal leaving the lateral's point.
surface[*name***].lateral[***number***].point[***number***].exit_normal.z**	z value of the unit vector of the normal leaving the lateral's point.
surface[*name***].lateral[***number***].cog**	coordinates [x, y, z] of the centre of gravity of the lateral.
surface[*name***].lateral[***number***].cog.x**	x coordinate of the centre of gravity of the lateral.
surface[*name***].lateral[***number***].cog.y**	y coordinate of the centre of gravity of the lateral.
surface[*name***].lateral[***number***].cog.z**	z coordinate of the centre of gravity of the lateral.
surface[*name***].lat_closed**	*1* if the surface's laterals are closed; *0* if open.
surface[*name***].nlats**	number of laterals in the surface.
surface[*name***].lateral.selected**	list of names of the currently selected laterals of the surface.
surface[*name***].lateral[***number***].selected**	1 if lateral is selected; 0 if unselected.

Surface longitudinal commands

Command	**Description**
surface[*name***].longitudinal[***number***].start**	coordinates [x, y, z] of the start position of the longitudinal.
surface[*name***].longitudinal[***number***].start.x**	x coordinate of start position of the longitudinal.
surface[*name***].longitudinal[***number***].start.y**	y coordinate of start position of the longitudinal.
surface[*name***].longitudinal[***number***].start.z**	z coordinate of start position of the longitudinal.
surface[*name***].longitudinal[***number***].end**	coordinates [x, y, z] of the end position of the longitudinal.
surface[*name***].longitudinal[***number***].end.x**	x coordinate of end position of the longitudinal.

surface[*name*].**longitudinal**[*number*].**end.y**	y coordinate of end position of the longitudinal.
surface[*name*].**longitudinal**[*number*].**end.z**	z coordinate of end position of the longitudinal.
surface[*name*].**longitudinal**[*number*].**number**	number of points in the longitudinal.
surface[*name*].**longitudinal**[*number*].**length**	length of the longitudinal.
surface[*name*].**longitudinal**[*number*].**length_between(***a*; *b***)**	length along the longitudinal between longitudinal points a and b.
surface[*name*].**longitudinal**[*number*].**exists**	*1* if longitudinal exists; *0* otherwise.
surface[*name*].**longitudinal**[*number*].**id**	unique identity number of the longitudinal.
surface[*name*].**longitudinal**[*number*].**name**	name of the longitudinal.
surface[*name*].**longitudinal**[*number*].**point**[*number*]	coordinates [x, y, z] of the position of the longitudinal's point.
surface[*name*].**longitudinal**[*number*].**point**[*number*]**.x**	x coordinate of the position of the longitudinal's point.
surface[*name*].**longitudinal**[*number*].**point**[*number*]**.y**	y coordinate of the position of the longitudinal's point.
surface[*name*].**longitudinal**[*number*].**point**[*number*]**.z**	z coordinate of the position of the longitudinal's point.
surface[*entity_name*].**evaluate(***T; U***).udirb** **surface**[*entity_name*].**evaluate(***T; U***).udira**	tangent vector U, direction before/after (around lateral) of the specified (T,U) point on the surface.
surface[*entity_name*].**evaluate(***T; U***).tdirb** **surface**[*entity_name*].**evaluate(***T; U***).tdira**	tangent vector T, direction before/after (along longitudinal) of the specified (T,U) point on the surface .
surface[*entity_name*].**evaluate(***T; U***)**	coordinates of the specified (T,U) point on the surface .
surface[*entity_name*].**evaluate(***T; U***).position**	coordinates of the specified (T,U) point on the surface.
surface[*entity_name*].**evaluate(***T; U***).normal**	normal direction of the specified (T,U) point on the surface.
surface[*entity_name*].**evaluate(***T; U***).draft_angle**	draft angle of the surface at specified (T,U) point.

surface[*entity_name***].evaluate(***T; U***).curvature.min**	minimum curvature of the surface at specified (T,U) point.
surface[*entity_name***].evaluate(***T; U***).cuvature.max**	maximum curvature of the surface at specified (T,U) point.
surface[*name***].longitudinal[***number***]. point[***number***].entry_magnitude**	magnitude entering the longitudinal's point.
surface[*name***].longitudinal[***number***]. point[***number***].exit_magnitude**	magnitude leaving the longitudinal's point.
surface[*name***].longitudinal[***number***]. point[***number***].entry_tangent**	unit vector of the tangent direction entering the longitudinal's point.
surface[*name***].longitudinal[***number***]. point[***number***].entry_tangent.x**	x value of the unit vector which defines the tangent direction entering the longitudinal's point.
surface[*name***].longitudinal[***number***]. point[***number***].entry_tangent.y**	y value of the unit vector which defines the tangent direction entering the longitudinal's point.
surface[*name***].longitudinal[***number***]. point[***number***].entry_tangent.z**	z value of the unit vector which defines the tangent direction entering the longitudinal's point.
surface[*name***].longitudinal[***number***]. point[***number***].exit_tangent**	unit vector of the tangent direction leaving the longitudinal's point.
surface[*name***].longitudinal[***number***]. point[***number***].exit_tangent.x**	x value of the unit vector which defines the tangent direction leaving the longitudinal's point.
surface[*name***].longitudinal[***number***]. point[***number***].exit_tangent.y**	y value of the unit vector which defines the tangent direction leaving the longitudinal's point.
surface[*name***].longitudinal[***number***]. point[***number***].exit_tangent.z**	z value of the unit vector which defines the tangent direction leaving the longitudinal's point.
surface[*name***].longitudinal[***number***]. point[***number***].entry_tangent.azimuth**	azimuth angle of the tangent entering the point.
surface[*name***].longitudinal[***number***]. point[***number***].entry_tangent.elevatio n**	elevation angle of the tangent entering the point.
surface[*name***].longitudinal[***number***]. point[***number***].exit_tangent.azimuth**	azimuth angle of the tangent leaving the point.
surface[*name***].longitudinal[***number***]. point[***number***].exit_tangent.elevation**	elevation angle of the tangent leaving the point.

surface[*name*].longitudinal[*number*].point[*number*].entry_normal	unit vector of the normal entering the longitudinal's point.
surface[*name*].longitudinal[*number*].point[*number*].entry_normal.x	x value of the unit vector of the normal entering the longitudinal's point.
surface[*name*].longitudinal[*number*].point[*number*].entry_normal.y	y value of the unit vector of the normal entering the longitudinal's point.
surface[*name*].longitudinal[*number*].point[*number*].entry_normal.z	z value of the unit vector of the normal entering the longitudinal's point.
surface[*name*].longitudinal[*number*].point[*number*].exit_normal	unit vector of the normal leaving the longitudinal's point.
surface[*name*].longitudinal[*number*].point[*number*].exit_normal.x	x value of the unit vector of the normal leaving the longitudinal's point.
surface[*name*].longitudinal[*number*].point[*number*].exit_normal.y	y value of the unit vector of the normal leaving the longitudinal's point.
surface[*name*].longitudinal[*number*].point[*number*].exit_normal.z	z value of the unit vector of the normal leaving the longitudinal's point.
surface[*name*].longitudinal[*number*].cog	coordinates [x, y, z] of the centre of gravity of the longitudinal.
surface[*name*].longitudinal[*number*].cog.x	x coordinate of the centre of gravity of the longitudinal.
surface[*name*].longitudinal[*number*].cog.y	y coordinate of the centre of gravity of the longitudinal.
surface[*name*].longitudinal[*number*].cog.z	z coordinate of the centre of gravity of the longitudinal.
surface[*name*].lateral[*number*].point[*number*].entry_tangent.flare	flare angle of the longitudinal entering the point.
surface[*name*].lateral[*number*].point[*number*].entry_tangent.twist	twist angle of the longitudinal entering the point.
surface[*name*].lateral[*number*].point[*number*].exit_tangent.flare	flare angle of the longitudinal leaving the point.
surface[*name*].lateral[*number*].point[*number*].exit_tangent.twist	twist angle of the longitudinal leaving the point.
surface[*name*].longitudinal[*number*].point[*number*].entry_tangent.flare	flare angle entering the point.
surface[*name*].longitudinal[*number*].point[*number*].entry_tangent.twist	twist angle entering the point.
surface[*name*].longitudinal[*number*].point[*number*].exit_tangent.flare	flare angle leaving the point.

surface[*name*].longitudinal[*number*].point[*number*].exit_tangent.twist	twist angle leaving the point.
surface[*name*].lon_closed	*1* if the surface's longitudinals are closed; *0* if open.
surface[*name*].nlons	number of longitudinals in the surface.
surface[*name*].longitudinal.selected	list of names of the currently selected laterals or longitudinals of the surface.
surface[*name*].longitudinal[*number*].selected	1 if lateral or longitudinal is selected; 0 if unselected.

Spine commands

Command	Description
surface[*name*].spine.exists	*1* if the spine exists; *0* otherwise.
surface[*name*].spine.id	unique identity number of the spine.
surface[id *n*].spine.name	name of the spine that has the given identity number.
surface[*name*].spine.number	number of spine points.
surface[*name*].spine.length	length of the spine.
surface[*name*].spine.length_between(*a*; *b*)	length along the spine between spine points a and b.
surface[*name*].spine.start	start coordinates [x, y, z] of the spine.
surface[*name*].spine.start.x	x coordinate of the start of the spine.
surface[*name*].spine.start.y	y coordinate of the start of the spine.
surface[*name*].spine.start.z	z coordinate of the start of the spine.
surface[*name*].spine.end	end coordinates [x, y, z] of the spine.
surface[*name*].spine.end.x	x coordinate of the end of the spine.
surface[*name*].spine.end.y	y coordinate of the end of the spine.
surface[*name*].spine.end.z	z coordinate of the end of the spine.
surface[*name*].spine.point[*number*]	coordinates [x, y, z] of the spine point.
surface[*name*].spine.point[*number*].x	x coordinate of the spine point.
surface[*name*].spine.point[*number*].y	y coordinate of the spine point.
surface[*name*].spine.point[*number*].z	z coordinate of the spine point.
surface[*name*].spine.point[*number*].tangent	unit vector of the tangent direction through the spine point.

surface[*name***].spine.point[***number***].tangent.x**	x value of the unit vector of the tangent direction through the spine point.
surface[*name***].spine.point[***number***].tangent.y**	y value of the unit vector of the tangent direction through the spine point.
surface[*name***].spine.point[***number***].tangent.z**	z value of the unit vector of the tangent direction through the spine point.
surface[*name***].spine.point[***number***].entry_tangent.azimuth**	azimuth angle of the tangent entering the spine point.
surface[*name***].spine.point[***number***].entry_tangent.elevation**	elevation angle of the tangent entering the spine point.
surface[*name***].spine.point[***number***].exit_tangent.azimuth**	azimuth angle of the tangent leaving the spine point.
surface[*name***].spine.point[***number***].exit_tangent.elevation**	elevation angle of the tangent leaving the spine point.

Symbol commands

The following symbol commands are available:

Command	Description
symbol[*name***].exists**	*1* if the symbol exists; *0* otherwise.
symbol[*name***].id**	unique identity number of the symbol in the model.
symbol[id *n*].name	name of the symbol that has the given identity number.
symbol[*name***].position[***pin number***]**	coordinates [x, y, z] of the named pin.
symbol[*name***].position[***pin number***].x**	x coordinate of the named pin.
symbol[*name***].position[***pin number***].y**	y coordinate of the named pin.
symbol[*name***].position[***pin number***].z**	z coordinate of the named pin.
symbol[*name***].number**	number of pins in the symbol.
symbol[*name***].style.colour**	colour number of line style used to draw the symbol.
symbol[*name***].style.color**	color (USA) number of line style used to draw the symbol.
symbol[*name***].style.gap**	gap of line style used to draw the symbol.

symbol[*name*].style.weight	weight of line style used to draw the symbol.
symbol[*name*].style.width	width of line style used to draw the symbol.
symbol[*name*].level	level on which the symbol exists.
symbol[*name*].area	area of triangulated symbols.
symbol[*name*].volume	volume of triangulated symbols.
symbol.constraint.exists	*1* if scaling constraint exists; *0* otherwise.
symbol.constraint.type	*Fixed Size* or *Fixed Distance* to indicate the type of scaling constraint.
symbol.constraint.origin	the coordinates of the scaling constraint plane origin.
symbol.constraint.xaxis	a vector representing the X axis of the scaling constraint plane.
symbol.constraint.yaxis	a vector representing the Y axis of the scaling constraint plane.
symbol.constraint.zaxis	a vector representing the Z axis of the scaling constraint plane.
symbol_def[*name*].exists	*1* if the symbol definition exists; *0* otherwise.
symbol_def[*name*].id	unique identity number for the symbol definition.
symbol_def[id *n*].name	name of the symbol definition that has the given identity number.

Text commands

Command	Description
text[*name*].exists	*1* if the text exists; *0* otherwise.
text[*name*].id	unique identity number of the text in the model.
text[id *n*].name	name of the text that has the given identity number.
text[*name*].string	text string.
text[*name*].livetext	*1* if text created using PowerShape standard text editor; *0* for DUCT editor.
text[*name*].colour	number of the colour used by the text.
text[*name*].level	level on which the text exists.
text[*name*].string.unstripped	text string with format characters.
text[text_name].string.unstripped.length	returns length of unstripped text.
text[*text_name*].string.unstripped.char[*ipos*]	returns the character at the specified position in unstripped text string, where *ipos* is greater or equal to *0* and less than the string length
text[*name*].string.stripped	text string without format characters.
text[*text_name*].string.stripped.length	returns length of stripped text.
text[*text_name*].string.stripped.char[*ipos*]	returns the character at the specified position in stripped text string, where *ipos* is greater or equal to *0* and less than the string length.
text[*text_name*].string.stripped.locate[*string*]	returns the location of *string* in stripped text string. If *string* isn't found, *-1* is returned.
text[*name*].font	name of the font used by the text.

text[*name*].origin	the origin of the text is output as one of the following strings: ■ *Bottom Left* ■ *Bottom Centre* ■ *Bottom Right* ■ *Centre Left* ■ *Centre* ■ *Centre Right* ■ *Top Left* ■ *Top Centre* ■ *Top Righ*t
text[*name*].position	coordinates [x, y, z] of the position at which the text was placed.
text[*name*].position.x	x coordinate of the position at which the text was placed.
text[*name*].position.y	y coordinate of the position at which the text was placed.
text[*name*].position.z	z coordinate of the position at which the text was placed.
text[*name*].char_height	height of the characters.
text[*name*].char_spacing	spacing between individual characters (pitch).
text[*name*].angle	angle of the text.
text[*name*].line_spacing	spacing between lines of text.
text[*name*].justification	justification of the text is output as one of the following strings: ■ *Left* ■ *Centre* ■ *Right*
text[*name*].horizontal	*1* if text characters are horizontal; *0* otherwise.
text[*name*].italic	*1* if text is italic; *0* otherwise.

Tolerance commands

Command	Description
tolerance.general	value of general tolerance.
tolerance.drawing	value of drawing tolerance.

Units commands

Command	Description
unit[type].name	name of the units for type. For example, type length's output can be mm.
unit[type].factor	number by which the default unit is multiplied by to give the units in unit[type].name.

For example, *type* **length** has default units *mm*. If **unit[length].name** is *inches*, then the **unit[length].factor** is *0.039370*.

Updated object commands

You can use these commands to query which objects were updated as a result of the last operation. These objects are accessed from the updated list.

Command	Description
updated.exists	*1* if at least one item is in the updated list; *0* otherwise.
Updated.clearlist	objects are removed from the updated list.
updated.number	number of items in the updated list.
updated.object[*number*]	object type and its name in the updated list. For example, *Line[4]*, *Arc[1].* If *n* items are updated, then *number* is the item's number in the updated list.

updated.object[_number_**].**_syntax_	object information as specified by the _syntax_ for **object updated.object[number].** The _syntax_ you can use is given under each type of object.
	For example, if **updated.object[1]** is _Line[2]_, then you can specify the _syntax_ as any syntax after **Line[name]**. For further details see Line (see page 178). For the x coordinate of the start of the line, you can use **updated.object[1].start.x** where **start.x** is the syntax.
updated.type[_number_**]**	type of an object in the updated list. For example, _Line_, _Arc_.
	If _n_ objects are updated, then _number_ is the item's number in the updated list.
	When you compare the type of an object with a text string, you must use the correct capitalisation. For example, to check that **updated.type[0]** is a composite curve, use:
	updated.type[0] == 'Composite Curve'
	not:
	updated.type[0] == 'Composite curve'
	updated.type[0] == 'composite curve'
updated.name[_number_**]**	name of an item in the updated list.
	If n items are updated, then _number_ is the item's number in the updated list.
	In all cases _number_ is from 0 to (n-1).

User commands

Command	Description
user	details of the user currently using PowerShape. It is output in the following form: user login : user name : start macro : security level
user.login	login of the user currently using PowerShape.
user.name	name of the user currently using PowerShape.
user.macro	pathname of the login macro of the user currently using PowerShape.
user.security	security level of the current user using PowerShape.

Product version commands

Command	Description
version	version of PowerShape that is being used, for example, 18121.
version.major	first digits of the version of PowerShape being used. For example, if you are using 18121, version.major returns 18.
version.minor	middle digit of the version of PowerShape being used. For example, if you are using 18121, version.minor returns 1.
version.revision	last two digits of the version of PowerShape being used. For example, if you are using 18121, version.revision returns 21.
version.has.excel	tests if MS Excel is installed.

View commands

Command	Description
view[*name*].exists	*1* if the view exists; *0* otherwise.
view[*name*].id	unique identity number of the view.
view[id *n*].name	name of the view that has the given identity number.
view[*name*].rotation_centre	[x y z] coordinates of the rotation centre of the view.
view[*name*].rotation_centre.x	x coordinate of the rotation centre of the view.
view[*name*].rotation_centre.y	y coordinate of the rotation centre of the view.
view[*name*].rotation_centre.z	z coordinate of the rotation centre of the view.

Window commands

Command	Description
cwindow clear	clears the Command window.
window.selected	number of the selected window.
window.number	number of windows opened.
window[*name*].exists	*1* if the window exists; *0* otherwise.
window[*name*].id	unique identity number of the window.
window[*name*].size	size of the window in x and y.
window[*name*].size.x	size of the window in x.
window[*name*].size.y	size of the window in y.
window[*name*].type	type of the window from one of the following: model or drawing.
window[*name*].model	name of the model opened in the window.
window[*name*].drawing	name of the drawing if opened in the window and a blank string otherwise.

Workplane commands

If you don't specify the name of the workplane, the active workplane is used; for example, **workplane.origin** returns the origin of the active workplane. An error is displayed when there is no active workplane.

The following workplane commands are available:

Command	Description
workplane[*name***].active**	*1* if the workplane is active; *0* otherwise.
workplane.active	name of the active workplane. If no workplane is active, *World* is returned, even in a foreign language.
workplane[*name***].xaxis**	unit vector which defines the orientation of the X-axis of workplane from its origin.
workplane[*name***].xaxis.x**	x value of the unit vector which defines the orientation of the X-axis of workplane from its origin.
workplane[*name***].xaxis.y**	y value of the unit vector which defines the orientation of the X-axis of workplane from its origin.
workplane[*name***].xaxis.z**	z value of the unit vector which defines the orientation of the X-axis of workplane from its origin.
workplane[*name***].yaxis**	unit vector which defines the orientation of the Y-axis of workplane from its origin.
workplane[*name***].yaxis.x**	x value of the unit vector which defines the orientation of the Y-axis of workplane from its origin.
workplane[*name***].yaxis.y**	y value of the unit vector which defines the orientation of the Y-axis of workplane from its origin.
workplane[*name***].yaxis.z**	z value of the unit vector which defines the orientation of the Y-axis of workplane from its origin.
workplane[*name***].zaxis**	unit vector which defines the orientation of the Z-axis of workplane from its origin.
workplane[*name***].zaxis.x**	x value of the unit vector which defines the orientation of the Z-axis of workplane from its origin.

workplane[*name***].zaxis.y**	y value of the unit vector which defines the orientation of the Z-axis of workplane from its origin.
workplane[*name***].zaxis.z**	z value of the unit vector which defines the orientation of the Z-axis of workplane from its origin.
workplane[*name***].exists**	*1* if the workplane exists; *0* otherwise.
workplane[*name***].id**	unique identity number of the workplane in the model.
workplane[id *n*].name	name of the workplane that has the given identity number.
workplane[*name***].level**	level on which the workplane exists.
workplane[*name***].locked**	*1* if the workplane is locked; *0* otherwise.
workplane[*name***].origin**	coordinates [x, y, z] of the origin of the workplane.
workplane[*name***].origin.x**	x coordinate of the origin of the workplane.
workplane[*name***].origin.y**	y coordinate of the origin of the workplane.
workplane[*name***].origin.z**	z coordinate of the origin of the workplane.
workplane[*name***].style.colour**	colour number of line style used to draw the workplane.
workplane[*name***].style.color**	color (USA) number of line style used to draw the workplane.
workplane[*name***].style.gap**	gap of line style used to draw the workplane.
workplane[*name***].style.weight**	weight of line style used to draw the workplane.
workplane[*name***].style.width**	width of line style used to draw the workplane.

Autodesk Legal Notice

a. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(g) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c. **"Distribute"** means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

d. **"License Elements"** means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, Noncommercial, ShareAlike.

e. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

f. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

g. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

h. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

i. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

j. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

d. to Distribute and Publicly Perform Adaptations.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights described in Section 4(e).

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(d), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(d), as requested.

b. You may Distribute or Publicly Perform an Adaptation only under: (i) the terms of this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-NonCommercial-ShareAlike 3.0 US) ("Applicable License"). You must include a copy of, or the URI, for Applicable License with every copy of each Adaptation You Distribute or Publicly Perform. You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License. You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

c. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

d. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, (iv) consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(d) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

e. For the avoidance of doubt:

i. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,

iii. Voluntary License Schemes. The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(c).

f. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING AND TO THE FULLEST EXTENT PERMITTED BY APPLICABLE LAW, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THIS EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

Creative Commons Notice

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of this License.

Creative Commons may be contacted at https://creativecommons.org/.

Certain materials included in this publication are reprinted with the permission of the copyright holder.

Creative Commons FAQ

Autodesk's Creative Commons FAQ can be viewed online at https://knowledge.autodesk.com/customer-service/share-the-knowledge, and is reproduced here for offline users.

Creative Commons is a simple, open licensing model which allows individuals to freely modify, remix, and share digital content created for learning and support.

Borrow from the Autodesk Learning, Support and Video libraries to build a new learning experience for anyone with any particular need or interest. It's out there. You can use it. It's yours.

In collaboration with Creative Commons, Autodesk invites you to share your knowledge with the rest of the world, inspiring others to learn, achieve goals, and ignite creativity.

What is Creative Commons?
Creative Commons (CC) is a nonprofit organization that offers a simple licensing model that frees digital content to enable anyone to modify, remix, and share creative works.

How do I know if Autodesk learning content and Autodesk University content is available under Creative Commons?
All Autodesk learning content and Autodesk University content released under Creative Commons is explicitly marked with a Creative Commons icon specifying what you can and cannot do. Always follow the terms of the stated license.

What Autodesk learning content is currently available under Creative Commons?
Over time, Autodesk will release more and more learning content under the Creative Commons licenses.

Currently available learning content:

- Autodesk online help-Online help for many Autodesk products, including its embedded media such as images and help movies.

- Autodesk Learning Videos-A range of video-based learning content, including the video tutorials on the Autodesk YouTube™ Learning Channels and their associated iTunes® podcasts.

- Autodesk downloadable materials-Downloadable 3D assets, digital footage, and other files you can use to follow along on your own time.

Is Autodesk learning and support content copyrighted?
Yes. Creative Commons licensing does not replace copyright. Copyright remains with Autodesk or its suppliers, as applicable. But it makes the terms of use much more flexible.

What do the Autodesk Creative Commons licenses allow?
Autodesk makes some of its learning and support content available under two distinct Creative Commons licenses. The learning content is clearly marked with the applicable Creative Commons license. You must comply with the following conditions:

- **Attribution-NonCommercial-ShareAlike (CC BY-NC-SA)** This license lets you copy, distribute, display, remix, tweak, and build upon our work noncommercially, as long as you credit Autodesk and license your new creations under the identical terms. Terms of this license can be viewed online at https://creativecommons.org/licenses/by-nc-sa/3.0/us/

- **Attribution-NonCommercial-No Derivative Works (CC BY-NC-ND)** This license lets you copy, distribute, and display only verbatim copies of our work as long as you credit us, but you cannot alter the learning content in any way or use it commercially. Terms of this license can be viewed online at https://creativecommons.org/licenses/by-nc-nd/3.0/us/deed.en_US

- **Special permissions on content marked as No Derivative Works** For video-based learning content marked as No Derivative Works (ND), Autodesk grants you special permission to make modifications but only for the purpose of translating the video content into another language.

These conditions can be modified only by explicit permission of Autodesk, Inc. Send requests for modifications outside of these license terms to *creativecommons@autodesk.com*.

Can I get special permission to do something different with the learning content?

Unless otherwise stated, our Creative Commons conditions can be modified only by explicit permission of Autodesk, Inc. If you have any questions or requests for modifications outside of these license terms, email us at creativecommons@autodesk.com.

How do I attribute Autodesk learning content?

You must explicitly credit Autodesk, Inc., as the original source of the materials. This is a standard requirement of the Attribution (BY) term in all Creative Commons licenses. In some cases, such as for the Autodesk video learning content, we specify exactly how we would like to be attributed.

This is usually described on the video's end-plate. For the most part providing the title of the work, the URL where the work is hosted, and a credit to Autodesk, Inc., is quite acceptable. Also, remember to keep intact any copyright notice associated with the work. This may sound like a lot of information, but there is flexibility in the way you present it.

Here are some examples:

"This document contains content adapted from the Autodesk® Maya® Help, available under a Creative Commons Attribution-NonCommercial-Share Alike license. Copyright © Autodesk, Inc."

"This is a Finnish translation of a video created by the Autodesk Maya Learning Channel @ www.youtube.com/mayahowtos. Copyright © Autodesk, Inc."

"Special thanks to the Autodesk® 3ds Max® Learning Channel @ www.youtube.com/3dsmaxhowtos. Copyright © Autodesk, Inc."

Do I follow YouTube's standard license or Autodesk's Creative Commons license?

The videos of the Autodesk Learning Channels on YouTube are uploaded under YouTube's standard license policy. Nonetheless, these videos are released by Autodesk as Creative Commons Attribution-NonCommercial-No Derivative Works (CC BY-NC-ND) and are marked as such.

You are free to use our video learning content according to the Creative Commons license under which they are released.

Where can I easily download Autodesk learning videos?
Most of the Autodesk Learning Channels have an associated iTunes podcast from where you can download the same videos and watch them offline. When translating Autodesk learning videos, we recommend downloading the videos from the iTunes podcasts.

Can I translate Autodesk learning videos?
Yes. Even though our learning videos are licensed as No Derivative Works (ND), we grant everyone permission to translate the audio and subtitles into other languages. In fact, if you want to recapture the video tutorial as-is but show the user interface in another language, you are free to do so. Be sure to give proper attribution as indicated on the video's Creative Commons end-plate. This special permission only applies to translation projects. Requests for modifications outside of these license terms can be directed to creativecommons@autodesk.com.

How do I let others know that I have translated Autodesk learning content into another language?
Autodesk is happy to see its learning content translated into as many different languages as possible. If you translate our videos or any of our learning content into other languages, let us know. We can help promote your contributions to our growing multilingual community. In fact, we encourage you to find creative ways to share our learning content with your friends, family, students, colleagues, and communities around the world. Contact us at creativecommons@autodesk.com.

I have translated Autodesk learning videos into other languages. Can I upload them to my own YouTube channel?
Yes, please do and let us know where to find them so that we can help promote your contributions to our growing multilingual Autodesk community. Contact us at creativecommons@autodesk.com.

Can I repost or republish Autodesk learning content on my site or blog?
Yes, you can make Autodesk learning material available on your site or blog as long as you follow the terms of the Creative Commons license under which the learning content is released. If you are simply referencing the learning content as-is, then we recommend that you link to it or embed it from where it is hosted by Autodesk. That way the content will always be fresh. If you have translated or remixed our learning content, then by all means you can host it yourself. Let us know about it, and we can help promote your contributions to our global learning community. Contact us at creativecommons@autodesk.com.

Can I show Autodesk learning content during my conference?
Yes, as long as it's within the scope of a noncommercial event, and as long as you comply with the terms of the Creative Commons license outlined above. In particular, the videos must be shown unedited with the exception of modifications for the purpose of translation. If you wish to use Autodesk learning content in a commercial context, contact us with a request for permission at creativecommons@autodesk.com.

Can I use Autodesk learning content in my classroom?
Yes, as long as you comply with the terms of the Creative Commons license under which the learning material is released. Many teachers use Autodesk learning content to stimulate discussions with students or to complement course materials, and we encourage you to do so as well.

Can I re-edit and remix Autodesk video learning content?
No, but for one exception. Our Creative Commons BY-NC-ND license clearly states that "derivative works" of any kind (edits, cuts, remixes, mashups, and so on) are not allowed without explicit permission from Autodesk. This is essential for preserving the integrity of our instructors' ideas. However, we do give you permission to modify our videos for the purpose of translating them into other languages.

Can I re-edit and remix Autodesk downloadable 3D assets and footage?
Yes. The Autodesk Learning Channels on YouTube provide downloadable 3D assets, footage, and other files for you to follow along with the video tutorials on your own time. This downloadable material is made available under a Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA) license. You can download these materials and experiment with them, but your remixes must give us credit as the original source of the content and be shared under the identical license terms.

Can I use content from Autodesk online help to create new materials for a specific audience?
Yes, if you want to help a specific audience learn how to optimize the use of their Autodesk software, there is no need to start from scratch. You can use, remix, or enrich the relevant help content and include it in your book, instructions, examples, or workflows you create, then Share-Alike with the community. Always be sure to comply with the terms of the Creative Commons license under which the learning content is released.

What are the best practices for marking content with Creative Commons Licenses?

When reusing a CC-licensed work (by sharing the original or a derivative based on the original), it is important to keep intact any copyright notice associated with the work, including the Creative Commons license being used. Make sure you abide by the license conditions provided by the licensor, in this case Autodesk, Inc.

Trademarks

The following are registered trademarks or trademarks of Autodesk, Inc., and/or its subsidiaries and/or affiliates in the USA and other countries: 3ds Max, ADSK, Alias, ATC, AutoCAD LT, AutoCAD, Autodesk, Autodesk Construction Cloud, Autodesk Forge, Autodesk Fusion 360, BIM 360, BuildingConnected, Civil 3D, Dancing Baby, The (image) Eagle, FBX, FeatureCAM, Flame, FormIt, Forge, Forge Devcon, Forge Fund, Fusion 360, Glue, Green Building Studio, ICMLive, InfoWater, InfoWorks, InfraWorks, Innovyze, Instructables, Inventor, Make Anything, Maya, Moldflow, MotionBuilder, Mudbox, Navisworks, Netfabb, PartMaker, Plangrid, PowerInspect, PowerMill, PowerShape, Pype, RasterDWG, Redshift, RealDWG, ReCap, Revit, Shotgun, SketchBook, Spacemaker, Tinkercad, TrustedDWG, VRED.

All other brand names, product names or trademarks belong to their respective holders.

Disclaimer

THIS PUBLICATION AND THE INFORMATION CONTAINED HEREIN IS MADE AVAILABLE BY AUTODESK, INC. "AS IS." AUTODESK, INC. DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.