PowerSHAPE 2015 R2

Reference Help

Customising PowerSHAPE

PowerSHAPE

Acknowledgements

This documentation references a number of registered trademarks and these are the property of their respective owners. For example, Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States.

Patent Information

Emboss functionality is subject to patent number GB 2389764 and patent applications US 10/174524 and GB 2410351.

Morphing functionality is subject to patent application GB 2401213.

PowerSHAPE 2015 R2. Published on 09 March 2015

Contents

Customising PowerSHAPE

You can customise PowerSHAPE functionality by creating:

- Macros
- OLE applications

Customising PowerSHAPE means making PowerSHAPE behave how you want it to.

As PowerSHAPE is continuously enhanced by adding many great features, it is impossible to cater for the needs of every single user. One solution is for you to create your own applications using either macros or add-in applications.

We encourage you to tell us about any new features you want in PowerSHAPE.

Introduction to customising PowerSHAPE

Customising PowerSHAPE divides into two broad sections:

- Macros (see page 5)
- OLE applications (see page 136)
  - HTML-based
  - Add-in

Macros and add-in applications use object information (see page 153).

There are also two tutorials to help you:

- Macro tutorial (see page 52)

- HTML application tutorial (see page 109)

Macros

A macro file is a file stored on disk, which contains commands and comments. The main use of a macro file is to store often-used or complicated sequences of commands for repeated use.

When you have mastered writing macro files, you greatly enhance the power and flexibility of PowerSHAPE and can tailor the software for your personal use.

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

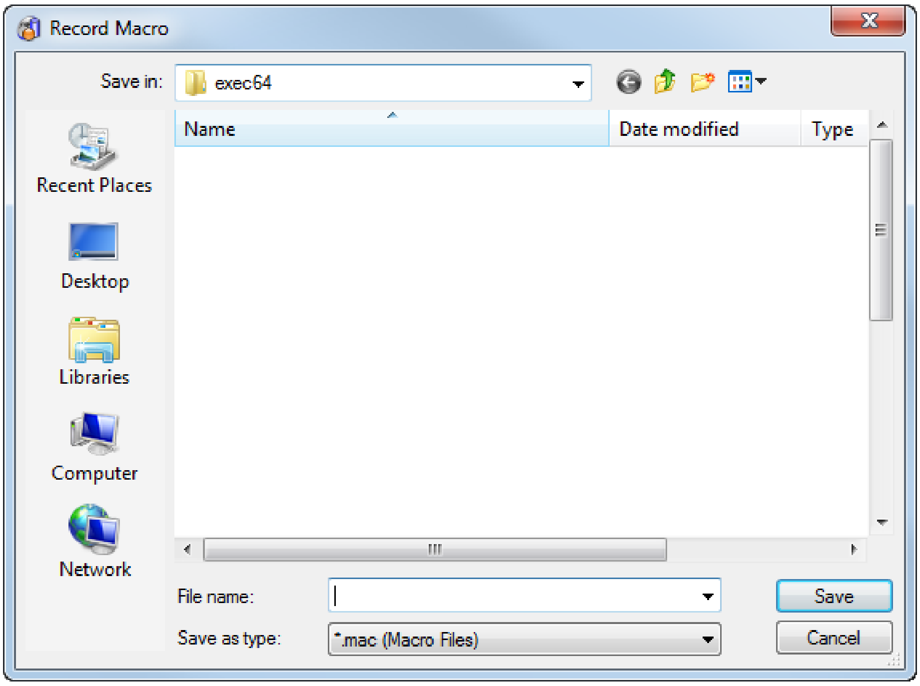
You can create macros by

- recording sequences of commands as you use PowerSHAPE (see page 5).
- writing your own macro using a text editor (see page 10).

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.

1 Select **Macro > Record** to display the Record Macro dialog (see page 7).



2 In the **File name** text 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.

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

4 Now work through the set of commands you want to record.

5 To stop recording the macro, select **Macro > Record**. You can use any text editor to view and edit a macro.

If you record a macro of **Paint Triangles** or the commands on the **Mesh Fixing and Editing** toolbar, an extra macro file is created. This file is named with the following convention:

<psmacroname>_cc_<nnnn>.mac

where

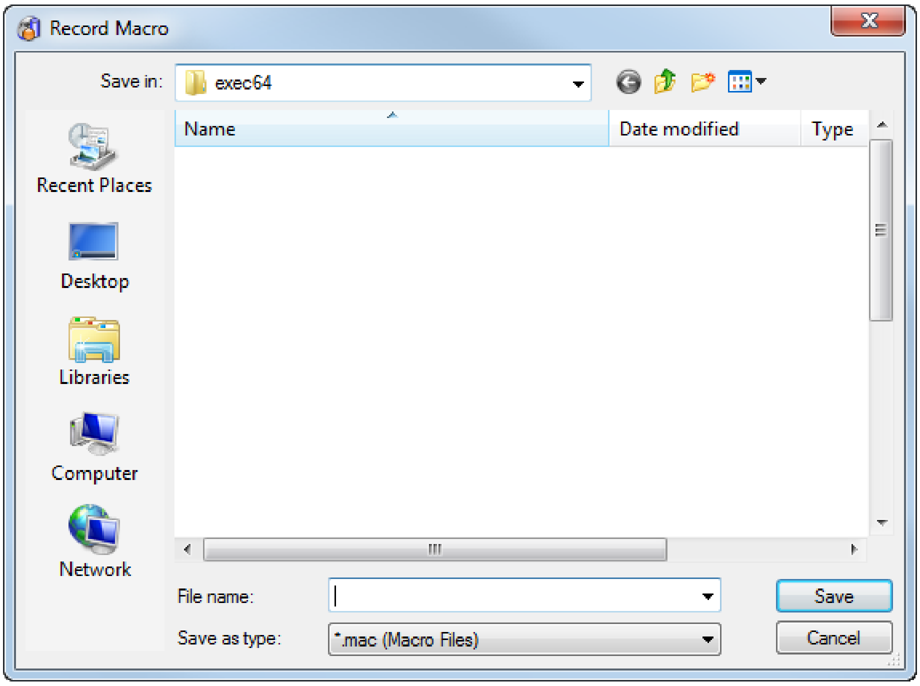
<psmacroname> is the name of macro being recorded.

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

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

Record Macro dialog

Use this dialog to specify or choose a file to record the macro to.



Save in — Select the correct directory.

 — Go up one level in the folder structure.

 — Create a new folder.

 — Display a menu containing options on how to display the files in the dialog.

File name — Select a file from the drop-down list, or enter the name of the file.

Save as type — This displays the filter pattern which filters the file names of the current directory. By default, the pattern is ***.mac** which displays all files with the extension *.mac*.

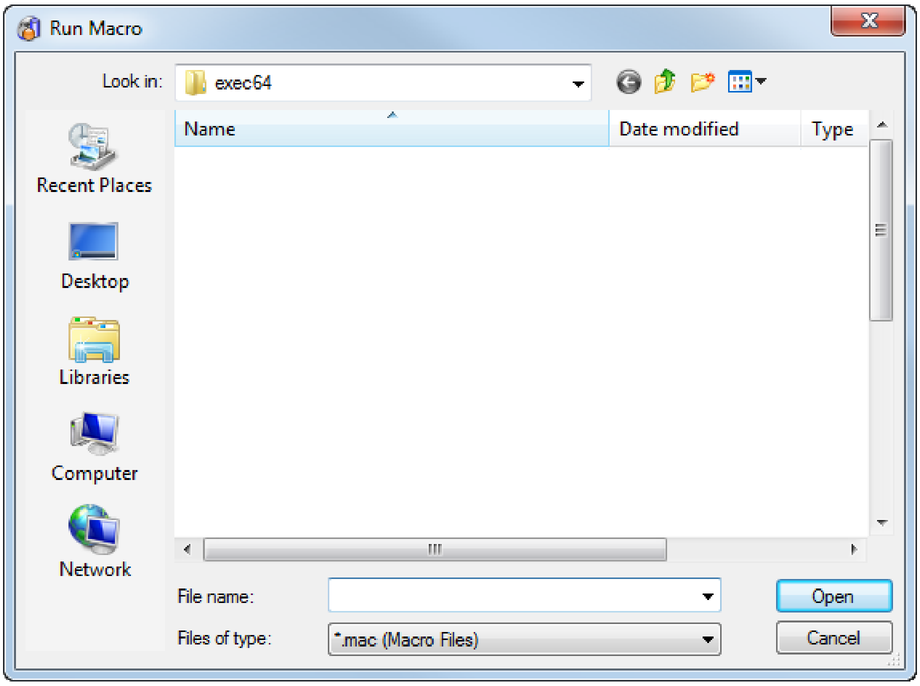
Save — Save the macro file.

Cancel — Cancel macro recording and close the dialog.

Running macros

Select this option to run a previously recorded macro.

1 Select **Macro > Run** to display the **Select A Macro To Run** dialog.



2 Select the macro you want to run. Its name appears in the **File name** text box.

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

You can run the macro in one go or 'step through' the commands (see page 8).

How do I stop a running macro?

You can abandon a macro while it is running. For example, you may realise the wrong macro is running or there is an error in your macro.

Press 

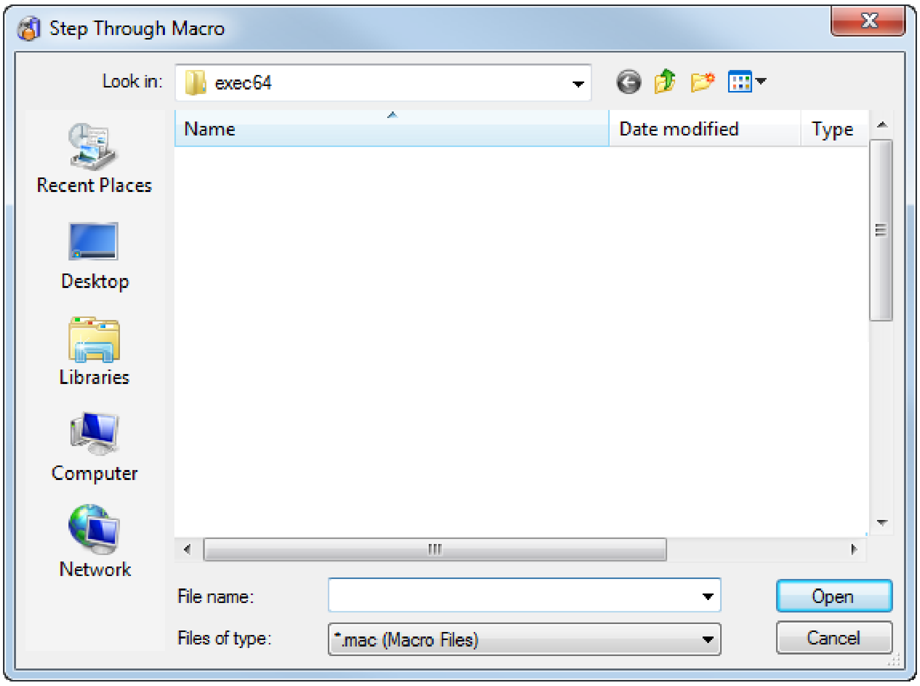
once. The macro finishes the command it is currently processing and stops.

Running a macro one command at a time

PowerSHAPE enables you to run a macro one command at a time and then pauses. You can then run the next command in the macro. This is known as stepping a macro.

Stepping through a macro enables you to check its commands.

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



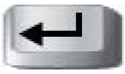
2 Use the dialog to select the filename of the macro.

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

The command window is displayed at the bottom of the screen.

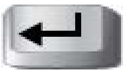
The first command line of the macro is listed in the command window:

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

4 In the command window, press the **Enter**  key to carry out the command and continue to the next command.

The next command is printed out in the command window:

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

5 Press the **Enter**  key until the macro finishes.

How do I stop a stepping macro?

To stop stepping through a macro, select **Macro > Abandon** from the menu.

Writing macros

When you write your own macros, they can be more elaborate than recorded macros. For example, you can add comments or add testing conditions. The following table gives examples of additional commands that can be added when you write, rather than record, a macro.

`$$` `//`	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
	store information in variables
	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 99)
`skip`	skip a block of commands
`input free` `execute pause`	pause a running macro
`return`	end a macro

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
3. Run the macro.

Finding out PowerSHAPE commands

When you use PowerSHAPE by entering menu clicks and filling in dialogs, commands are being sent to the program by the menus and dialogs. These are the commands that must be entered in your macro file if you want to drive PowerSHAPE from a macro instead of from the menus.

To find the commands to use in a macro:

1. Record a macro of the operations you wish 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.

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 will help anyone examining the file to understand it. 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. It may be obvious what each section does when you write the file but if you examine it in 3 months time they may be difficult to understand.

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, this is NOT allowed:

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

However you are allowed to 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. Do this by putting $$ or // at the beginning of the line which contains the command you wish to remove. For example,

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

This is known as *commenting out* a command.

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.

Variable types

A variable type is the type of information stored by the variable. A variable can have only one type. Its type is decided when you first use the variable and it cannot be changed.

The following types exist:

- 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 todetermine the type of an expression or variable :

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

The command returns a string that is:

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

Examples

```
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.

Assigning values

You can define a variable and assign it a value (see page 25). The following defines variable *bolts* with type *integer* and assigns it a value of 5.

```
INT $bolts = 5
```

You can assign values to variables by performing complex calculations.

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 $n= 'joe'`	initialises the *$*n variable to '*joe*'
`create arc full`	
`0 0 0`	
`select`	
`modify`	
`VAR_NAME $n`	
`accept`	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 47) into the environment of the calling macro using the `exportenv` variable

  ```
  exportenv path
  ```

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

Lower level macros have access to a copy of environment variables. They can change the contents of the variables, but those changes are discarded when the macro returns.

Creating user-input information into a macro

Most macro files are written requiring some user interaction. For example, the you might need to enter the position of an object or the dimensions of the object. 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 it is run.

  This implies that every time you wish to change the variables, you must open up the file in a text editor and change them. 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.

Prompting the user

The **INPUT** command is used where you want the user to enter information. You can ask for the user to enter one of the following:

- point (see page 16)
- selection of items (see page 17)
- number (see page 18)
- string (see page 18)
- yes or no response to a query (see page 19)

Point information

If you want the user to enter a point, use the following 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,

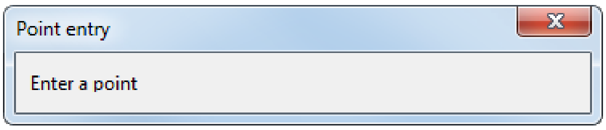
variable_name_y

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 in one of the following ways:

- 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'
```

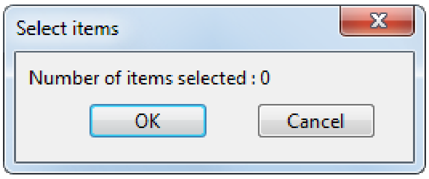
This command displays a dialog, which shows the number of objects selected. The characters in the string are used for the title on the dialog. When objects are selected, the number of objects selected are shown in the dialog.

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

Example

```
INPUT SELECTION 'Select items'
```

When a macro containing this command 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 153) 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 35)

`WHILE` loop (see page 40)

Number information

Use this command to let the user enter a number.

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

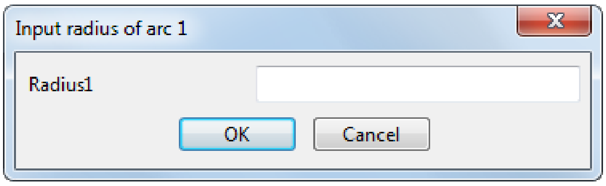
This command displays a dialog where:

- *'string'* is used as the dialog title.
- *variable_name* is the label of the text box.

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 to enter a string:

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

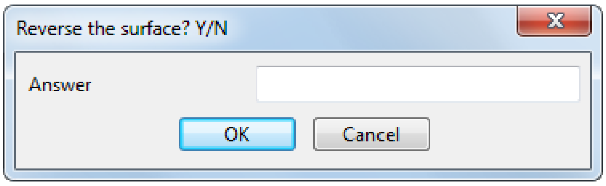
Like `INPUT NUMBER`, this command displays a dialog where:

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

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**. Tthe value is assigned to variable *Answer*.

Query information

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

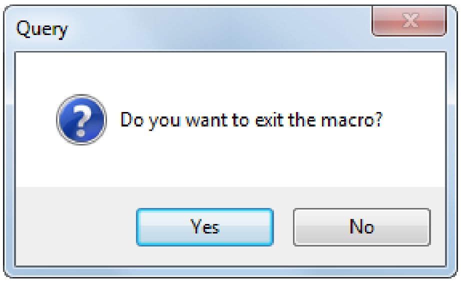
```
INPUT QUERY 'string' $variable_name
```

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*.

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.

```
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, select **View > Window > Command** or double-click the command box in the status bar.*

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.

Output from a macro

Use the following sections to output informaton from macros:

Displaying information (see page 21)

Displaying values of variables (see page 22)

Using an OUTFILE to display information (see page 22)

Example macro to generate and display a report file (see page 23)

Displaying information

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

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

Example

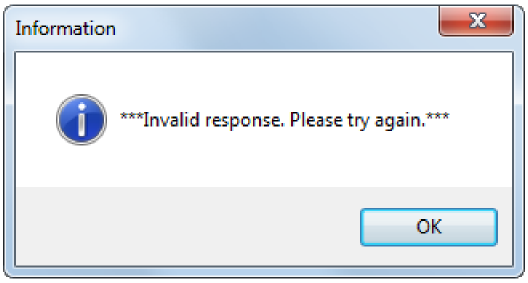
If a user provides an incorrect response, a macro displays an error message and prompts for another response:

```
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 remove the dialog from the screen, click **OK**.

Displaying the command window

Messages can be displayed in the command window or in dialogs. Use one of the following techniques to open the command window:

- Select **View > Window > Command**
- Double-click the command box in the status bar.

Users do not normally have the command window displayed.

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.

Examples

```
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..

Examples

```
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

Using an OUTFILE to display information

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

To produce a file

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.

The example macro to generate and display a report file (see page 23) uses the four sections.

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@delcam.com">someone@delcam.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
```

Exporting an image file

Use the following macro command in a macro to export an image file of a rendered image:

```
Render ToFile [replace] filename
```

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 = 'Delcam'
  // Print the first character 'D'
  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.

For further details, see:

Assigning values to variables - advanced users (see page 26)

Using expressions in macros (see page 29)

Assigning values to variables - advanced users

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 one of **INT, REAL, 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.

Using object information

You can assign object information to a macro variable (see page 153), 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:

- **a** object type
- **b** object name in square brackets
- **c** sub-object names

Suppose you have a line whose name is **2**, then all the information about line 2 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 34)
- 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 set of opertaors 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

Example 1

```
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, then you must use:

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

and not:

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

Example 2

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

Comparing variables - logical operators

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 (see page 30)which define 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

Examples:

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.

Operators

For each variable type, the operators perform various tasks.

Spaces may be included on either 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.

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.

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

When using the `IF` (see page 35) command, you can decide 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 wish 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.

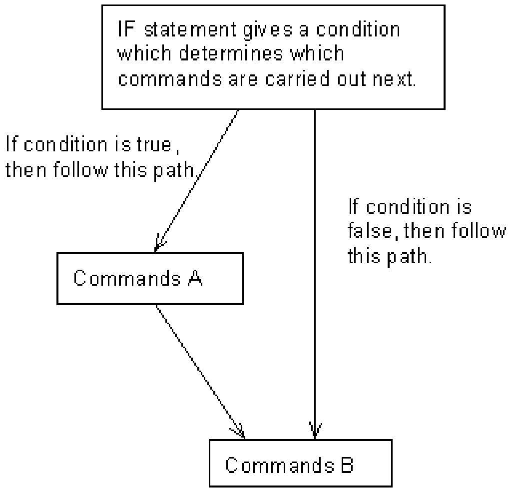
- if the value in invalid, tell the user that their input is invalid and ask them to enter another value.

IF

When a certain 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"
}
```

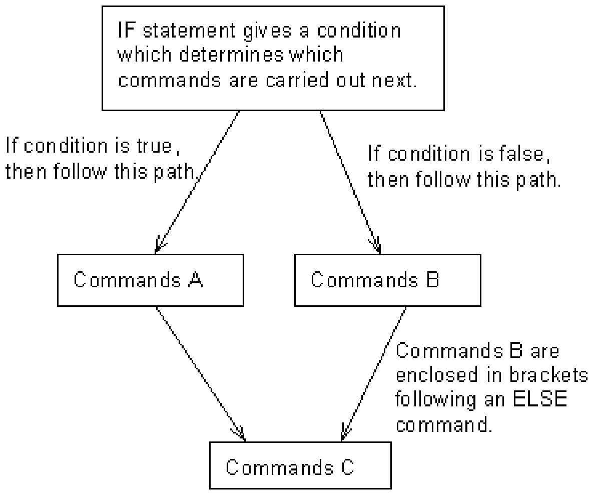
The first bracket must be the last item on the line and on the same line as the IF. The closing bracket must be on a line by itself.

You can also define commands that are only carried out when the condition is *false*. These commands are defined using the `IF-ELSE` (see page 36) and `IF-ELSEIF-ELSE` (see page 36) 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*.



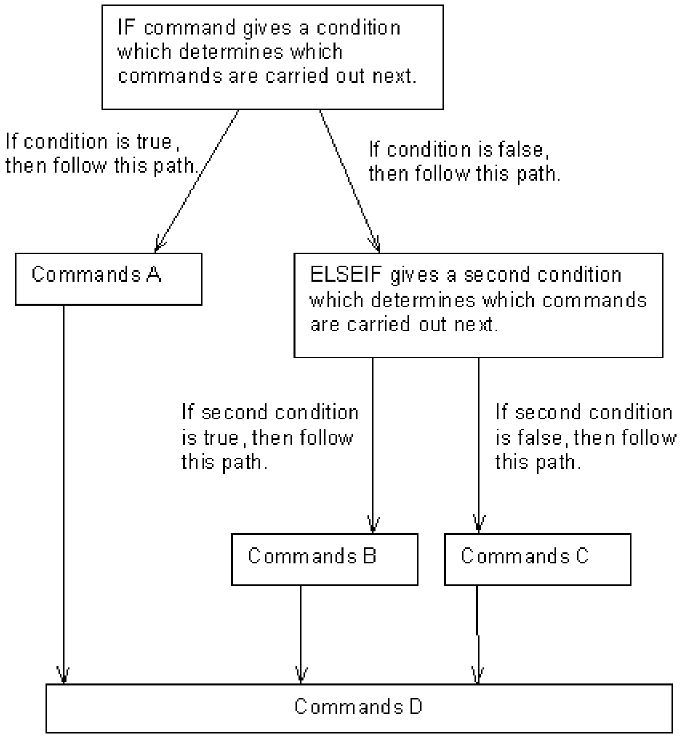
IF - ELSEIF - ELSE

```
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 may be any number of `ELSEIF` statements in a block but not more than one `ELSE`. `ELSEIF` may 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

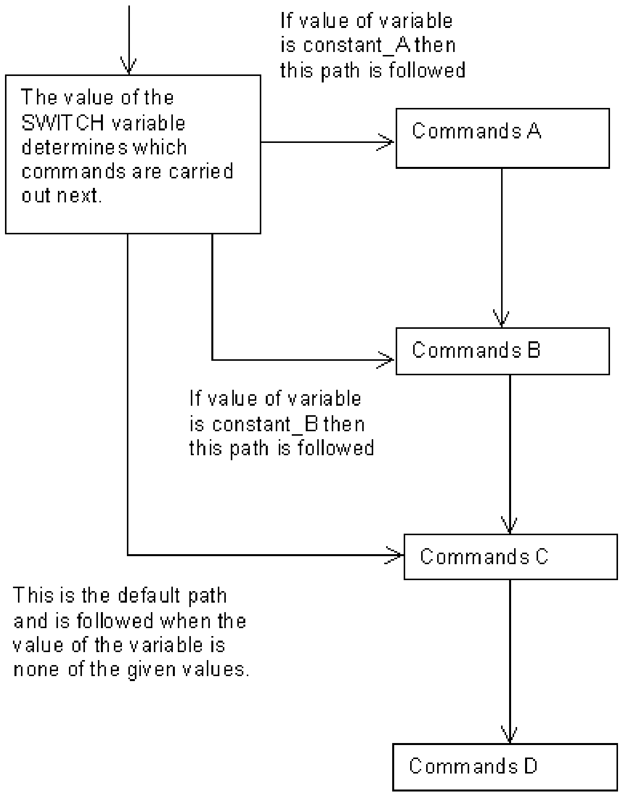
When you compare a variable with a number of possible values and each value determines a different outcome, it is recommended that you use the `SWITCH` command (see page 107).

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.

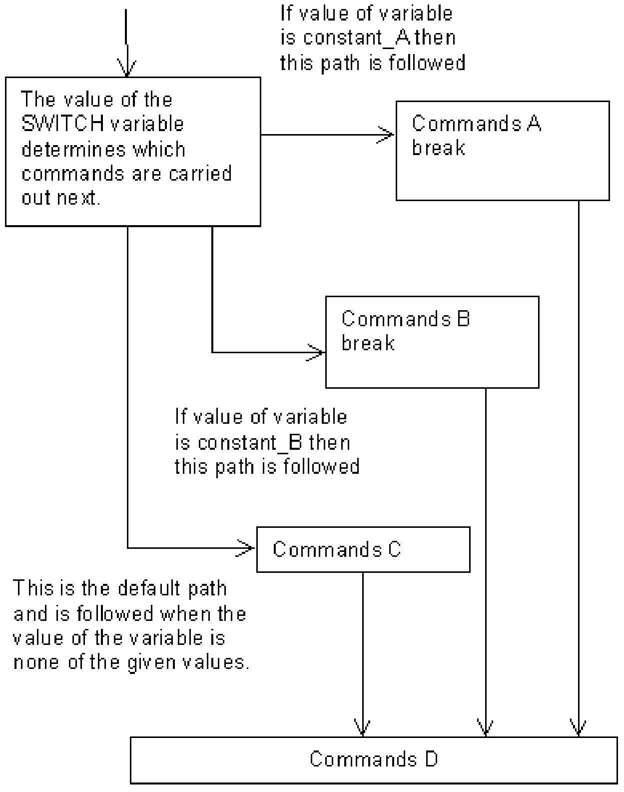


*The commands are executed through the **switch** command. Once a match is found all the commands in the remaining **case** statements are executed. You may prevent this from happening by using a **break** statement.*

```
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 a set of commands a number of times are known as *loops*. There are two loop structures

- WHILE loop (see page 40)
- DO - WHILE loop (see page 40)

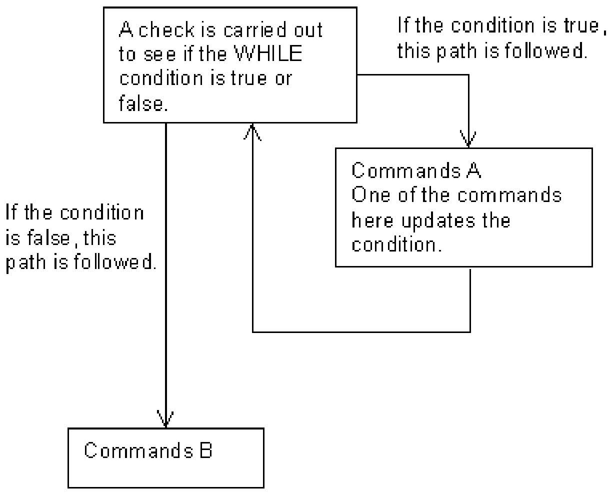
WHILE loop

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. Once 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 in order to:

- cancel the loop using the `BREAK` command
- continue with the next iteration using the `CONTINUE` command.

DO- WHILE loop

The `WHILE` loop checks its conditional test first to decide whether to carry out its commands, whereas the `DO-WHILE` loop carries out its commands and then checks its conditional test.

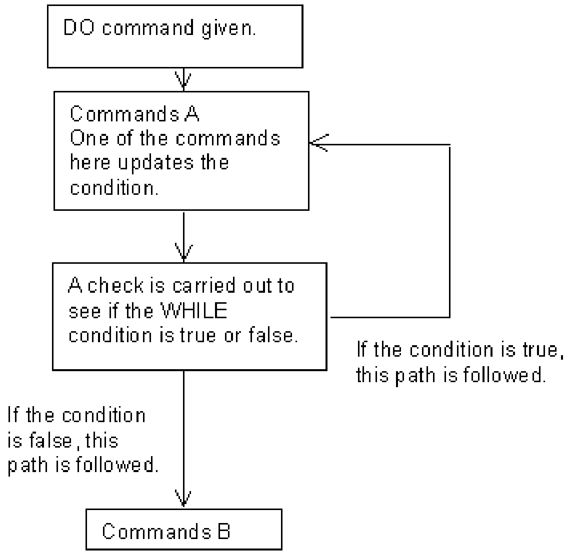
```
DO {
```

```
  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 Once 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 in order 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.

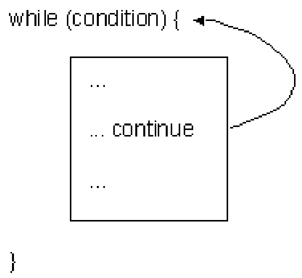
An example is given below.

```
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)
}
```

Example

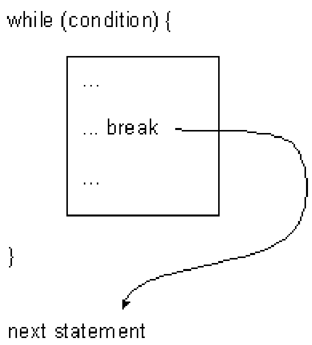
The user is asked to enter the number of holes. Before the calculation, you need to make sure that the number is valid. Using the CONTINUE command allows the user to enter the value again.



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 42) 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 define 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.

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
```

Example

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
```

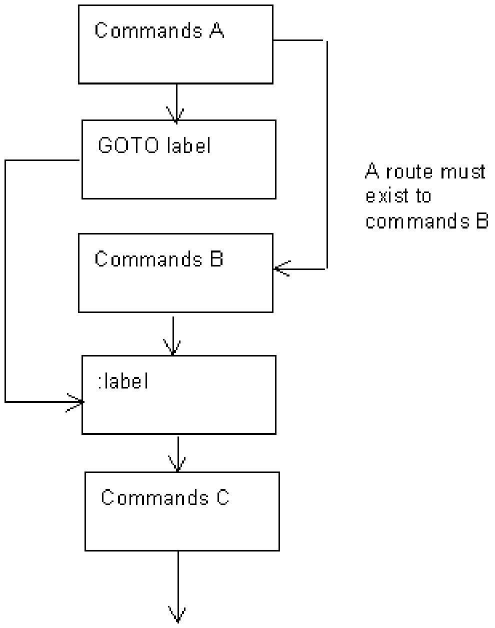
Labels

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

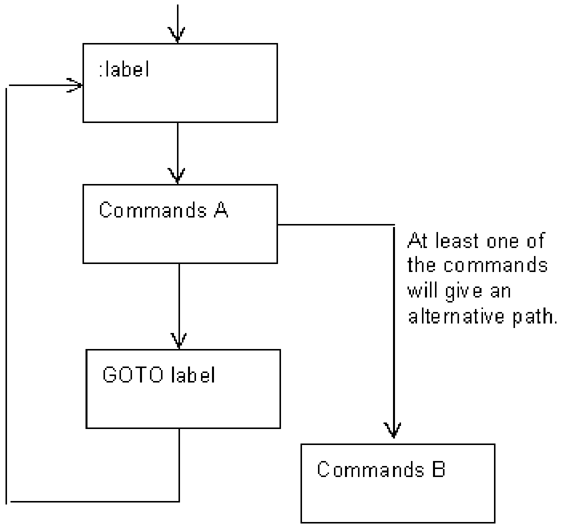
Use a label as follows:

- 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:

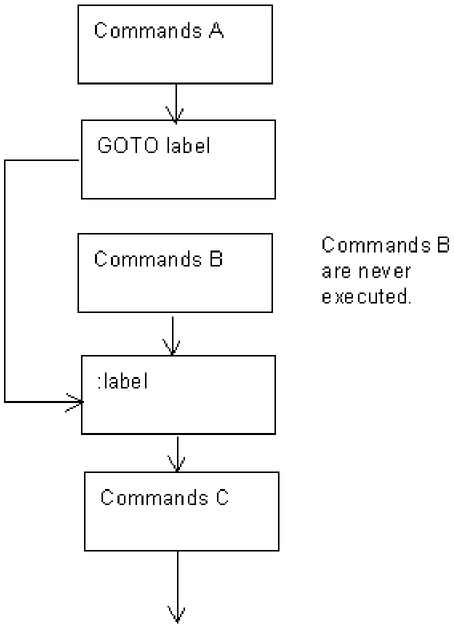


- Before a `GOTO` command

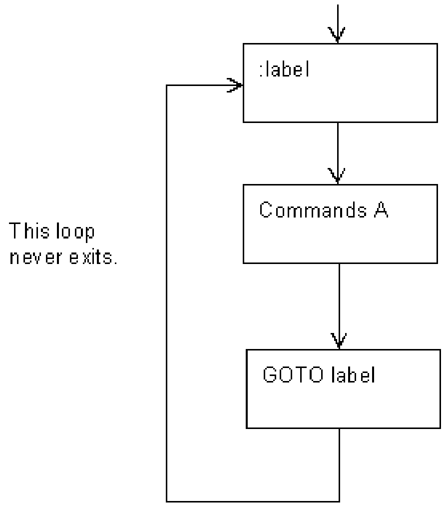


NOTE:

1. Ensure that a path exists to all the commands in the macro; otherwise you will have commands which are not used.



2. 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"
```

Passing values into a macro

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 19).

Passing expressions as arguments

You can 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 variable 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 display:

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 may wish to step certain commands in a macro whilst the macro is running.

To switch on stepping mode from within a macro at a particular point, use the command:

```
EXECUTE STEP
```

To switch off stepping, 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 (see page 49)
- a pause that waits for the user to press a button to continue the macro (see page 50).

Pause for predefined time

You can 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.

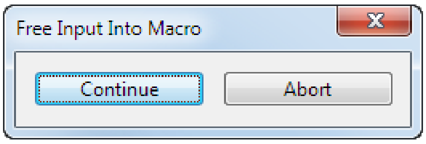
The command is:

```
EXECUTE PAUSE integer
```

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

Pause with a button to continue

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 doesn't exist, nothing will happen
- To add a composite curve,

  `ADD COMPCURVE fred AT COMPOSITE 3 KEYPOINT 5`

  If the keypoint doesn't exist, nothing will happen
- 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` surpresses the dialog.

`EXPORTOPTS IGNOREBADTRIMOFF` causes the message dialog to be displayed

Skipping command lines

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

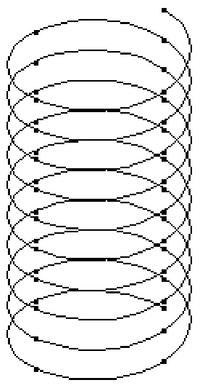
```
SKIP 17
```

Macro tutorial - Helix

Use the following sections to practise using the the macro commands :

Introduction to the helix macro

In this example, you will create a macro to create a helix.



We suggest you go through this example before attempting to create your own macros.

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

c:\dcam\product\powershapexxxx\file\examples\Macro_Writing

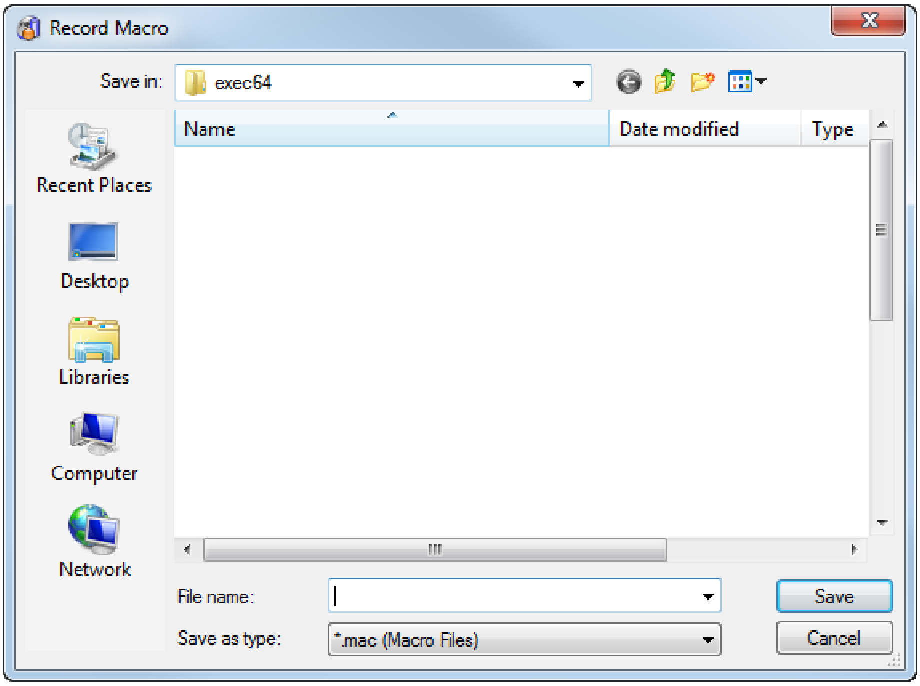
where **XXXX** is the version number of the software and **c** is the drive on which the software is installed.

Recording 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. Once you have the basic commands, you can enhance your macro.

1 Make sure you have a model open.

2 From the main menu, select **Macro** followed by **Record** to display the **Select A File To Record To** dialog.



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

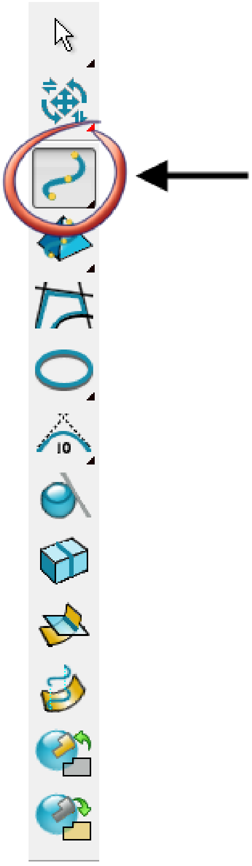
3 In the **File name** box, type

helix_turn.mac

4 Click **Save**.

5 From the **Main** toolbar, click **Curve** .

6 From the curve creation menu, click **Bezier Curve** .



.

This makes sure that the **Curve** option is selected when you run the macro.

7 Type in the co-ordinates of the points of the curve in the graphics window:

0 10 0

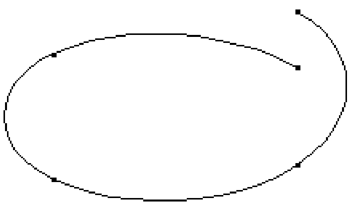
-10 10 1

-10 -10 1

10 -10 1

10 10 1

This creates a spiral shape.



8 Click **Select** . This exits curve creation.

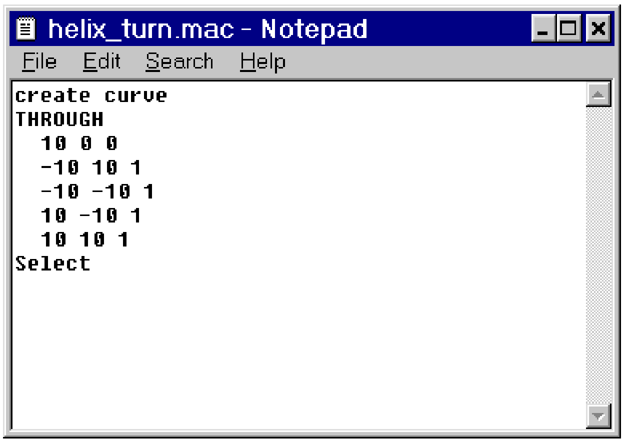
9 From the main menu, select **Macro** followed by **Record** to stop recording the macro. The **Record** option displayed ✔ to indicate a macro was being recorded.

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:\dcam\product\powershapexxxx\file\examples\Macro_Writing
where **XXXX** is the version number of software and **c** is the drive on which the software is installed.

For further details, see Running the macro (see page 56).

Viewing the text in the macro

Open the macro in a text editor, for example **Notepad.**



The command below tells the software to enter curve creation mode.

create curve

The command below selects the Curve option from the Curve creation menu.

THROUGH

The command below 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 command below exits curve creation mode and goes back to selection mode.

Select

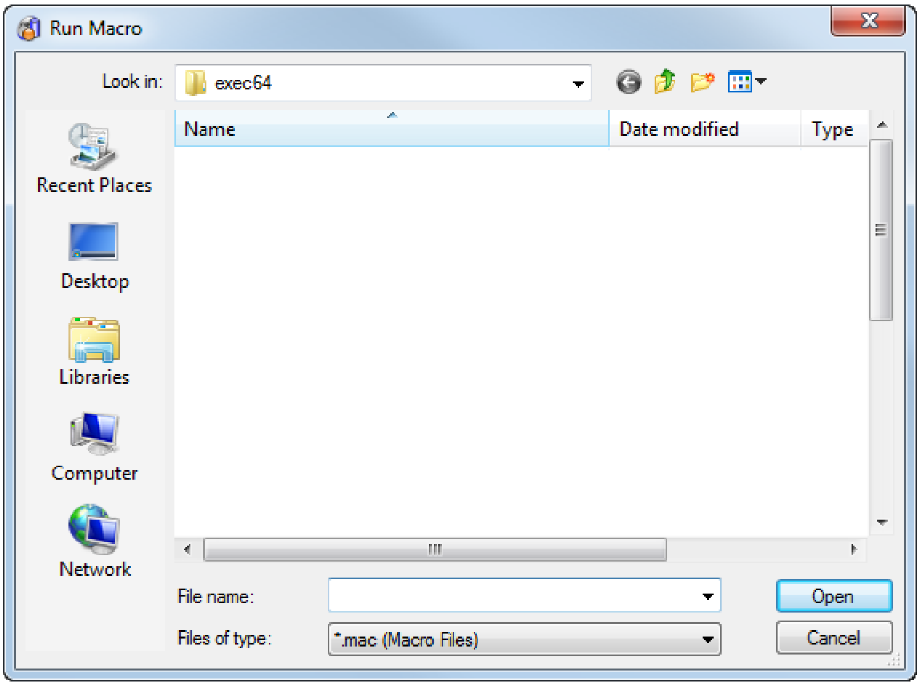
Running the macro

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 From the main menu, select **Macro** followed by **Run** to display the **Select A Macro To Run** dialog.



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 file in a text editor.

2 Edit the co-ordinates in your macro to:

50 0 0

-50 50 5

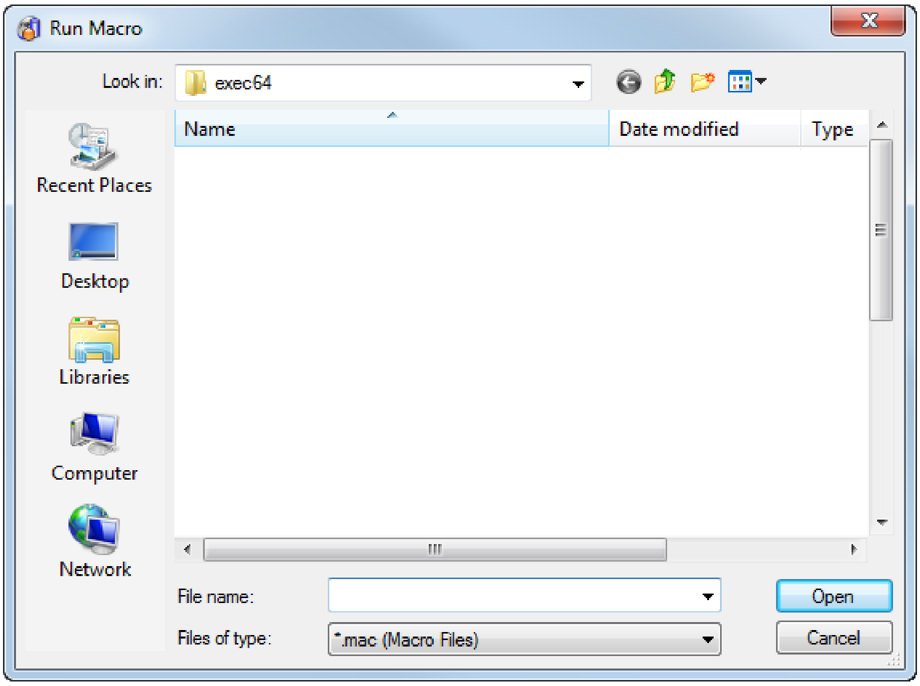
-50 -50 5

50 -50 5

50 50 5

3 Save the file.

4 From the main menu, select **Macro** followed by **Run** to display the **Select A Macro To Run** dialog.



5 Select your macro file.

6 Click **Open**.

This creates the required helix.

Adding variables

You may want to create a helix using different values. We will change the co-ordinates values in the macro to use variables.

The macro will then create a helix using the following variables:

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

Editing your macro

You can either:

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

  d:\dcam\product\powershapexxxx\file\examples\Macro_Writing

  where **XXXX** is the version number of the software and d is the drive on which the software is installed.

The changes are given in bold 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

More information on LET

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

LET $radius = 10

For each of the co-ordinates, we have replaced the value with a single variable. For example,

-50 50 5

has become

$neg_radius $radius $zheight

This makes it easier to change the values. Instead of changing all the co-ordinates each time we want to create a different size helix, we simply assign new values to the variables.

There are different variable for the negative and positive radius. The co-ordinate 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.

For further details, see Using expressions in macros (see page 29).

Run your macro that includes variables

1. Select your macro file or *helix_variable.mac*. For further details see, Running the macro (see page 56).
2. Click **Open**.

   This creates the required helix.

If you want to change the values of the radius and pitch, you simply open the macro file and edit two values in the macro. This saves time changing all the co-ordinate values.

Adding a loop

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

Editing your macro

You can either:

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

  d:\dcam\product\powershapexxxx\file\examples\Macro_Writing

where XXXX is the version number of the software and d is the drive on which the software is installed.

The changes are given in bold 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
```

More information on variable 'numturn'

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, then carry out the commands in the brackets, { }.*

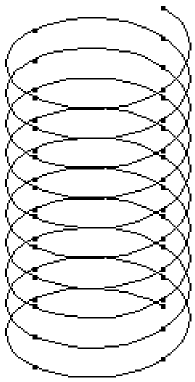
When the last bracket is reached, PowerSHAPE checks if numturn equals zero. If numturn does not equal zero, then the commands in the brackets {} are carried out again. If numturn equal zero then the commands below the last bracket are carried out.

Run your macro that includes a loop

1 Select your macro file or *helix_loop.mac*. For further details see, Running the macro (see page 56).

2 Click **Open.**

The helix now 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 put at the start of a line to show it is a comment. You can also use blank lines to separate blocks of commands.

Editing your macro

You can either:

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

  d:\dcam\product\powershapexxxx\file\examples\Macro_Writing

  where **XXXX** is the version number of the software and d is the drive on which the software is installed.

The changes are given in bold text.

// This macro creates a helix

// Written by: Razia Ghani

// 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
```

More information on adding comments

We have added commands such as:

```
// Calculating values for the co-ordinates
```

The two slashes // tell PowerSHAPE that this line contains a comment. The macro behaves the same with or without these comments added. The comments can remind you of what a block of commands does.

Run your macro that includes comments

1. Select your macro file or *helix_comments.mac.* For further details see, Running the macro (see page 56).
2. Click **Open**.

   The same helix is created as described in Run your macro that includes a loop (see page 60).

Interacting with the user

If you don't want to open the macro every time you create a helix with a difference size, you can display dialogs to enter values.

Editing your macro

You can either:

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

  d:\dcam\product\powershapexxxx\file\examples\Macro_Writing

where **XXXX** is the version number of the software and d is the drive on which the software is installed.

1 Comment out the following commands in your macro.

  // LET $radius = 10

  // LET $pitch = 4

  // LET $numturn = 10

  We don't need these commands, they will be replaced with new commands. You can leave the commands in your macros as comments, just in case you decide to use the commands 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

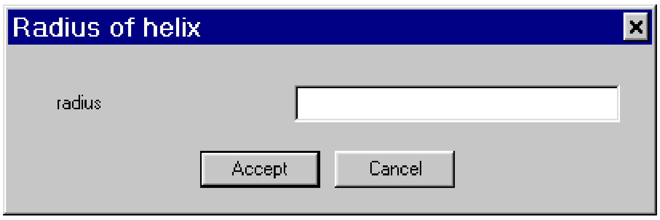
More information on interacting with the user

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.



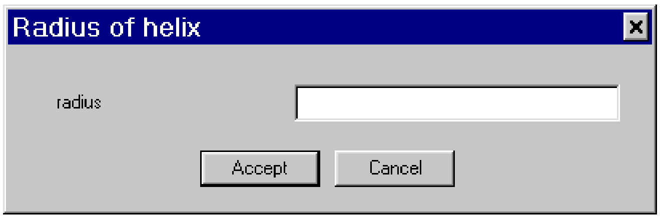
The string '**Radius of helix**' is the title of the dialog. When the user enters a value, it is assigned to the variable *$radius*. The name of the variable is on the left of the data box on the dialog.

Run your macro file that interacts with the user

1 Select your macro file or *helix_interact.mac.* For further details see, Running the macro (see page 56).

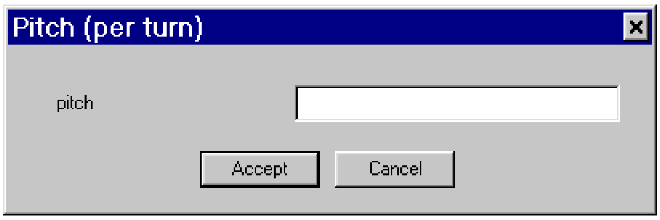
2 Click **Open**.

3 While the macro is running, the first dialog is displayed.



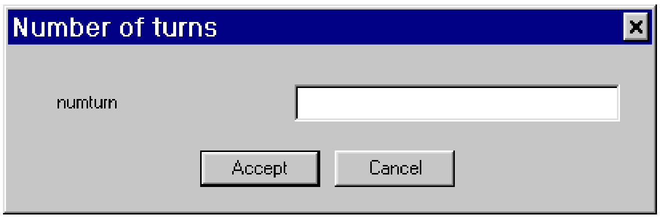
4 Enter a value and click **OK**.

5 The **Pitch** dialog is displayed.



6 Enter a value and click **OK**.

7 Finally the **Number of turns** dialog is displayed.



8 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.

Editing your macro

You can either:

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

  d:\dcam\product\powershapexxxx\file\examples\Macro_Writing

where XXXX is the version number of the software and d is the drive on which the software is installed.

The changes are given in bold text.

// This macro creates a helix

```
// Written by: Razia Ghani

// 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 dialogue box.

The dialogue box remains dislayed 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
```

Run your macro that changes the origin of the helix

1 Select your macro file or *helix_origin.mac*. For further details see, Running the macro (see page 56).

2 Click **Open**.

   The following dialog appears asking for you to input a position.

3 Click a point on the screen.

4 The three dialogs are displayed as described in Run your macro file that interacts with the user (see page 63).

5 Enter values in each and click **Accept**.

The helix is drawn on the screen.

Creating a helix around a cylinder

The helix is now constructed relative to a user-defined point. We want to extend the macro so that the helix is constructed around an existing primitive cylinder (surface).

When the macro is running, the user will select the cylinder. We will then ask the user:

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

The helix is then drawn around the cylinder.

The macro will also:

- Let the user select the cylinder.
- Create a temporary workplane at the workplane of the cylinder. The temporary workplane gives us the centre of the helix and the orientation of the workplane.

Editing your macro

You can either:

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

  d:\dcam\product\powershapexxxx\file\examples\Macro_Writing

where **XXXX** is the version number of the software and d is the drive on which the software is installed.

The changes are given in bold text.

```
// This macro creates a helix
// Written by: Razia Ghani

// Clear the selection list
SELECT CLEARLIST

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

// Displays dialogue boxes 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

More information on creating a helix around a cylinder

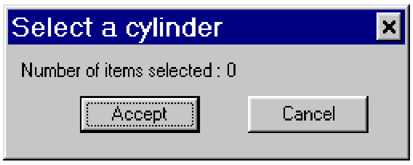
Before the cylinder is selected, we clear the selection list using the following command.

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 153).

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 70)

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:

  d:\dcam\product\powershapexxxx\file\examples\Macro_Writing

where **XXXX** is the version number of the software and d is the drive on which the software is installed.

The changes are given in bold text.

// This macro creates a helix

// Written by: Razia Ghani

// Clear the selection list

SELECT CLEARLIST

// Selecting a cylinder

INPUT SELECTION 'Select a cylinder'

LET cyl = selection.object[0]

// Displays dialogue boxes 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
```

More information on creating a helix around a cylinder

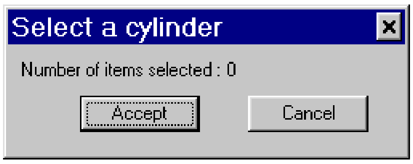
Before the cylinder is selected, we clear the selection list using the following command.

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 153).

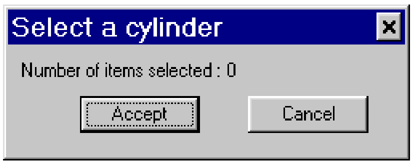
When you write macros, we advise you to assign the selected objects you want to use later in your macro to other variables. If the selection changes, you will obviously lose your selection.

For further details, see: Creating a workplane at the origin of the cylinder (see page 70)

Run your macro that creates a helix around a cylinder

1. Create a primitive cylinder (surface).
2. Select your macro file or *helix_cyl.mac*.For further details see, Running the macro (see page 56).
3. Click **Open**.

4 The following dialog is displayed asking for you to select a cylinder.



5 Select the primitive cylinder.

6 Click **Accept**.

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

8 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.

In our macro, we will check:

- 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, we tell the user that a single cylinder must be selected and then give an option to exit the macro. If the user decides to continue, they are asked to select a cylinder again.

We will also check if the helix is smaller or larger than the cylinder.

Run your macro file

We will run the macro to check if the tests work.

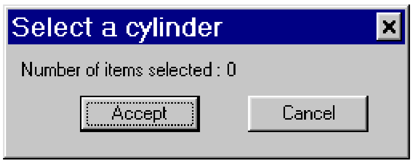
1 Create different objects in your model to test your macro. Make sure you have a primitive cylinder.

2 From the **Main** menu, select **Macro** followed by **Run** to display the **Select A Macro To Run** dialog.

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

4 Click **Open**.

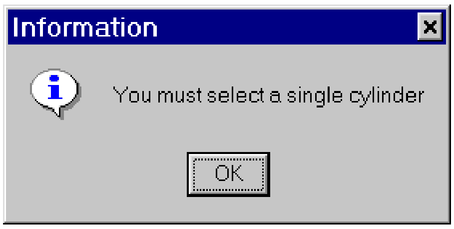
5 The following dialog appears asking for you to select a cylinder.



Select a couple of objects.

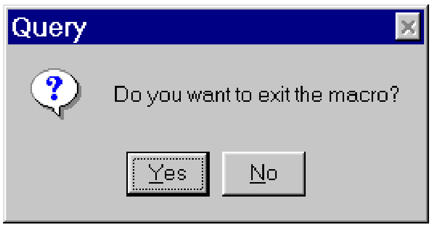
6 Click **OK**

7 The **Information** dialog appears telling you that a single cylinder must be selected.



8 Click **OK**.

9 The **Query** dialog appears asking if you want to exit the macro.



If you click **Yes**, the macro exits. If you click **No**, the Select a cylinder dialog appears as given in Step 5 above.

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

11 Run the macro again.

12 Select a couple of objects.

13 This time when you come to the **Query** dialog asking you whether to exit the macro, click **No**.

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

15 Click **OK**.

16 The two dialogs appear as described earlier asking for the pitch and the number of turns. Enter values in each and click **OK**

If the helix is larger than the cylinder the following dialog will appear.

If the helix is smaller, the following appears.

If the helix fits the cylinder, no dialog is displayed.

In all cases, the helix is created around the cylinder.

17 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:\dcam\product\powershapeXXXX\file\examples\Macro_Writing

where **XXXX** is the version number of the software and **c** is the drive on which the software is installed.

The changes are given in bold text.

// This macro creates a helix

// Written by: Razia Ghani

// 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 dialogue boxes 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*.

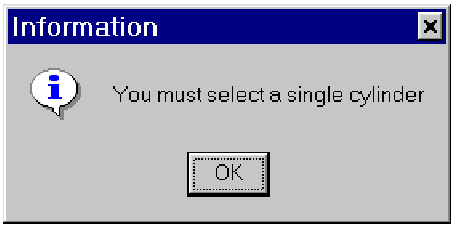
- Once we have tested the selection and we still don't have a single cylinder selected, we want to tell the user that a single cylinder must be selected and ask whether to exit the macro.

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

  IF $no_cyl {

- The following command displays one of the dialogs.

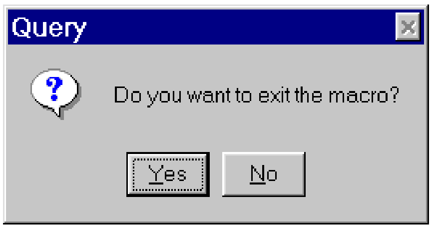
```
PRINT ERROR 'You must select a single cylinder'
```



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'
}
```

Running the macro to test that the tests work

Run the macro to check if the tests work.

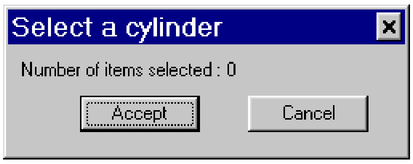
1 Create different objects in your model to test your macro. Make sure you have a primitive cylinder.

2 From the **Main** menu, select **Macro** followed by **Run** to display the **Select A Macro To Run** dialog.

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

4 Click **Open**.

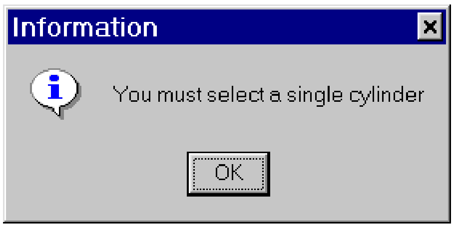
5 The following dialog appears asking for you to select a cylinder.



6 Select a couple of objects.

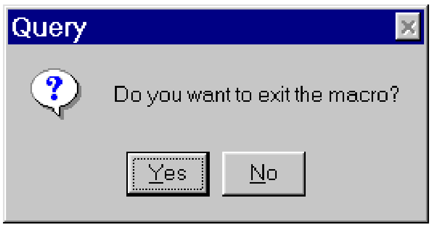
7 Click **OK**

8 The **Information** dialog appears telling you that a single cylinder must be selected.



9 Click **OK**.

10 The **Query** dialog appears asking if you want to exit the macro.



If you click **Yes**, the macro exits. If you click **No**, the **Select a cylinder** dialog is displayed as shown in step 5 above.

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

12 Run the macro again.

13 Select a couple of objects.

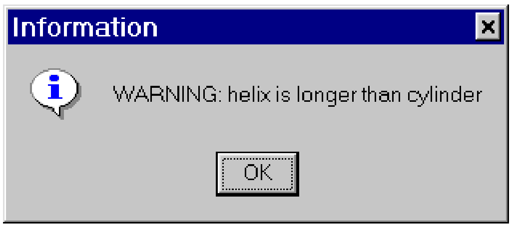
14 This time when you come to the **Query** dialog asking you whether to exit the macro, click **No**.

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

16 Click **Accept**.

17 The two dialogs appear as described earlier asking for the pitch and the number of turns. Input values in each and click **OK**

If the helix is larger than the cylinder the following dialog will appear.



If the helix is smaller, the following appears.

If the helix fits the cylinder, no dialog is displayed.

In all cases, the helix is created around the cylinder.

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

Macros - working examples

Use the following macro examples in your own 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
needs to 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 geometry.

```
// this creates the geometry to be used in the macro
// Two intersecting planes are created and then a curve
is created from the intersection this will be the created
item
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 doesnot 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 $fred
Accept
TEXT FONT Delcam Sans SerIF
TEXT HEIGHT 0.3
TEXT PITCH 0.1
SELECT
select clearlist
//
TOOLS PREFERENCES
UNITPREFS
TEXTPREFS
TEXT LIVETEXT ON
ACCEPT
//
LET com = concatenate('''; ($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

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's the powershape macro that made the frames
(PowerShape5811 or later):
// 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')
  //
  // Render to the file.
  render 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 now have to turn these frames into a movie.
// You could use, for example, Microsoft Movie Maker
(doesn't read .png), ImageMagick, or something ELSE.
```

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 concatentate
  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 aded 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 a 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 suface 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 know directory.

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

```
// to import all files of type x_t from a known directory
// Each file is imported into it's 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 linegth 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'
```

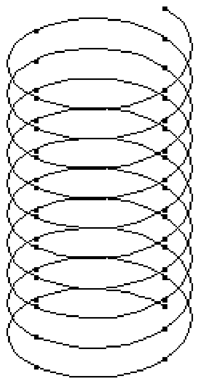
Using WHILE loop to create point at centre of arc

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 HTM 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.

More information on FORM and INPUT commands

- 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

We will open the file in Internet Explorer inside PowerSHAPE to see what the html page looks like.

1. Start up PowerSHAPE.
2. Select **View > Windows > Command** to display 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:

4. 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

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. For further details see Example using Javascript (see page 130)

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>
```

More information on VBscript

- 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.
- Add a procedure (see page 112)
- Link the procedure to the **Apply** button. (see page 116)

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 Macro tutorial (see page 52) before creating your own HTML applications

The following are commands from the macro in the 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 either 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.

More information on the onClick command

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. Press the right mouse button 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.

Exiting the HTML application

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

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.delcam.com'" ><p>

3. Save the file.

More information on adding the Quit button

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

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

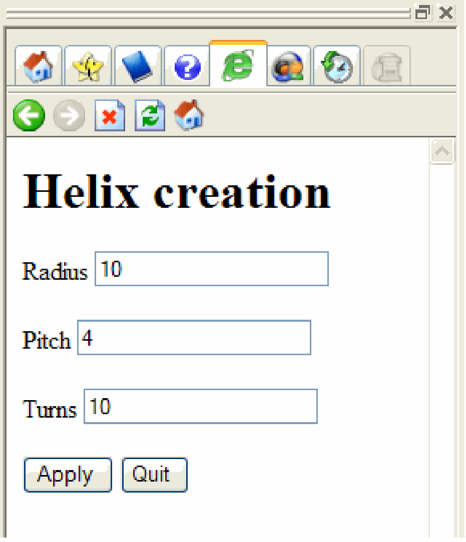
When you click the **Quit** button, the action is defined by the following:

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

This opens the Delcam 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.

Testing the Quit button

1 Press the right mouse button in the Browser window to display the context menu and select **Refresh**.



2 Click the **Quit** button in the Browser window to displays the Delcam home page.

3 To go back to the helix application, press the right mouse button in the Browser window to display the context menu and select **Back**.

More information on adding the Quit button.

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

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

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

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

This opens the Delcam 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.

Entering 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 119)

2 Adding the code (see page 119)

Changing the interface

1 Open the HTML file.

2 Add the following code before the code for the **Apply** button.

```
<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>
```

3 Save the HTML file.

More information on changing the interface

- 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 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 will provide 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.

More information on INPUT POINT command

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 has been used in the application as follows:

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

pshape.Exec sends the command from the vbscript to PowerSHAPE.

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

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

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

We will now 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.

More information on the Apply_Click procedure

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, that 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

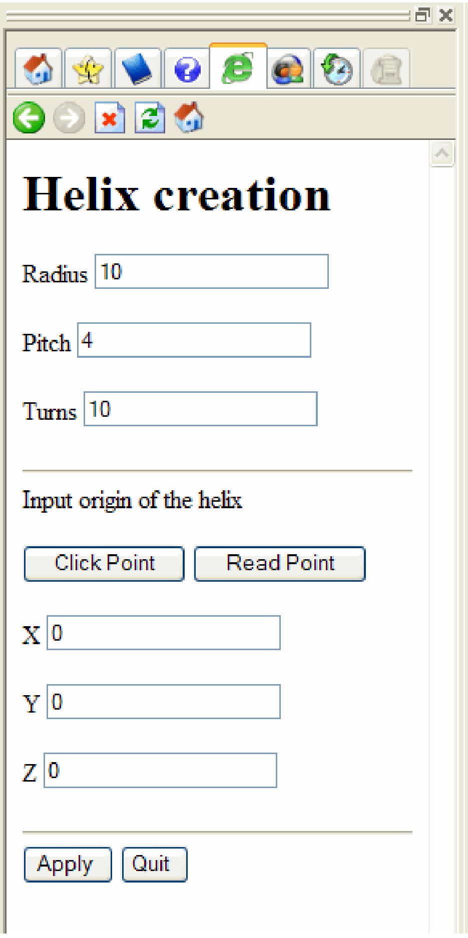
You are now ready to test your application. Complete the following tests:

1 Defining the origin of the helix by entering values for X, Y and Z (see page 122)

2 Defining the origin of the helix using the mouse (see page 122)

Defining the origin of the helix by entering values for X, Y and Z

1. Click the right mouse button 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.

Defining 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 a button to the interface.

1 Go back to the HTML file.

2 Add the following code before the **Apply** button.

```
Create helix around a cylinder<p>

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


<hr>
```

3 Save the HTML file.

Boolean variable called cylinder

In some commands, you will need to know if a cylinder is selected or not. You can use a Boolean variable called *cylinder* to indicate if a cylinder is selected or not. When the program is run, the cylinder variable needs to be set to *false*. Once a cylinder is selected and used in the HTML application, the cylinder variable is set to *true*.

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 will select a cylinder and then click the **Select cylinder** button. The **cyl_select** procedure that this button calls needs to be added.

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.

More information on the cyl_select() procedure

- 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, we use a temporary workplane.

In the **Apply_click** procedure, the commands can be updated to

- Create a temporary workplane (see page 125)
- Input the first point of the helix relative to the temporary workplane (see page 127)
- Delete the temporary workplane (see page 128)

Creating 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.

More information on creating a workplane

- 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

The first coordinate of the helix is going to be different, depending on whether a cylinder is selected or not.

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.

More information on the first point relative to the workplane

If you have selected a cylinder, the helix needs to start at the coordinates in relation to the temporary workplane. Otherwise the coordinates need to be relative to the coordinates in the **X**, **Y** and **Z** boxes.

You are already familiar with the new commands added here

Deleting the workplane

You need to 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.

More information on deleting the temporary workplane

Once the helix is created, the temporary workplane is deleted and the cylinder variable is changed to false. This indicates no cylinder is selected.

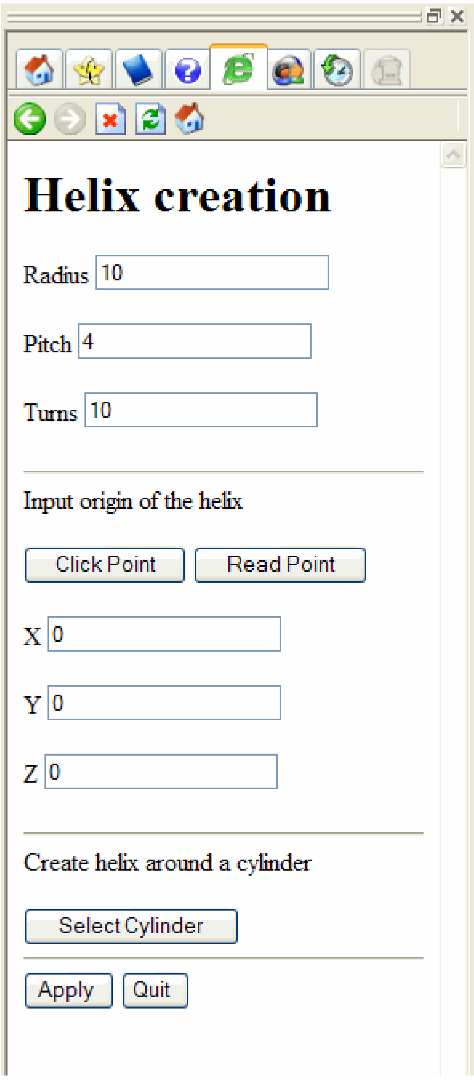
We have used the PowerSHAPE commands from the macro tutorial.

Testing the new code

You are now ready to test your application.

1 Save your HTML file.

2 Click the right mouse button in the Browser window and select **Refresh** from the context menu.

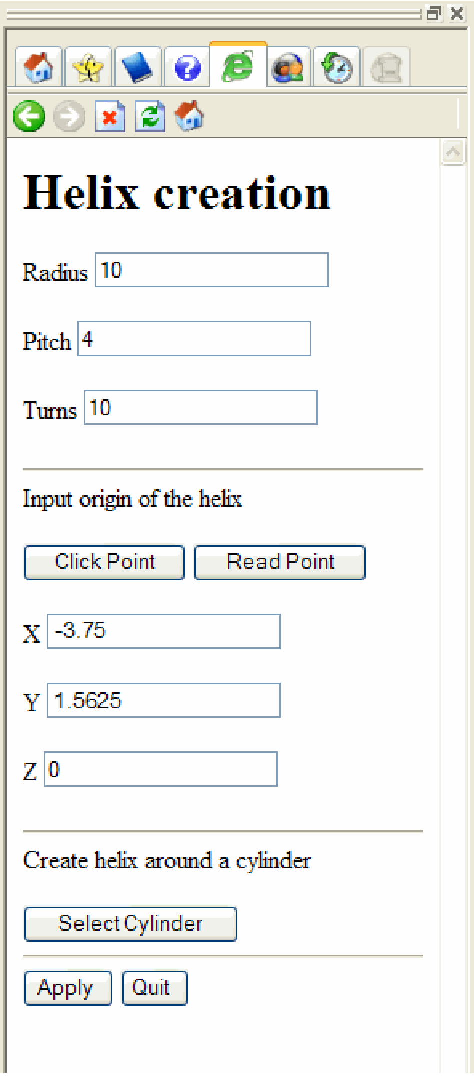


3 Create a cylinder surface. We will use the cylinder to create a helix.

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.

Summary

You have now created an application using HTML. You could further enhance the application by adding,

- tests to check the input data.
- ✓ and ✗ to indicate if a cylinder is selected or not.

Example using Javascript

You can use other script languages instead of vbscript.

The Javascript version of the final code of the helix example is given below.

```
<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.delcam.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
- add-in

These applications allow you to:

- perform commonly used operations
- create easy-to-use interfaces

Both types of applications use the same OLE commands.

What is a HTML-based application?

A HTML-based application is one which is made from 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 (http://www.microsoft.com)

The *HTML application tutorial* introduces you to creating HTML-based applications using vbscripts.

What is an add-in application?

An add-in application is one which runs outside PowerSHAPE, but communicates commands with PowerSHAPE.

If you have purchased third party software such as Visual Basic, you can create applications using that software and add them into PowerSHAPE. Hence the name add-in applications.

You can write add-in applications using programming languages such as Microsoft Visual Basic and Microsoft Visual C++. The OLE commands in the programs allows you to communicate with PowerSHAPE.

Using Visual Basic

In our examples for add-in applications, we use Visual Basic. The creation of add-in applications using Visual Basic requires fundamental knowledge of VB.NET

You can find full details on automating PowerSolution products on our web site: http://www.delcam.com (http://www.delcam.com)

The sections you will find useful are:

Introduction (http://www.delcam.com/vb/DOTNet/Introduction.htm)

Using VBdotNet With PowerSolution (http://www.delcam.com/vb/DOTNet/Using_VBdotNET_With_PowerSolution_Products.pdf)

What are the PowerSHAPE OLE commands?

The OLE commands are the same regardless of the programming language.

The following sections use HTML examples. If you are using VB.NET, you should refer to the relevant section of our web site

http://www.delcam.com/vb/DOTNet/Using_VBdotNET_With_PowerSolution_Products.pdf (http://www.delcam.com/vb/DOTNet/Using_VBdotNET_With_PowerSolution_Products.pdf)

The commands are covered in more details in the following sections:

- Sending commands to PowerSHAPE
- Getting information from PowerSHAPE
- Getting information about a model
- Showing/hiding the PowerSHAPE window

- Controlling the PowerSHAPE window
- Finding the version number of PowerSHAPE
- How do I know if PowerSHAPE is busy?
- Showing/hiding dialogs when executing commands
- Exiting PowerSHAPE using my application
- Selecting objects
- Running run a HTML-based application
- How do I run an add-in application?

We also show you how to input points and select objects using the OLE commands.

Before you can use the PowerSHAPE OLE server, you must connect to an existing PowerSHAPE session. For further details see Connecting to PowerSHAPE (see page 138)

Connecting to PowerSHAPE

You can connect to an existing PowerSHAPE session. 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
```

For add-in applications, use:

```
Set pshape = Getobject(,"PowerSHAPE.Application")
```

Both methods create the object *pshape,* that is connected to an existing PowerSHAPE session.

With these methods, when you quit the applications, the PowerSHAPE session remains open.

HTML example using vbscript

```
<script language="vbscript" >
set pshape = window.external
...
...
...
</script>
```

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 macros (see page 10) command to run in PowerSHAPE.

Example

In this 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 will return a VARIANT variable, which means the result can be a number, a string, or even 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 153)

Getting information about a model

You can use the following method to get information about an open model:

pshape.activedocument

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 becomes associated with the current active model. You can use this object to check if 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.*

Example

While your application is running, the user can have more than one model open. You can restrict the commands in your add-in application to just one model. The active document method allows you to observe a model, you can then check if the model is active.

```
<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 window PowerSHAPE

You can do the following to the PowerSHAPE window:

- minimise

- maximise
- normalised
- bring to foreground

The **WindowState** property sets the state of the PowerSHAPE window.

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.

HTML example using vbscript

The following minimises the PowerSHAPE window to the taskbar, carries out some commands and then maximises 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 following property returns a string containing the version number of PowerSHAPE that your application is currently connected to:

```
pshape.Version
```

If you are not connected to PowerSHAPE, an error is returned.

How do I know if PowerSHAPE is busy?

The following property checks if the connected PowerSHAPE session is busy:

```
pshape.busy
```

If PowerSHAPE is busy, this property will return **True**, otherwise it will return **False**.

PowerSHAPE will be registered as *busy* in the following conditions:

- the **Import** or **Export** dialogs are open
- the **Print** dialog is open

- any PowerSHAPE dialog is displayed

This property is most useful when waiting for a user to input a position in an add-in application. For an example of this, see Add-in example using Visual Basic (see page 144)

Add-in example using Visual Basic

This example will wait for a point input in PowerSHAPE after clicking a button called **cmdIndicate**. It will then extract 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** was not included, the program would not wait until the point is entered. Therefore you would try to extract values that have not yet been set. Try removing this loop to see what happens!

Showing and hiding dialogs when executing commands

To access certain functions, (for example changing the name of an arc), you need to display the **Arc** dialog. When using the OLE server however, you do not normally want to see the dialog; you only want to access the functions within it.

The following commands control the display of the user interface and dialogs:

- Use **ShowForms** property to hide and display the dialogs when sending OLE command.

  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 PowerSHAPE 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 PowerSHAPE 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.

How do I exit PowerSHAPE using my application?

Use this command to exit the PowerSHAPE session you are connected to:

pshape.exit

No confirmation dialog will appear before PowerSHAPE quits.

Entering positions

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.

HTML 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 can't put the commands in the two procedures above into one procedure. If you do, the following will happen when you use the application.

- While the user is clicking the position, the application will automatically go to the next command line without receiving the *$pos* data from PowerSHAPE.
- If you pause the application using the **pshape.busy** property, it will get stuck in an infinite loop.

Selecting objects

We will now show you how to use selected objects in your application.

You can select objects to use in your application in two ways:

- Before it is run.
- As soon as the application is run, you can immediately 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, you simply click the button to say the selection is complete. You can then use the *selection* object information to interrogate the selection and operate on the selection.

For further details on the selection of object information, see Introduction to object information (see page 153)

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 your add-in application communicates to PowerSHAPE using the Windows interpreter. Therefore, running each command results in a very short delay. If you have many PowerSHAPE commands, this delay can last a few seconds.

To minimise the delay, we recommend that where you have a block of PowerSHAPE commands, you use a single execute command. Each line of PowerSHAPE 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 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

Once you have created a HTML-based application, you can run it in the PowerSHAPE browser window.

1 Start PowerSHAPE.

2 Select **View > Window > Command**

3 From 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.

4 Use the HTML-based application.

Running an add-in application

Once you have created or downloaded an add-in application, you can run it either inside or outside PowerSHAPE. When you run your application, it starts executing its commands.

How do I run my add-in application outside PowerSHAPE?

You can run your application in in one of the following ways:

- 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 from the Desktop or the Start menu. For further details see the Microsoft Windows Help documentation supplied with your operating system.

Running your add-in application in PowerSHAPE

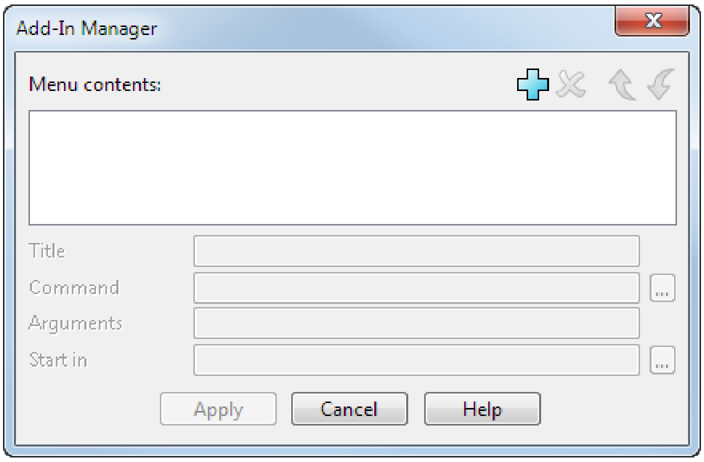
You can use the Add-in Manager in PowerSHAPE to create a link to your application. This lets you to run your application from within PowerSHAPE.

For further information, select from the following:

- Adding an add-in application to PowerSHAPE (see page 150)
- Running an add-in application in PowerSHAPE (see page 151)
- Changing the name of an item in the Add-in menu (see page 152)
- Changing the order of the items in the Add-in menu (see page 152)
- Deleting an item from the Add-in menu (see page 152)

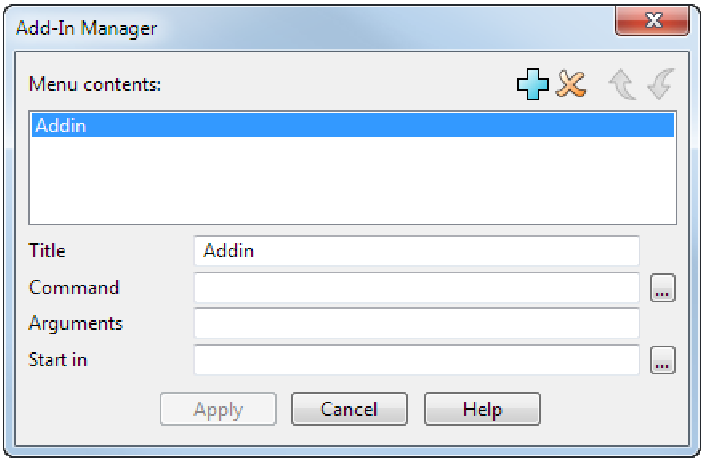
Adding an add-in application to PowerSHAPE

1 From the **Module** menu, select **Add-Ins** followed by **Manager** to display the **Add-In Manager** dialog.



2 Click the **Add** button on the dialog.

This adds a new item called **Add-in** in the list. This item is highlighted, ready for you to change its name.



3 Change the name of the item to something suitable. This name will appear in the **Add-ins** menu from the **Module** menu.

4 In the **Command** box, type the path where your application is stored. You can also click the **Browse** button to display the **Open** dialog, to search for your application .

5 In the **Arguments** box, input any arguments you want your application to use when it starts up.

6 In the **Start in** box, type the default path where you want your application to run.

7 If there are other items in the list and you want to change the position of the new item, use the **Move Item Up** button.

8 Click **Apply**. This adds the item to the **Add-Ins** menu (available from the **Module** menu).

9 Add other applications if you want.

10 Click **Close** to remove the dialog from the screen.

*Add-in applications are only available on the **Add-Ins** menu for the user who added them.*

Running an add-in application in PowerSHAPE

1 From the **Module** menu, select **Add-ins**

2 Select your application.

You must add your application to PowerSHAPE before you can run it in PowerSHAPE. For further details, see Adding an add-in application to PowerSHAPE (see page 150)

Changing the name of an item in the Add-in menu

1. Display the **Add-in Manager** dialog.
2. Select the item.
3. Press the **F2** key.
4. Edit the name.
5. Click **Apply** to change the name.

Changing the order of items in the Add-in menu

1. Display the **Add-in Manager** dialog.
2. Select the item you want to move.
3. Use the **Move Item Up**  and **Move Item Down**  buttons to change the position of the selected item.
4. Click **Apply** to change the order.

Deleting an item from the Add-in menu

1. Display the **Add-in Manager** dialog.
2. Select the item you want to delete.
3. Click the **Delete** button .
4. Click **Apply** to delete the item.

PowerSolutionDOTNetOLE control

This control allows you to use a special set of OLE commands in your application. This set of commands is designed to make programming easier in VB.NET

For details on using the PowerSolutionDOTNetOLE control, see our web site : http://www.delcam.com/vb/DOTNet/UsingTheClassLibrary.html (http://www.delcam.com/vb/DOTNet/UsingTheClassLibrary.html)

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.

Information on the different objects can be found in the following sections:

Introduction to object information

You can access information about PowerSHAPE 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.*

Arc

The following groups of arc commands are available:

Identity number of arc (see page 155)

Start position of arc (see page 155)

End position of arc (see page 155)

Mid position of arc (see page 155)

Radius of arc (see page 156)

Centre position of arc (see page 156)

Length of arc (see page 156)

Centre mark of arc (see page 156)

Angles of arc (see page 156)

Style of arc (see page 157)

Level of arc (see page 157)

Arc exists

arc[*name*].exists
1 if arc exists. *0* otherwise.

Identity number of arc

arc[*name*].id
unique identity number of the arc in the model.

Name of arc

arc[id *n*].name
name of the arc that has the given identity number.

Start position of arc

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.

End position of 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.

Mid position of 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.

Radius of arc

arc[*name*].radius
radius value of the arc.

Centre position of 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.

Length of arc

arc[*name*].length
length of the circumference of the arc.

Centre mark of 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

Angles of arc

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.

Style of 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.

Level of arc

arc[*name*].level
level on which the arc exists.

Application paths

app.paths

Path information for some of directories that PowerSHAPE uses. Output from using this command will look something like this:

Program : C:\Program Files\Delcam\PowerSHAPExxxx\sys\exec\powershape.exe

Document : C:\Program Files\Delcam\PSDocxxxx\help

Pre-config macro : C:\Program Files\Delcam\powershapexxxx/lib/macro/preconfig.mac

Post-config macro : C:\Program Files\Delcam\powershapexxxx/lib/macro/postconfig.mac

Login macro : C:\Program Files\Delcam\powershapexxxx/lib/macro/login.mac

Temp : C:\Documents and Settings\xxx\Local Settings\Temp

Shareddb : C:\Documents and Settings\All Users\Shared Documents\Delcam\shareddb

Parts : C:\Documents and Settings\All Users\Shared Documents\Delcam\parts

Local config : C:\Documents and Settings\xxx\Application Data\PowerSHAPE\

Home : C:\Documents and Settings\xxx\Application Data\

Assembly

Definitions:

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

Checks:

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 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.

Actions:

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

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

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

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

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

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

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

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

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

solid['*ToolSolid*'].hide

1 if the solid is owned by another item and not displayed. *0* if the solid is hidden.

Clipboard

clipboard.valid
1 if there is something on the clipboard *and 0* otherwise.

Composite curve

Commands for composite curves can take either of the following forms:

compcurve[*name*]……

composite curve[*name*]…..

To avoid duplication, the format **compcurve[*name*]** is used throughout.

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.

Points in composite curve

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.

Tangent direction at a 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 which defines the tangent direction entering the point.

compcurve[*name*].point[*number*].entry_tangent.y
y value of the unit vector which defines the tangent direction entering the point.

compcurve[*name*].point[*number*].entry_tangent.z
z value of the unit vector which 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 which defines the tangent direction leaving the point.

compcurve[*name*].point[*number*].exit_tangent.y
y value of the unit vector which defines the tangent direction leaving the point.

compcurve[*name*].point[*number*].exit_tangent.z
z value of the unit vector which defines the tangent direction leaving the point.

Azimuth and elevation angles at a 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.

compcurve[*name*].point[*number*].exit_tangent.elevation
elevation angle of the tangent leaving the point.

Magnitude at a point

compcurve[*name*].point[*number*].entry_magnitude
magnitude entering the point.

compcurve[*name*].point[*number*].exit_magnitude
magnitude leaving the point.

Items in composite curve

compcurve[*name*].item.number
number of items that make up the composite curve.

Length of 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.

Area of composite curve

compcurve[*name*].area
area of the composite curve.

If the composite curve is closed and planar, the area is the enclosed area.

If the composite curve is open, it is closed with a straight line for the area measurement.

If the composite curve is 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.

Bounding box around composite curve

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.

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

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.

Filleting a composite curve

compcurve[*name*]

will fillet the composite curve, where *name* is the name of the composite curve.

Style of composite curve

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.

Level of composite curve

compcurve[*name*].level
level on which the composite curve exists.

Created

You can 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.

Created objects exist (see page 168)

Number of items created (see page 168)

Identity of item created

Clearlist (see page 168)

Interrogating created items (see page 168)

Created objects exist

created.exists
1 if at least one item is in the creation list. *0* otherwise.

Number of items created

created.number
number of items in the creation list.

Clearlist

created.clearlist
clears the creation list.

Interrogating created items

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 191) .

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)*.

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 created.type[0] is a composite curve, then 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

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.

Type of curve

curve[*name*].type
checks the curve and returns one of the following strings:

Bezier

Bspline

Number of points in curve

curve[*name*].number
number of points in the curve.

Closed curve

curve[*name*].closed
1 if the curve is closed. *0* otherwise.

Start and end positions of curve

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.

Points in a curve

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.

Tangent direction at a curve point

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.

Selected points

The following variables have been added in PowerSHAPE 2015 R2:

`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).

Azimuth and elevation angles at a curve point

curve[*name*].point[*number*].entry_tangent.azimuth
azimuth angle of the tangent entering the point.

curve[*name*].point[*number*].entry_tangent.elevation
elevation angle of the tangent entering the point.

curve[*name*].point[*number*].exit_tangent.azimuth
azimuth angle of the tangent leaving the point.

curve[*name*].point[*number*].exit_tangent.elevation
elevation angle of the tangent leaving the point.

Magnitude at a curve point

curve[*name*].point[*number*].entry_magnitude
magnitude entering the curve's point.

curve[*name*].point[*number*].exit_magnitude
magnitude leaving the curve's point.

Length of curve

curve[*name*].length
length of the curve.

curve[*name*].length_between(*a; b*)
length along the curve between key points a and b.

Area of curve

curve[*name*].area
area of the curve.

If the curve is closed and planar, the area is the enclosed area.

If the curve is open, it is closed with a straight line for the area measurement.

If the curve is non-planar, the curve is projected onto the current principal plane and the area is measured from the projected curve.

Bounding box around 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.

Centre of gravity of 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.

Style of 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.

Level of curve

curve[*name*].level
level on which the curve exists.

Dimension

The following groups of dimension command are available:

Dimension exists

dimension[*name*].exists
1 if dimension exists. *0* otherwise.

Identity number of dimension

dimension[*name*].id
unique identity number of the dimension in the model.

Name of dimension

dimension[id *n*].name
name of the dimension that has the given identity number.

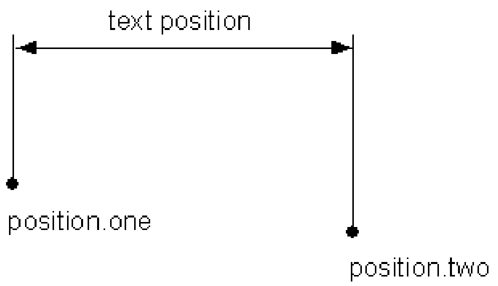
Dimension value

dimension[*name*].value
value of dimension.

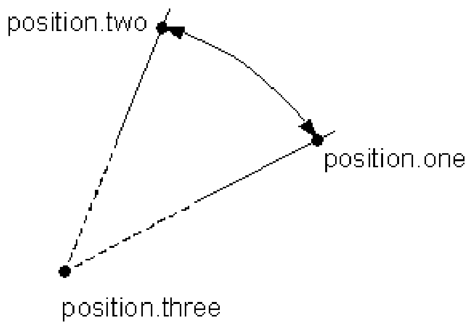
Position of the dimension

A dimension is defined by its text position and various other positions, depending on the type of dimension. 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*.

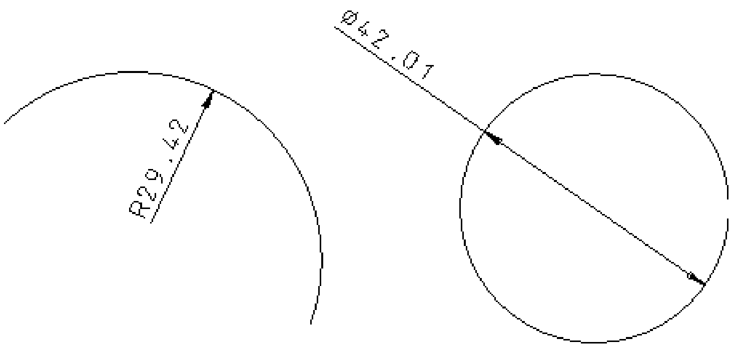
A linear dimension is defined as shown below. It has a text position (*position.text*), *position.one* and *position.two*



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*.



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.

Diameter of dimension

dimension[*name*].diameter
1 if the dimension measures a diameter. *0* otherwise.

Leader of dimension

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 leaders on small dimensions** is on. *0* otherwise.

When you have a dimension with leaders placed on either side of the dimension, the **Internal leaders 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*

Annotation of dimension

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.

Witness of dimension

dimension[*name*].witness.style
style name of the witness line of the dimension.

Tolerance of 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 $\begin{matrix}75.15\\74.95\end{matrix}$

dimension[*name*].tolerance.decimal
number of decimal places of the tolerance.

Style of dimension

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.

Level of dimension

dimension[*name*].level
level on which the dimension exists.

Drawing

The following groups of drawing commands are available:

Drawing exists (see page 179)

Drawing description (see page 179)

Identity number of drawing (see page 179)

Name of drawing (see page 179)

Drawing dimensions (see page 179)

Drawing templates used (see page 179)

Number of views (see page 180)

Number of objects (see page 180)

Updating (see page 180)

Drawing exists

drawing[*name*].exists
1 if drawing exists. *0* otherwise.

Drawing description

drawing[*name*].description
description of the drawing.

Number of drawings

drawing.number
the number of drawings

Identity number of drawing

drawing[*name*].id
unique identity number of the drawing.

Name of 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 dimensions

drawing[*name*].width
width of the drawing.

drawing[*name*].height
height of the drawing.

Drawing templates used

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.

Number of views

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

Number of objects

drawing[*name*].no_of_items
number of objects on the drawing.

Updating

drawing[name].view[view_name].needs updating

1 if the view needs updating, *0* otherwise

Drawing view

Drawing view commands are only available in conjunction with Drawing commands as indicated in the following sections:

Extent of drawing view (see page 180)

Scale of drawing view (see page 181)

Origin of drawing view (see page 181)

Number of objects (see page 181)

Transform of drawing view (see page 181)

Converting between drawing view and world space (see page 181)

Extent of drawing view

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.

Scale of drawing view

drawing[*name*].view[*name*].scale
scale of the view.

Origin of drawing view

drawing[*name*].view[*name*].origin
coordinates [x, y, z] of the origin of the view.

Number of objects

drawing[*name*].view[*name*].no_of_items
number of objects in the view.

Transform of drawing 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 defines the elements of the first row of the rotation matrix.
- 4, 5, 6 defines the elements of the second row of the rotation matrix.
- 7, 8, 9 defines the elements of the third row of the rotation matrix.
- 12, 13, 14 defines elements of the translation vector.

Converting between drawing, view and world space

Use the following variables to convert between drawing, view and world space:

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]

You can also use X/Y/Z modifiers with these variables:

DRAWING[drawing_name].VIEW[view_name].DRAWING_TO_VIEW[x ; y ; z].X

returns the x-ordinate of the converted point.

Electrode

The following groups of line commands are available:

General (see page 182)

List (see page 184)

Datum (see page 185)

Blank (see page 185)

Holder (see page 185)

Burn region (see page 186)

Quantity (see page 186)

Undersize (see page 186)

Frames (see page 187)

In some of the 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.*

Use the following to find out if the first electrode exists:

`electrode[index 1].exists`

General (Electrode)

`electrode.number`
number of electrodes in the model.

`electrode[name].exists`
1 if the electrode exists and *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 (see page 208)

`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*`].list`
a list 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.

List

`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.

Datum

`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.

Blank

`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.

Holder

`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|machining|edm>.items
electrode[*name*].holder.<base|machining|edm>.item.number
number of items that make up 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).

Burn region

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.

Quantity

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

Undersize

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)

Frames

Use the following macro variables for electrode frames:

`electrode[...].frame.exists`
returns *1* if the electrode has a frame, *0* otherwise

`electrode[...].frame.length`
returns the length of electrode frame.

`electrode[...].frame.width`
returns the width of electrode frame.

`electrode[...].frame.height`
returns the height of electrode frame.

`electrode[...].frame.has_chamfer`
returns *1* if the electrode frame has a chamfer, *0* otherwise.

`electrode[...].frame.chamfer_size`
returns the size of chamfer on the electrode frame.

Evaluation

evaluation
1 if evaluation copy of software is being used. *0* otherwise.

File

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 and 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

The following groups of hatch commands are available:

Hatch exists

hatch[*name*].exists
1 if drawing exists. *0* otherwise.

Identity number of hatch

hatch[*name*].id
unique identity number of the hatch in the model.

Name of hatch

hatch[id *n*].name
name of the hatch that has the given identity number.

Crossed hatch

hatch[*name*].cross
1 if hatch is crossed. *0* otherwise.

Filled hatch

hatch[*name*].fill
1 if hatch is filled. *0* otherwise.

Hatch angle

hatch[*name*].angle
first angle of hatch.

hatch[*name*].angle1
first angle of hatch.

hatch[*name*].angle2
second angle of hatch.

Hatch spacing

hatch[*name*].spacing
spacing of hatch.

Hatch boundaries

hatch[*name*].boundaries
number of boundaries enclosing the hatch.

Style of 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.

Level of hatch

hatch[*name*].level
level on which the hatch exists.

Lateral

Lateral commands are only available in conjunction with a surface command. For information see Laterals and longitudinals (see page 230).

Level

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

The following groups of line commands are available:

Start coordinates of a line (see page 191)

End coordinates of a line (see page 192)

Line exists (see page 192)

Identity number of line (see page 192)

Name of line (see page 192)

Length of line (see page 192)

Style of line (see page 192)

Level of line (see page 193)

Start coordinates of a line

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.

End coordinates of a 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 exists

line[*name*].exists
1 if line exists. *0* otherwise.

Identity number of line

line[*name*].id
unique identity number of the line in the model.

Name of line

line[id *n*].name
name of the line that has the given identity number.

Length of line

line[*name*].length
length of the line.

Style of 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.weight
weight of line style used to draw the line.

line[*name*].style.width
width of line style used to draw the line.

Level of line

line[*name*].level
level on which the line exists.

Angles of a line

Use the following variables for finding the apparent and elevation angles of a line (these are the same values as shown on the line editing form).

The commands return a *REAL* value, with the angle in the current units - degrees or radians).

`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.

You can optionally 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
```

Longitudinal

Longitudinal commands are only available in conjunction with a surface command. For information see Laterals and longitudinals (see page 230)

Model

The following groups of model commands are available:

Selected model

model.selected
name of the selected model.

model[*name*].selected
1 if the named model is selected. *0* otherwise.

Model exists

model[*name*].exists
1 if the named model exists. *0* otherwise.

Identity number of model

model[*name*].id
unique identity number of the model.

Name of model

model[id *n*].name
name of the model that has the given identity number.

Model open

model[*name*].open
1 if the named model is open. *0* otherwise.

Number of objects in model

The following give the number of objects in the selected model.

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

Model file size

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 that the model must be open. If the model is closed, **model[*name*].filesize** is assigned -1.

The collective size of the model's files in its directory will be slightly larger by about 500bytes. The size can be even larger if untruncated files exist. The command ***Tools - Compress Model*** *can sort out untruncated files as well as reducing the actual database size too.*

Access rights

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

model.path
pathname of the currently selected model.

model[*name*].path
pathname of the named model

For example, **model[mouse].path** returns the pathname D:/dcam/parts/m142.

Locked

model.locked
1 if the currently selected model is locked. *0* otherwise.

model[*name*].locked
1 if the named model is locked. *0* otherwise.

Changed

model.changed
1 if the currently selected model has changed. *0* otherwise.

model[*name*].changed
1 if the named model has changed. *0* otherwise.

Corrupted

model.corrupt
1 if the currently selected model is corrupted. *0* otherwise.

model[*name*].corrupt
1 if the named model is corrupted. *0* otherwise.

File Doctor

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.

Version

model.version
current model version

model.previous_version
version of model prior to upgrade when the model was opened

Updated

model.upgraded
1 if the model was upgraded on opening, *0* otherwise.

Parameter

parameter[*name*].value
value of parameter.

parameter[*name*].exists
1 if parameter exists and 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

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.

Start coordinates of a pcurve

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 and 0 otherwise.

End coordinates of a pcurve

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 and 0 otherwise.

Coordinates of a point on a pcurve

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.

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 and 0 otherwise.

Point

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.

Position of point

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.

Style of 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.

Level of point

point[*name*].level
level on which the point exists.

Printer

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 set and 0 otherwise.

printer[*name*].image_string
image command for this printer.

printer[*name*].plot_string_set
1 if the plot command set. *0* otherwise.

printer[*name*].plot_string
plot command for this printer.

printer[*name*].initialised
1 if the printer is initialise and 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

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

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 string such as { '1'; '1'; 'fred' };one string per selected item

Other selection options

Interrogating selected items (see page 202)

Selection positions (see page 203)

Bounding box around individual objects (see page 206)

Number of selected surface curves/surface curve points (see page 207)

Interrogating selected items

selection.object[*number*]
object type and its name in the selection. For example, *Line[4]*, *Arc[1]*.

If there are *n* items selected, then *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* you can use 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 191) .

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 there are *n* objects selected, then *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, if you want to check that selection.type[0] is a composite curve, then you must use:

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

and not:

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

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

selection.name[*number*]
name of an item in the selection.

If there are *n* items selected, then *number* is the item's number in the selection.

In all cases, number *is from 0 to (n-1).*

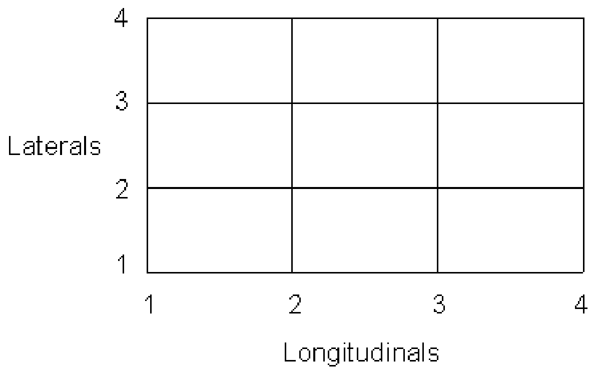
Selection positions

Currently, the selection position is only calculated if there is only one object in the selection. Therefore, the number in brackets [ ] is always zero.

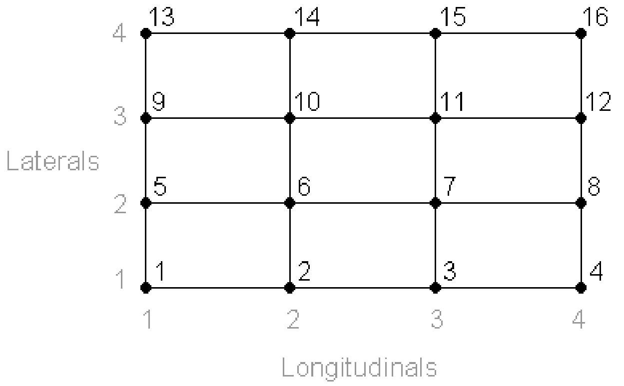
selection.key_point[0]
the number of the selected keypoint in a surface or curve.

For a curve, if the keypoint is the nth point, then **selection.key_point[0]** is *n*.

For a surface, we will use the following surface to describe how the numbers are worked out.



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 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*.

Bounding box around selection

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.

Bounding box around individual objects

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.

Number of selected surface curves/surface curve points

`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`).

Shareddb

shareddb.path
pathname of the shared database that is being used, for example, c:/dcam/shareddb.

Sketcher

sketch
1 if Sketcher is on. *0* otherwise.

Solid

The following groups of solid commands are available:

Solid name

`SOLID[<solid-name>].CLOSEST_FACE(<x>; <y>; <z>`
returns a string representing the name of the closest face of a solid to a given point. The point is entered in current units and absolute coordinates.

Solid exists

solid[*name*].exists
1 if the solid exists. *0* otherwise.

Owner

The following macro variables determine the owner of an entity.

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

solid_active
retrieves the id of the active solid

solid.active
returns the name of the currently active solid.

Identity number of solid

solid[*name*].id
unique identity number of the solid in the model.

Name of solid

solid[id *n*].name
name of solid that has the given identity number.

Solid version

solid[N].parasolid
returns *1* if the solid is a parasolid, else *0*.

solid[N].v8
returns *1* if the solid is a version 8 solid, else *0*.

Active

solid[*name*].active
1 if the solid is active. *0* otherwise.

Ghost

solid[*name*].ghost
1 for a ghost solid. *0* for a normal solid.

Type

solid[*name*].type
checks the solid and retrieves one of the following strings:

Plane

Block

Sphere

Cylinder

Cone

Torus

Extrusion

GeneralSolid

Revolution

ShoeLast

Surfaces in a solid

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]**.

Bounding box around solid

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.

Origin of primitive and extruded solids only

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.

Dimensions of primitive and extruded solids only

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

Workplane of primitive (solid)

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

SOLID[<name>].YAXIS

SOLID[<name>].ZAXIS

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>].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

Examples

PRINT SOLID[1].XAXIS

PRINT SOLID[1].XAXIS.Z

Surface area

solid[*name*].area
surface area of the solid.

Volume of solid

solid[*name*].volume
volume of the solid.

Watertight

solid[*name*].watertight
1 if the solid is watertight within tolerance. *0* otherwise.

Closure

solid[*name*].closed
1 if the solid is closed. *0* otherwise.

Centre of gravity

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.

Moment of inertia

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.

Linked edges

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.

Valid boundaries

solid[*name*].trimming_valid
1 if boundaries in the solid are valid. *0* otherwise.

Connected

solid[*name*].connected
1 if the surfaces which define the solid connect together within tolerance. *0* otherwise.

Features

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*].suppressed
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

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

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:

length - Length of the pocket

width - Width of the pocket

height - Height of the protrusion. This will return the same value as **depth**

depth - Depth of the pocket. This will return 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 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

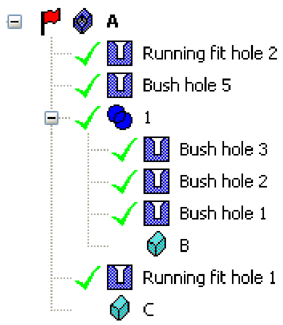
solid[*name*].children
solid[*name*].features
number of features on the solid.

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.

In the example below, **print feature['1'].children** will return the value *4*.



feature[*name*].features.all
number of features, including all sub-branches.

Feature selected

feature[*name*].selected
1 for a selected feature. *0* otherwise.

Feature suppressed

feature[*name*].suppressed
1 for suppressed feature . *0* otherwise.

Feature error

feature[*name*].error
1 for an error state for a feature. *0* otherwise.

Feature exists

feature[*name*].exists
1 if the solid feature exists. *0* otherwise.

Identity number of feature

feature[*name*].id
unique identity number of the solid feature in the model.

Workplane of feature

The following return the X, Y or Z unit axis vector of the feature's workplane. The vector is defined in relation to the currently active workplane:

FEATURE[<name>].XAXIS

FEATURE[<name>].YAXIS

FEATURE[<name>].ZAXIS

The following return the X, Y or Z entity of the unit axis vector of the feature's workplane. The vector is defined in relation 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

Examples

PRINT FEATURE[1].XAXIS

PRINT FEATURE[1].XAXIS.Y

Name of solid

feature[id *n*].name
name of solid feature that has the given identity number.

Type

feature[*name*].type
checks the solid feature and retrieves a string indicating the type of feature.

Number of surfaces

feature[*name*].surfaces
number of visible surfaces that make up the feature.

Name of surface

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.

Length of feature

feature[*name*].length
length of the feature - applies to cut and boss features only.

Angle of feature

feature[*name*].angle
angle of the feature - applies to cut, boss and bulge features only.

Radius of feature

feature[*name*].radius
radius of feature - applies to fillet feature only.

Pre-machined status

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.

Existed at birth flag

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.

Scaling constraints (features)

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.

Material

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.

Style of 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.

Level of solid

solid[*name*].level
level on which the solid exists.

Scaling Constraints (solids)

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.

Picking faces of a solid

When in face selection mode, you can use commands to pick the faces of a selected solid.

- Use the following commands to replace the currently selected faces with named faces:

```
PICK FACE NAME <face_name>
PICK FACE <face_name>
PICK FACE REPLACE NAME <face_name>
PICK FACE NAME <face_name>
```

This is the same as using the mouse to select the faces.

- Use the following commands to add the named face to the current selection:

```
PICK FACE ADD NAME <face_name>
PICK FACE ADD <face_name>
```

This is the same as holding down the SHIFT key and clicking the left mouse button.

- Use the following commands toggle the named face into/out of the current selection:

```
PICK FACE TOGGLE NAME <face_name>
PICK FACE TOGGLE <face_name>
```

This is the same as holding down the CTRL key and clicking the left mouse button.

<face_name> can be a word, string, integer or variable.. The following are all 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 will be selected, spreading out from the named face.
- **Solid Draft Face**
- **Solid Replace Face**
- **Solid Divide Face**

Spine

Spine commands are only available in conjunction with a surface command. For details see Spines (see page 239).

Surface

The following groups of surface commands are available:

General (see page 224)	
Reference direction (see page 224)	Primitives (see page 224)
Trimmed surface (see page 227)	Minimum block size (see page 227)
Surface type (see page 228)	Area of surface (see page 228)
Diameter of surface (see page 229)	Volume of surface (see page 229)
Centre of gravity of surface (see page 229)	Evaluate position (see page 229)
Evaluate normal (see page 229)	Evaluate curvature (see page 230)
Nearest t and u parameters (see page 230)	Laterals and longitudinals (see page 230)
Owner (see page 239)	Material (see page 239)

Spines (see page 239)

Trim regions (see page 242)

Boundaries (see page 242)

Pcurves (see page 242)

Style of surface (see page 242)

Level of surface (see page 243)

The following commands can also be used:

Number of selected surface curves/surface curve points (see page 207)

General surface commands

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 will be tangent to the surface. The first 3 coordinates are a point outside the surface and the last 3 are the initial guess point on the surface.

Reference direction

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.

Primitives

Surface syntax in this section applies to primitive surfaces (including extrusions). It outputs data about the surface's dimensions and workplane instrumentation.

Dimensions of surface (see page 225)
Origin of surface (see page 225)
Axes directions of primitive (see page 226)
Workplane of primitive (see page 227)

Dimensions of surface

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.

Origin of surface

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.

Axes directions of primitive

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.

Workplane of primitive (surface)

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:

SURFACE[<name>].XAXIS

SURFACE[<name>].YAXIS

SURFACE[<name>].ZAXIS

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:

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

Examples

PRINT SURFACE[1].YAXIS

PRINT SURFACE[1].YAXIS.Z

Trimmed surface

surface[*name*].trimmed
1 if the surface's local trim flag is set. *0* otherwise.

Minimum block size

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 type

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

Area of surface

surface[*name*].area
area of the surface.

Diameter of surface

surface[*name*].diameter
diameter of surface.

Volume of surface

surface[*name*].volume
volume of the surface.

Centre of gravity of 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.

Evaluate position

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.

Evaluate normal

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.

Evaluate curvature

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.

Nearest 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 complicated surfaces, you can supply guessed t and u values close to the coordinates to speed up the calculations. The guessed values are added in the 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].

Laterals and longitudinals

Click one of the following:

Closed laterals and longitudinals (see page 231)

Number of laterals and longitudinals (see page 231)

Number of selected surface curves/surface curve points (see page 207)

Laterals

Longitudinals

Closed laterals and longitudinals

surface[*name*].lat_closed
1 if the surface's laterals are closed. *0* if open.

surface[*name*].lon_closed
1 if the surface's longitudinals are closed. *0* if open.

Number of laterals and longitudinals

surface[*name*].nlats
number of laterals in the surface.

surface[*name*].nlons
number of longitudinals in the surface.

Start and end positions of lateral

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.

Number of points in lateral

surface[*name*]**.lateral**[*number*]**.number**
number of points in the lateral.

Length of 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.

Lateral exists

surface[*name***].lateral[***number***].exists**
1 if lateral exists and 0 otherwise.

Identity number of lateral

surface[*name***].lateral[***number***].id**
unique identity number of the lateral.

Name of lateral

surface[*name***].lateral[***number***].name**
name of the lateral.

Lateral points

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.

Tangent magnitude at lateral points

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.

Tangent direction at lateral points

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.

Azimuth and elevation angles at lateral points

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.

Normal at lateral points

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.

Centre of gravity at lateral

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.

Start and end positions of longitudinal

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.

Number of points in longitudinal

surface[*name*]**.longitudinal**[*number*]**.number**
number of points in the longitudinal.

Length of 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.

Longitudinal exists

surface[*name***].longitudinal[***number***].exists**
1 if longitudinal exists. *0* otherwise.

Identity number of longitudinal

surface[*name***].longitudinal[***number***].id**
unique identity number of the longitudinal.

Name of longitudinal

surface[*name***].longitudinal[***number***].name**
name of the longitudinal.

Longitudinal points

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 tangent vector at any (T,U) value

There are variables to calculate the surface tangent vector at any (T, U) value.

- Tangent vector U, direction before/after (around lateral) of the specified (T,U) point on the surface
  `surface[entity_name].evaluate(T; U).udirb`
  `surface[entity_name].evaluate(T; U).udira`
- Tangent vector T, direction before/after (along longitudinal) of the specified (T,U) point on the surface
  `surface[entity_name].evaluate(T; U).tdirb`
  `surface[entity_name].evaluate(T; U).tdira`
- Coordinates 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`
- Normal direction of the specified (T,U) point on the surface
  `surface[entity_name].evaluate(T; U).normal`
- Draft angle of the surface at specified (T,U) point
  `surface[entity_name].evaluate(T; U).draft_angle`
- Minimum curvature of the surface at specified (T,U) point
  `surface[entity_name].evaluate(T; U).curvature.min`
- Maximum curvature of the surface at specified (T,U) point
  `surface[entity_name].evaluate(T; U).cuvature.max`

Tangent magnitude at longitudinal points

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.

Tangent direction at longitudinal points

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.

Azimuth and elevation angles at longitudinal points

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.elevation
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.

Normal at longitudinal points

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
x value of the unit vector of the normal leaving the longitudinal's point.

Centre of gravity at longitudinal

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.

Flare and twist

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.

Owner

The following macro variables determine the owner of an entity.

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.

Material

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.

Spines

Click one of the following:

Spine exists

surface[*name*].spine.exists
1 if the spine exists. *0* otherwise.

Identity number of spine

surface[*name*].spine.id
unique identity number of the spine.

Name of spine

surface[id *n*].spine.name
name of the spine that has the given identity number.

Number of spine points

surface[*name*].spine.number
number of spine points.

Length of spine

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.

Start position of spine

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.

End position of 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.

Position of the spine points

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.

Tangent direction at a 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.

Azimuth and elevation angles at spine points

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.

Trim regions

surface[*name*]**.trimming_valid**
1 if the trim boundaries on the surface form a valid trim region. *0* otherwise.

Boundaries

surface[*name*]**.boundaries**
number of boundaries on the surface.

Pcurves

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.

Style of surface

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.

Level of surface

surface[_name_**].level**
level on which the surface exists.

Symbol

The following groups of symbol commands are available:

Symbol exists

symbol[_name_**].exists**
1 if the symbol exists. *0* otherwise.

Identity number of symbol

symbol[_name_**].id**
unique identity number of the symbol in the model.

Name of symbol

symbol[id _n_**].name**
name of the symbol that has the given identity number.

Pins

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.

Style of 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.

Level of symbol

symbol[*name*].level
level on which the symbol exists.

Area and volume of symbols

symbol[*name*].area
area of triangulated symbols.

symbol[*name*].volume
volume of triangulated symbols.

Scaling Constraints - 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 Definition

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

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.

Unstripped text

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

Stripped text

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.

Text editor

text[*name*].livetext
1 if text created using PowerSHAPE standard text editor. *0* for DUCT editor.

Colour of text

text[*name*].colour
number of the colour used by the text.

Level of text

text[*name*].level
level on which the text exists.

Tolerance

tolerance.general
value of general tolerance.

tolerance.drawing
value of drawing tolerance.

Units

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

You can use the commands in this group to query which objects were updated as a result of the last operation. These objects are accessed from the updated list.

Updated objects exist (see page 248)

Clear the updated list (see page 248)

Number of items updated (see page 248)

Interrogating updated items (see page 248)

Updated objects exist

updated.exists
1 if at least one item is in the updated list. *0* otherwise.

Clear the updated list

Updated.clearlist
Objects are removed from the updated list**.**

Number of items updated

updated.number
number of items in the updated list.

Interrogating updated items

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 191). 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.

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 ***updated.type[0]*** *is a composite curve, then you must use:*

updated.type[0] == 'Composite Curve'

and 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

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.

Version

version
version of PowerSHAPE that is being used, for example, *7240*

version.major
first digit of the version of PowerSHAPE being used. For example, if you are using 7240, **version.major** would return *7*.

version.minor
second digit of the version of PowerSHAPE being used. For example, if you are using 7240, **version.minor** would return *2*.

version.revision
last two digits of the version of PowerSHAPE being used. For example, if you are using 7240, **version.revision** would return *40*.

version.has.excel
tests if MS Excel is installed.

View

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

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, drawing, or render.

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

*If you don't specify the name of the workplane, the active one is used, for example, **workplane.origin** returns the origin of the active workplane. An error is given if there is no active workplane.*

The following groups of workplane commands are available:

Active

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.

Axes directions

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 exists

workplane[*name*].exists
1 if the workplane exists. *0* otherwise.

Identity number of workplane

workplane[*name*].id
unique identity number of the workplane in the model.

Name of workplane

workplane[id *n*].name
name of the workplane that has the given identity number.

Level of workplane

workplane[*name*].level
level on which the workplane exists.

Locked

workplane[*name*].locked
1 if the workplane is locked. *0* otherwise.

Origin of workplane

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.

Style of 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.