
Autodesk® PowerMill® 2017

Macro Programming Guide



Autodesk® PowerMill® 2017

© 2016 Delcam Limited. All Rights Reserved. Except where otherwise permitted by Delcam Limited, this publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose.

Certain materials included in this publication are reprinted with the permission of the copyright holder.

Trademarks

The following are registered trademarks or trademarks of Autodesk, Inc., and/or its subsidiaries and/or affiliates in the USA and other countries: 123D, 3ds Max, Alias, ArtCAM, ATC, AutoCAD LT, AutoCAD, Autodesk, the Autodesk logo, Autodesk 123D, Autodesk Homestyler, Autodesk Inventor, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSketch, AutoSnap, AutoTrack, Backburner, Backdraft, Beast, BIM 360, Burn, Buzzsaw, CADmep, CAiCE, CAMduct, Civil 3D, Combustion, Communication Specification, Configurator 360, Constructware, Content Explorer, Creative Bridge, Dancing Baby (image), DesignCenter, DesignKids, DesignStudio, Discreet, DWF, DWG, DWG (design/logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DWGX, DXF, Ecotect, Ember, ESTmep, FABmep, Face Robot, FBX, FeatureCAM, Fempro, Fire, Flame, Flare, Flint, ForceEffect, FormIt 360, Freewheel, Fusion 360, Glue, Green Building Studio, Heidi, Homestyler, HumanIK, i-drop, ImageModeler, Incinerator, Inferno, InfraWorks, Instructables, Instructables (stylized robot design/logo), Inventor, Inventor HSM, Inventor LT, Lustre, Maya, Maya LT, MIMI, Mockup 360, Moldflow Plastics Advisers, Moldflow Plastics Insight, Moldflow, Moondust, MotionBuilder, Movimento, MPA (design/logo), MPA, MPI (design/logo), MPX (design/logo), MPX, Mudbox, Navisworks, ObjectARX, ObjectDBX, Opticore, P9, PartMaker, Pier 9, Pixlr, Pixlr-o-matic, PowerInspect, PowerMill, PowerShape, Productstream, Publisher 360, RasterDWG, RealDWG, ReCap, ReCap 360, Remote, Revit LT, Revit, RiverCAD, Robot, Scaleform, Showcase, Showcase 360, SketchBook, Smoke, Socialcam, Softimage, Spark & Design, Spark Logo, Sparks, SteeringWheels, Stitcher, Stone, StormNET, TinkerBox, Tinkercad, Tinkerplay, ToolClip, Topobase, Toxik, TrustedDWG, T-Splines, ViewCube, Visual LISP, Visual, VRED, Wire, Wiretap, WiretapCentral, XSI

All other brand names, product names or trademarks belong to their respective holders.

Disclaimer

THIS PUBLICATION AND THE INFORMATION CONTAINED HEREIN IS MADE AVAILABLE BY AUTODESK, INC. "AS IS." AUTODESK, INC. DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

Contents

Macros	1
Creating macros	1
Recording macros in PowerMill	2
Running macros	3
Editing macros.....	3
Running macros from within macros	4
Writing your own macros	5
PowerMill commands for macros	6
Adding comments to macros	7
Macro User Guide	8
Variables in macros.....	27
Using expressions in macros.....	47
Operator precedence	48
Executing a macro string variable as a command using DOCOMMAND ..	50
Macro functions	51
IF statement	55
IF - ELSE statement	56
IF - ELSEIF - ELSE statement.....	57
SWITCH statement	58
Running macros without displaying GUI items	61
BREAK statement in a SWITCH statement	61
Repeating commands in macros	62
Creating sequences	66
RETURN statement.....	67
Printing the value of an expression	68
Constants	69
Built-in functions	69
Entity based functions	103
Model hierarchy	107
Model Component Functions.....	107
Model Hierarchies	108
Nodes.....	108
Walking the hierarchy	109
Getting a Node by its Path.....	109
Getting the Hierarchy as a List	110
Model metadata.....	110
Feature Parameters	111
Working with files and directories	112
File reading and writing in macros	113
Frequently asked questions	116
Organising your macros.....	118

Recording the pmuser macro..... 119

Turning off error and warning messages and locking graphic updates..... 120

Recording a macro to set up NC preferences..... 121

Tips for programming macros 122

Index

125

Macros

A **macro** is a file which contains a sequence of commands to automate recurrent operations. You can create macros by recording operations as they occur in PowerMill, or by entering the commands directly into a text editor. Recorded macros have a **.mac** extension, and can be run from the **Macro** node in the Explorer.

You can record single or multiple macros to suit your needs. You can call a macro from within another macro.

There are two types of macros:

- The initialisation macro, **pmuser.mac**, is run when PowerMill starts. By default, a blank copy of this macro exists in the [C:\Program Files\Autodesk\PowerMillxxxxx\lib\macro](#) folder. By overwriting or adding PowerMill commands to it, you can set up your own default parameters and settings. You can also place the **pmuser** macro in the **pmill** folder, directly below your **Home** area. Doing this enables personalised macro settings for individual login accounts.
- User-defined macros are macros you define to automate various operations.



In addition to tailoring PowerMill by the creation of an initialisation macro, you can create macros for undrawing, drawing and resetting leads and links, setting NC preferences, defining regularly used machining sequences, and so on.

Creating macros

You can create macros by:

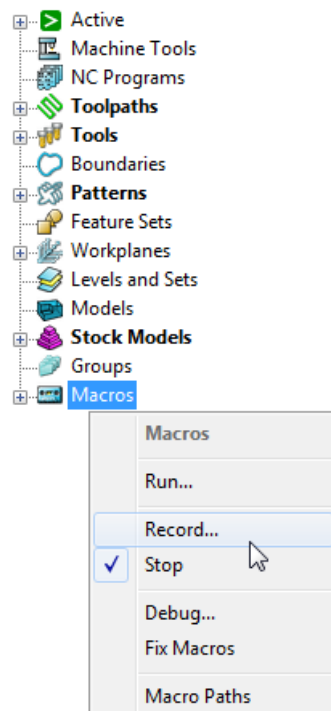
- Recording (see page 2) a sequence of commands within PowerMill.
- Writing your own macro (see page 5) using a text editor.

Recording macros in PowerMill

An easy way to create a macro is to record PowerMill commands as you work. Only the values that you change in the dialogs are recorded in the macro. Therefore, to record a value that is already set, you must re-enter it in a field, or re-select an option. For example, if the finishing tolerance is currently set to **0.1** mm, and you want the macro to store the same value, you must re-enter **0.1** in the **Tolerance** field during recording.


To record a macro:

- 1 From the **Macros** context menu, select **Record**.



This displays the **Select Record Macro File** dialog which is a standard Windows **Save** dialog.

- 2 Move to the appropriate directory, enter an appropriate **File name** and click **Save**.

The macro icon  **Macros** changes to red to show recording is in progress.



All dialog options that you want to include in your macro must be selected during its recording. If an option already has the desired value, re-enter it.

- 3 Work through the set of commands you want to record.
- 4 From the **Macros** context menu, select **Stop** to finish recording.

For more information, see Recording the pmuser macro (see page 119) and Recording the NC preference macro (see page 121).

Running macros

When you run a macro, the commands recorded in the macro file are executed.

- 1 Expand **Macros**, and select the macro you want to run.
- 2 From the individual macro menu, select **Run**.

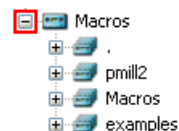


You can also run a macro by double-clicking its name in the Explorer.

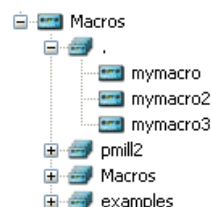
Running the macro you have just recorded

The location of the macro you have just recorded becomes the local folder. So, the macro you have just recorded is available in the local macro search path . However, the list of macros is not updated dynamically. To force an update:

- 1 Click next to **Macros** to collapse the contents.



- 2 Click next to **Macros** to expand and regenerate the contents.
- 3 Click next to to see the macros in this directory, which includes the one you have just created.

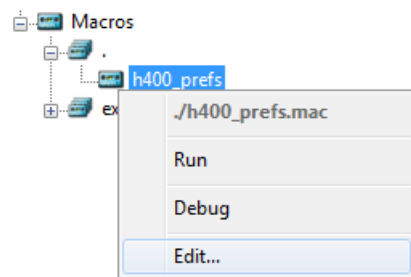


Editing macros

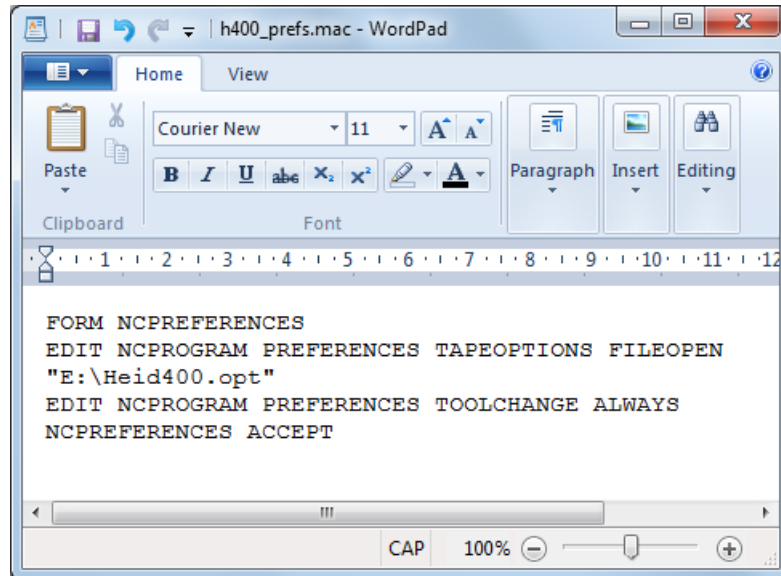
You can edit recorded macros to troubleshoot and correct any errors.

- 1 Expand **Macros** and select the macro you want to edit.

- 2 From the individual macro menu, select **Edit**.



A Windows WordPad document opens.



The text editor opened by default is the application associated with macro (.mac) files.

*Use the **Choose default program** option available in **Windows Explorer** to make changes to default file type associations.*

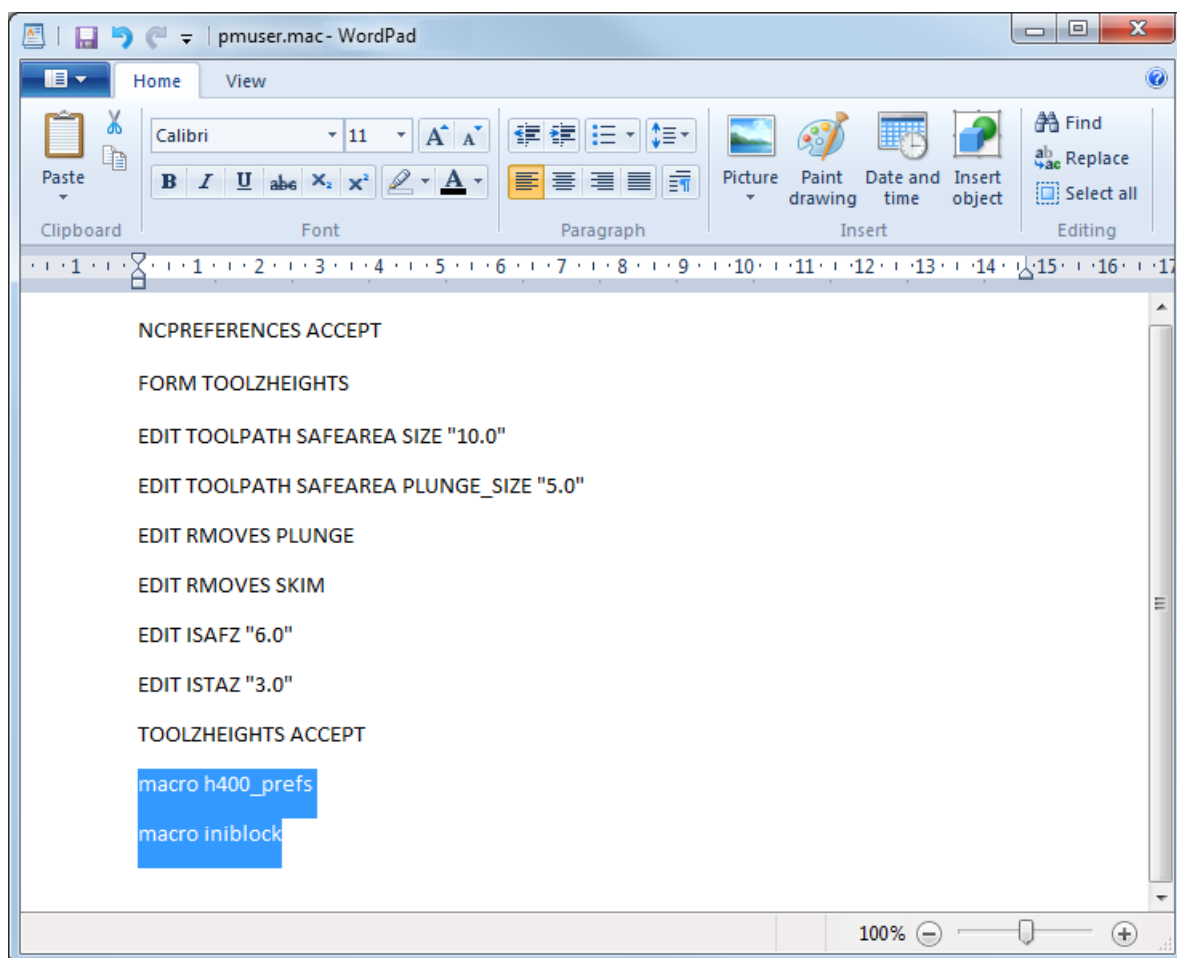
- 3 Edit the macro commands, and then save the file.

Running macros from within macros

You can create small macros that perform a single operation, and then call them from within a larger macro. This example shows how to add the **h400_prefs** macro and the **iniblock** macro to the **pmuser** macro.

- 1 From the **pmuser** macro context menu, select **Edit**.
- 2 Scroll to the bottom of the file, and add the following lines:
macro h400_prefs

macro iniblock



If you precede a line with two forward slash characters (//), it is treated as a comment, and is not executed.

- 3 Save and close **pmuser.mac**.
- 4 Exit and restart PowerMill to check that the settings from the **pmuser** macro are activated.

Writing your own macros

A more powerful way of creating macros is to write your own. The principles are described in the Macro User Guide (see page 8).

Macros enable you to:

- Construct expressions (see page 47).
- Use expressions to control macro flow (see page 22).
- Use a range of relational (see page 42) operators and logical (see page 43) operators.
- Evaluate both expressions (see page 47).

- Assign values to variables and parameters by using assignments (see page 28).

The **Menu** bar option **Help > Documentation > Parameters >Reference > Functions** lists all the standard functions you can use in macros.



For information about using parameter functions in setup sheets, see Using parameters in Setup Sheets.

PowerMill commands for macros

When you use PowerMill interactively, every menu click and entry in a dialog sends a command to the program. These are the commands that you must enter in your macro file if you want to drive PowerMill from a macro.

This example shows you how to:


- Find the PowerMill commands to include in your macros.
- Place them in a text editor such as WordPad.
- Display the macro in the Explorer.

To create a macro:

- 1 From the **Menu** bar, select **View > Toolbar > Command Window** to open the command window.
- 2 Select **Tools > Echo Commands** from the **Menu** bar to echo the issued commands in the command window.

```
Model\die
StockModel\
Group\
Macro\
PowerMill > Batch processing toolpath SteepAndShallow
Select template to use >Import Template
Template read correctly, it was created in PowerMILL Version 4.6.07.32.10441 on 24 APR 2003 10.00.53
PowerMill > Batch processing toolpath CornerFinishing_Auto
PowerMill > 0 toolpaths in session require batch processing
PowerMill > 0 toolpaths in session require batch processing
Overwrite 'true' file? 0 toolpaths in session require batch processing
PowerMill > 0 toolpaths in session require batch processing
0 toolpaths in session require batch processing
PowerMill >
```

- 3 To see the commands needed to calculate a block:

- a Click the **Block**  button on the **Main** toolbar.
- b When the **Block** dialog opens, click **Calculate**, and then click **Accept**.

The command window shows the commands issued:

```
PowerMill >  
Process Command : [FORM BLOCK\r]  
  
PowerMill >  
Process Command : [EDIT BLOCK RESET\r]  
  
PowerMill >  
Process Command : [BLOCK ACCEPT\r]  
  
PowerMill >
```

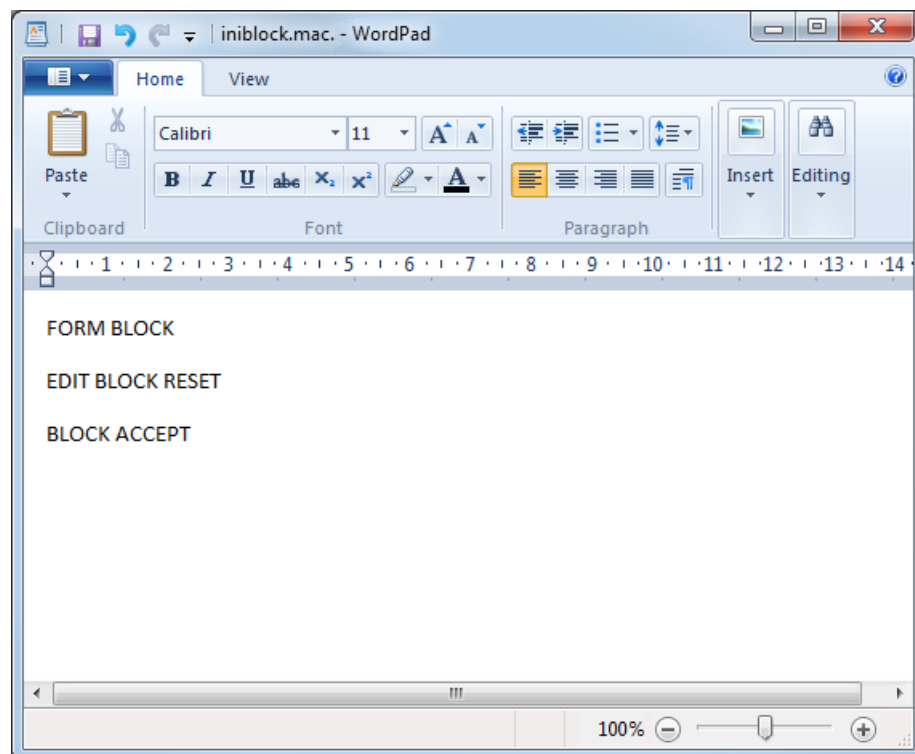
The commands are shown in square brackets; `\r` should be ignored. The commands you need are: **FORM BLOCK**, **EDIT BLOCK RESET**, and **BLOCK ACCEPT**.

- 4 Open WordPad, and enter the commands into it.



*The commands aren't case-sensitive so **FORM BLOCK** is the same as **Form Block** which is the same as **foRm bLock**.*

- 5 Save the file as say, **iniblock.mac**. The macro is added to the macro tree.



For more information see [Running macros](#) (see page 3).

Adding comments to macros

It is good practise 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 may help anyone examining the file to understand it. Comment lines start with `//`. For example:

```
// This macro imports my standard model, creates a block,  
// and a ball nosed tool.
```

It is also good practise to have comments explaining what each section of the macro file does. This may be obvious when you write the macro but later it may be difficult to understand. It is good practise to put the comments which describe commands before the actual commands:

```
// Clean all the Roughing boundaries  
MACRO Clean 'boundary\Roughing'
```

Another use of comments is to temporarily remove a command from a macro. When debugging or writing a macro, it is a good idea to write one step at a time and re-run the macro after each change. If your macro contains a lengthy calculation, or the recreation of toolpaths, you may want to temporarily comment out the earlier parts of the macro whilst checking the later parts. For example:

```
// Import the model  
// IMPORT TEMPLATE ENTITY TOOLPATH "Finishing/Raster-  
Flat-Finishing.ptf"
```

Macro User Guide

This example shows you how to use the PowerMill macro programming language to create a macro which prints the words of the counting song "Ten Green Bottles".

*10 green bottles sitting on the wall
10 green bottles sitting on the wall
And if 1 green bottle should accidentally fall
There will be 9 green bottles sitting on the wall*

*9 green bottles sitting on the wall
9 green bottles sitting on the wall
And if 1 green bottle should accidentally fall
There will be 8 green bottles sitting on the wall*

and so on until the last verse

*1 green bottle sitting on the wall
1 green bottle sitting on the wall
And if 1 green bottle should accidentally fall
There will be 0 green bottles sitting on the wall.*

The main steps are:

- 1 Creating the basic macro (see page 9).
- 2 Adding macro variables (see page 10).
- 3 Adding macro loops (see page 11).
- 4 Running macros with arguments (see page 12).
- 5 Decision making in macros (see page 14).
- 6 Using functions in macros (see page 16).
- 7 Using a SWITCH statement (see page 18).
- 8 Returning values from macros (see page 19).
- 9 Using a FOREACH loop in a macro (see page 22).
- 10 Using arrays in a FOREACH loop (see page 25).

Basic macro

This shows you how to create and run a basic macro using PowerMill's programming language.

- 1 In a text editor such as WordPad enter:

```
PRINT "10 green bottles sitting on the wall"  
PRINT "10 green bottles sitting on the wall"  
PRINT "And if 1 green bottle should accidentally fall"
```

```
PRINT "There will be 9 green bottles sitting on the wall"
```

- 2 Save the file as **example.mac**.
- 3 In PowerMill, from the **Tools** menu select **Toolbar > Command Window**.
- 4 From the **Macro** context menu, select **Run**. This displays the **Select Macro to Run** dialog.
- 5 Move to the appropriate directory, select **example.mac**, and click **Open**. The macro runs and the command windows displays the text enclosed in quotations marks (") in the macro.

```
PowerMill >  
PowerMill > Running Macro:- ./example.mac  
10 green bottles sitting on the wall  
10 green bottles sitting on the wall  
And if 1 green bottle should accidentally fall  
There will be 9 green bottles sitting on the wall  
PowerMill >
```

Adding macro variables

The first two lines of **example.mac** are the same. To minimise repetition (and for ease of maintenance) it is good practise to write the line once and recall it whenever it is needed. To do this you must create a local variable to hold the line of text.

You can create different types of variables (see page 27) in PowerMill. To store a line of text you must use a **STRING** variable.

- 1 Open **example.mac** in your text editor and change it to:

```
// Create a variable to hold the first line  
STRING bottles = "10 green bottles sitting on the wall"  
PRINT $bottles  
PRINT $bottles  
PRINT "And if 1 green bottle should accidentally fall"  
PRINT "There will be 9 green bottles sitting on the wall"
```



The first line is a comment which explains the second line.

- 2 Save the file as **example.mac**.
- 3 In PowerMill, **Run** the **Macro**. The command windows displays the same as before:

```
PowerMill >  
PowerMill > Running Macro:- ./example.mac  
10 green bottles sitting on the wall  
10 green bottles sitting on the wall  
And if 1 green bottle should accidentally fall  
There will be 9 green bottles sitting on the wall  
PowerMill >
```

You should be aware of several issues with variables:

- You must define all local variables before they are used, in this case `STRING bottles = "10 green bottles sitting on the wall"` defines the local variable **bottles**.
- The variable **bottles** is a local variable, so is only valid within the macro where it is defined. It is not a PowerMill variable. Typing it into the command window gives an error.

```
PowerMill >
PowerMill > PRINT $bottles
#ERROR: Invalid name
PowerMill >
PowerMill > |
```

- When you have defined a local variable you can use it as many times as you want in a macro.
- You can define as many local variables as you want in a macro.

Adding macro loops

There are two lines of the macro which are the same: `PRINT $bottles`. This is acceptable in this case because the line only appears twice, but if you wanted to repeat it 5 or 20 times it would be better to use a loop. PowerMill has three looping statements:

- WHILE (see page 64)
- DO - WHILE (see page 65)
- FOREACH (see page 63)

This example uses the WHILE statement to repeat the command 5 times.

- 1 Open **example.mac** in your text editor and change it to:

```
// Create a variable to hold the first line
STRING bottles = "10 green bottles sitting on the wall"

// Create a variable to hold the number of times
// you want to print the first line.
// In this case, 5
INT Count = 5

// Repeat while the condition Count is greater than 0
WHILE Count > 0 {
    // Print the line
    PRINT $bottles
    // Reduce the count by 1
    $Count = Count - 1
}

// Print the last two lines
PRINT "And if 1 green bottle should accidentally fall"
```

```
PRINT "There will be 9 green bottles sitting on the wall"
```



`$Count = Count - 1` is an assignment statement which is why the variable (`$Count`) to the left of `=` must be prefixed with `$`.



The empty lines are not necessary, but they make it easier to read the macro.

2 Save the file as **example.mac**.

3 In PowerMill, **Run the Macro**. The command windows displays:

```
PowerMill >
Select File > 10 green bottles sitting on the wall
10 green bottles sitting on the wall
10 green bottles sitting on the wall
10 green bottles sitting on the wall
10 green bottles sitting on the wall
10 green bottles sitting on the wall
And if 1 green bottle should accidentally fall
There will be 9 green bottles sitting on the wall
PowerMill >
```



*Changing `INT Count = 5` to `INT Count = 10` prints **10 green bottles sitting on the wall** ten times, rather than five.*

Running macros with arguments

The loop you added to **example.mac** works well if you always want to print **10 green bottles sitting on the wall** the same number of times. However, if you want to change the number of repetitions at run time, rather than editing the macro each time, it is much better to write the macro so it is given the number of repetitions. To do this you need to create a **Main FUNCTION** (see page 51).

1 Open **example.mac** in your text editor and change it to:

```
// Create a Main FUNCTION to hold the number of times
// you want to print the first line.
FUNCTION Main (INT Count) {

    // Create a variable to hold the first line
    STRING bottles = "10 green bottles sitting on the wall"

    // Repeat while the condition Count is greater than 0
    WHILE Count > 0 {
        // Print the line
        PRINT $bottles
        // Reduce the count by 1
        $Count = Count - 1
    }
}
```



```

// Print the last two lines
PRINT "If 1 green bottle should accidentally fall"
PRINT "There will be 9 green bottles sitting on the
wall"
}

```

2 Save the file as **example.mac**.

3 To run the macro you cannot select **Run** from the **Macro** context menu, as you need to give a value for **Count**. Therefore, in the command window type:

MACRO example.mac 5

Where **5** is the value for **Count**. The command windows displays:

```

PowerMill > MACRO example.mac 5
10 green bottles sitting on the wall
10 green bottles sitting on the wall
10 green bottles sitting on the wall
10 green bottles sitting on the wall
10 green bottles sitting on the wall
If 1 green bottle should accidentally fall
There will be 9 green bottles sitting on the wall
PowerMill > |

```



If you get a warning that the macro cannot be found, check you have created the necessary macro path (see page 118).

Adding your own functions

As well as a **Main** function you can create your own functions. This is useful as a way of separating out a block of code. You can use functions:

- to build up a library of useful operations
- to make a macro more understandable.



You can call a function any number of times within a macro.

This example separates out the printing of the first line into its own function so that the **Main** function is more understandable.

1 Open **example.mac** in your text editor and change it to:

```

FUNCTION PrintBottles(INT Count) {

// Create a variable to hold the first line
STRING bottles = "10 green bottles sitting on the
wall"

// Repeat while the condition Count is greater than
0
WHILE Count > 0 {
// Print the line
PRINT $bottles
// Reduce the count by 1
}
}

```

```

        $Count = Count - 1
    }
}

FUNCTION Main (INT Count) {

    // Print the first line Count number of times
    CALL PrintBottles(Count)

    // Print the last two lines
    PRINT "If 1 green bottle should accidentally fall"
    PRINT "There will be 9 green bottles sitting on the
    wall "
}

```

2 Save the macro.

3 Run the macro by typing **MACRO example.mac 5** in the command window.

```

PowerMill > MACRO example.mac 5
10 green bottles sitting on the wall
10 green bottles sitting on the wall
10 green bottles sitting on the wall
10 green bottles sitting on the wall
10 green bottles sitting on the wall
If 1 green bottle should accidentally fall
There will be 9 green bottles sitting on the wall
PowerMill > |

```

This produces the same result as before.



*The order of functions in a macro is irrelevant. For example, it does not matter whether the **Main** function is before or after the **PrintBottles** function.*



*It is important that each function name is unique and that the macro has a function called **Main**.*



You can have any number of functions in a macro.

Decision making in macros

The macro **example.mac** runs provided that you enter a positive argument. However, if you always want the **10 green bottles sitting on the wall** line printed at least once use:

- A **DO - WHILE** (see page 65) loop as it executes all the commands before testing the conditional expression.
- An **IF** (see page 55) statement.

DO - WHILE loop

1 Edit the **PrintBottles** function in **example.mac** to

```

FUNCTION PrintBottles(INT Count) {

    // Create a variable to hold the first line
    STRING bottles = "10 green bottles sitting on the
    wall"

    // Repeat while the condition Count is greater than
    0
    DO {
        // Print the line
        PRINT $bottles
        // Reduce the count by 1
        $Count = Count - 1
    } WHILE Count > 0
}

```

The main function remains unchanged:

```

FUNCTION Main (INT Count) {

    // Print the first line Count number of times
    CALL PrintBottles(Count)

    // Print the last two lines
    PRINT "And if 1 green bottle should accidentally
    fall"
    PRINT "There will be 9 green bottles sitting on the
    wall"
}

```

2 Type **MACRO example.mac 0** in the command window.

```

PowerMill > MACRO example.mac 0
10 green bottles sitting on the wall
If 1 green bottle should accidentally fall
There will be 9 green bottles sitting on the wall
PowerMill >

```

The **10 green bottles sitting on the wall** line is printed once.

IF statement

You can use an **IF** statement to ensure the **10 green bottles sitting on the wall** line is printed at least twice.

1 Edit the **Main** function in **example.mac** to:

```

FUNCTION Main (INT Count) {

    // Make sure that Count is at least two
    IF Count < 2 {
        $Count = 2
    }

    // Print the first line Count number of times
    CALL PrintBottles(Count)
}

```

```

// Print the last two lines
PRINT "And if 1 green bottle should accidentally
fall"
PRINT "There will be 9 green bottles sitting on the
wall"
}

```

The **PrintBottles** function remains unchanged:

```

FUNCTION PrintBottles(INT Count) {

    // Create a variable to hold the first line
    STRING bottles = "10 green bottles sitting on the
wall"

    // Repeat while the condition Count is greater than
    0
    WHILE Count > 0 {
        // Print the line
        PRINT $bottles
        // Reduce the count by 1
        $Count = Count - 1
    }
}

```

2 Type **MACRO example.mac 0** in the command window.

```

PowerMill > MACRO example.mac 0
10 green bottles sitting on the wall
10 green bottles sitting on the wall
And if 1 green bottle should accidentally fall
There will be 9 green bottles sitting on the wall
PowerMill >

```

The **10 green bottles sitting on the wall** line is printed twice.

More on functions in macros

So far you have only printed the first verse of the counting song "Ten Green Bottles". To make your macro print out all the verses you must change the **PrintBottles** function so it takes two arguments:

- **Count** for the number of times "X green bottles" is printed.
- **Number** for the number of bottles.

1 Edit the **PrintBottles** function in **example.mac** to

```

FUNCTION PrintBottles(INT Count, INT Number) {
    // Create a variable to hold the first line
    STRING bottles = String(Number) + " green bottles
sitting on the wall"

    // Repeat while the condition Count is greater than
    0
    WHILE Count > 0 {

```

```

        // Print the line
        PRINT $bottles
        // Reduce the count by 1
        $Count = Count - 1
    }
}

```

This adds a second argument to the **PrintBottles** function. It then uses a parameter function to convert the **Number** to a string value, **STRING (Number)**. It is then concatenated (+) with **green bottles sitting on the wall** to make up the **bottles** string.

2 Edit the **Main** function in **example.mac** to:

```

FUNCTION Main (INT Count) {
    // Make sure that Count is at least two
    IF Count < 2 {
        $Count = 2
    }

    // Start with ten bottles
    INT Bottles = 10

    WHILE Bottles > 0 {
        // Print the first line 'Count' number of times
        CALL PrintBottles(Count, Bottles)
        // Count down Bottles
        $Bottles = $Bottles - 1
        // Build the number of 'bottles_left' string
        STRING bottles_left = "There will be " +
            string(Bottles) + " green bottles sitting on the
            wall"
        // Print the last two lines
        PRINT "If 1 green bottle should accidentally fall"
        PRINT $bottles_left
    }
}

```

3 Type **MACRO example.mac 2** in the command window.

```

PowerMill > MACRO example.mac 2
10 green bottles sitting on the wall
10 green bottles sitting on the wall
If 1 green bottle should accidentally fall
There will be 9 green bottles sitting on the wall
9 green bottles sitting on the wall
9 green bottles sitting on the wall
If 1 green bottle should accidentally fall
There will be 8 green bottles sitting on the wall
...
...
If 1 green bottle should accidentally fall
There will be 0 green bottles sitting on the wall
PowerMill >

```



In **Main** when you **CALL PrintBottles** you give it two arguments **Count** and **Bottles** whilst within the **PrintBottles** function the **Bottles** argument is referred to as **Number**. The parameters passed to a function do not have to have the same names as they are called within the function.



The order you call the arguments is important.



Any changes made to the value of a parameter within a function does not alter the value of parameter in the calling function unless the parameter is defined as an **OUTPUT** (see page 19) value.

Using the SWITCH statement

So far you have used numerals to print the quantity of bottles but it would be better to use words. So, instead of printing **10 green bottles** ... print **Ten green bottles**

One way of doing this is to use a large **IF - ELSEIF** (see page 56) chain to select the text representation of the number. Another way is to use the **SWITCH** (see page 58) statement.

```
SWITCH Number {
  CASE 10
    $Text = "Ten"
    BREAK
  CASE 9
    $Text = "Nine"
    BREAK
  CASE 8
    $Text = "Eight"
    BREAK
  CASE 7
    $Text = "Seven"
    BREAK
  CASE 6
    $Text = "Six"
    BREAK
  CASE 5
    $Text = "Five"
    BREAK
  CASE 4
    $Text = "Four"
    BREAK
  CASE 3
    $Text = "Three"
    BREAK
  CASE 2
    $Text = "Two"
```

```

        BREAK
CASE 1
    $Text = "One"
    BREAK
DEFAULT
    $Text = "No"
    BREAK
}

```

The switch statement matches the value of its argument (in this case **Number**) with a corresponding case value and executes all the subsequent lines until it encounters a **BREAK** statement. If no matching value is found the **DEFAULT** is selected (in this case **No**).



***DEFAULT** is an optional step.*

Returning values from macros

This shows you how to create an **OUTPUT** variable from a **SWITCH** statement.

- 1 Create a new function called **NumberStr** containing the **SWITCH** statement in Using the SWITCH statement (see page 18) and a first line of:

```
FUNCTION NumberStr(INT Number, OUTPUT STRING Text) {
```

and a last line of:

```
}
```

- 2 Edit the **PrintBottles** function in **example.mac** to

```

FUNCTION PrintBottles(INT Count INT Number) {

    // Convert Number into a string
    STRING TextNumber = ''
    CALL NumberStr(Number,TextNumber)

    // Create a variable to hold the first line
    STRING bottles = TextNumber + " green bottles
    sitting on the wall"

    // Repeat while the condition Count is greater than
    0
    WHILE Count > 0 {
        // Print the line
        PRINT $bottles
        // Reduce the count by 1
        $Count = Count - 1
    }
}

```

This adds the **OUTPUT** variable to the **PrintBottles** function.

3 Edit the **Main** function in **example.mac** to:

```
FUNCTION Main (INT Count) {

    // Make sure that Count is at least two
    IF Count < 2 {
        $Count = 2
    }

    // Start with ten bottles
    INT Bottles = 10

    WHILE Bottles > 0 {
        // Print the first line Count number of times
        CALL PrintBottles(Count, Bottles)
        // Countdown Bottles
        $Bottles = $Bottles - 1

        // Convert Bottles to string
        STRING BottlesNumber = ''
        CALL NumberStr(Bottles, BottlesNumber)

        // Build the number of bottles left string
        STRING bottles_left = "There will be " +
            lcase(BottlesNumber) + " green bottles sitting on
            the wall"
        // Print the last two lines
        PRINT "If one green bottle should accidentally
            fall"
        PRINT $bottles_left
    }
}
```

The **BottlesNumber** variable is declared in the **WHILE** loop of the **MAIN** function.



Each code block or function can define its own set of local variables; the scope of the variable is from its declaration to the end of the enclosing block or function.

4 Add the **NumberStr** function into **example.mac**.

```
FUNCTION PrintBottles(INT Count, INT Number) {

    // Convert Number into a string
    STRING TextNumber = ''
    CALL NumberStr(Number, TextNumber)

    // Create a variable to hold the first line
    STRING bottles = TextNumber + " green bottles sitting
    on the wall"

    // Repeat while the condition Count is greater than 0
```



```

WHILE Count > 0 {
    // Print the line
    PRINT $bottles
    // Reduce the count by 1
    $Count = Count - 1
}
}

FUNCTION Main (INT Count) {

    // Make sure that Count is at least two
    IF Count < 2 {
        $Count = 2
    }

    // Start with ten bottles
    INT Bottles = 10

    WHILE Bottles > 0 {
        // Print the first line Count number of times
        CALL PrintBottles(Count, Bottles)
        // Countdown Bottles
        $Bottles = $Bottles - 1

        // Convert Bottles to string
        STRING BottlesNumber = ''
        CALL NumberStr(Bottles, BottlesNumber)

        // Build the number of bottles left string
        STRING bottles_left = "There will be " +
            lcase(BottlesNumber) + " green bottles sitting on the
            wall"
        // Print the last two lines
        PRINT "If one green bottle should accidentally fall"
        PRINT $bottles_left
    }
}

FUNCTION NumberStr(INT Number, OUTPUT STRING Text) {
    SWITCH Number {
        CASE 10
            $Text = "Ten"
            BREAK
        CASE 9
            $Text = "Nine"
            BREAK
        CASE 8
            $Text = "Eight"
            BREAK
        CASE 7
            $Text = "Seven"
    }
}

```

```

        BREAK
CASE 6
    $Text = "Six"
    BREAK
CASE 5
    $Text = "Five"
    BREAK
CASE 4
    $Text = "Four"
    BREAK
CASE 3
    $Text = "Three"
    BREAK
CASE 2
    $Text = "Two"
    BREAK
CASE 1
    $Text = "One"
    BREAK
DEFAULT
    $Text = "No"
    BREAK
    }
}

```

To run the macro:

Type **MACRO example.mac 2** in the command window.

```

PowerMill > MACRO example.mac 2
Ten green bottles sitting on the wall
Ten green bottles sitting on the wall
If one green bottle should accidentally fall
There will be nine green bottles sitting on the wall
Nine green bottles sitting on the wall
Nine green bottles sitting on the wall
If one green bottle should accidentally fall
There will be eight green bottles sitting on the wall
...
...
If 1 green bottle should accidentally fall
There will be 0 green bottles sitting on the wall
PowerMill >

```

Using a FOREACH loop in a macro

This example shows you how to use a **FOREACH** (see page 63) loop to control the number of bottles rather than a **WHILE** loop.

- 1 Edit the **Main** function in **example.mac** to:

```

FUNCTION Main (INT Count) {

    // Make sure that Count is at least two
    IF Count < 2 {
        $Count = 2
    }
}

```

```

    }

    FOREACH Bottles IN {10,9,8,7,6,5,4,3,2,1} {
        // Print the first line Count number of times
        CALL PrintBottles(Count, Bottles)
        // Countdown Bottles
        $Bottles = $Bottles - 1

        // Convert Bottles to string
        STRING BottlesNumber = ''
        CALL NumberStr(Bottles, BottlesNumber)

        // Build the number of bottles left string
        STRING bottles_left = "There will be " +
        lcase(BottlesNumber) + " green bottles sitting on
        the wall"
        // Print the last two lines
        PRINT "If one green bottle should accidentally
        fall"
        PRINT $bottles_left
    }
}

```

The rest of **example.mac** remains unaltered.

```

FUNCTION PrintBottles(INT Count, INT Number) {

    // Convert Number into a string
    STRING TextNumber = ''
    CALL NumberStr(Number,TextNumber)

    // Create a variable to hold the first line
    STRING bottles = TextNumber + " green bottles sitting
    on the wall"

    // Repeat while the condition Count is greater than 0
    WHILE Count > 0 {
        // Print the line
        PRINT $bottles
        // Reduce the count by 1
        $Count = Count - 1
    }
}

FUNCTION Main (INT Count) {

    // Make sure that Count is at least two
    IF Count < 2 {
        $Count = 2
    }

    FOREACH Bottles IN {10,9,8,7,6,5,4,3,2,1} {

```

```

// Print the first line Count number of times
CALL PrintBottles(Count, Bottles)
// Countdown Bottles
$Bottles = $Bottles - 1

// Convert Bottles to string
STRING BottlesNumber = ''
CALL NumberStr(Bottles, BottlesNumber)

// Build the number of bottles left string
STRING bottles_left = "There will be " +
lcase(BottlesNumber) + " green bottles sitting on the
wall"
// Print the last two lines
PRINT "If one green bottle should accidentally fall"
PRINT $bottles_left
}
}

FUNCTION NumberStr(INT Number, OUTPUT STRING Text) {
  SWITCH Number {
    CASE 10
      $Text = "Ten"
      BREAK
    CASE 9
      $Text = "Nine"
      BREAK
    CASE 8
      $Text = "Eight"
      BREAK
    CASE 7
      $Text = "Seven"
      BREAK
    CASE 6
      $Text = "Six"
      BREAK
    CASE 5
      $Text = "Five"
      BREAK
    CASE 4
      $Text = "Four"
      BREAK
    CASE 3
      $Text = "Three"
      BREAK
    CASE 2
      $Text = "Two"
      BREAK
    CASE 1
      $Text = "One"
      BREAK
  }
}

```

```

    DEFAULT
    $Text = "No"
    BREAK
  }
}

```



*You do not need to declare the type or initial value of the **Bottles** variable as the **FOREACH** loop handles this.*

To run the macro:

Type **MACRO example.mac 2** in the command window.

```

PowerMill > MACRO example.mac 2
Ten green bottles sitting on the wall
Ten green bottles sitting on the wall
If one green bottle should accidentally fall
There will be nine green bottles sitting on the wall
Nine green bottles sitting on the wall
Nine green bottles sitting on the wall
If one green bottle should accidentally fall
There will be eight green bottles sitting on the wall
...
...
If 1 green bottle should accidentally fall
There will be 0 green bottles sitting on the wall
PowerMill >

```

This gives exactly the same output as the Returning values from macros (see page 19) example. It shows you an alternative way of creating the same output.

Using arrays in a FOREACH loop

This example shows you how to use an array (see page 33) in a **FOREACH** loop, rather than using a list, to control the number of bottles.

- 1 Edit the **Main** function in **example.mac** to:

```

FUNCTION Main (INT Count) {

    // Make sure that Count is at least two
    IF Count < 2 {
        $Count = 2
    }

    // Define an array of bottle numbers
    INT ARRAY BottleArray[10] = {10,9,8,7,6,5,4,3,2,1}

    FOREACH Bottles IN BottleArray {
        // Print the first line Count number of times
        CALL PrintBottles(Count, Bottles)
        // Count down Bottles
        $Bottles = $Bottles - 1

        // Convert Bottles to string
        STRING BottlesNumber = ''
    }
}

```

```

CALL NumberStr(Bottles, BottlesNumber)

// Build the number of bottles left string
STRING bottles_left = "There will be " +
lcase(BottlesNumber) + " green bottles sitting on
the wall"
// Print the last two lines
PRINT "If one green bottle should accidentally
fall"
PRINT $bottles_left
}
}

```

The rest of **example.mac** remains unaltered.

2 Type **MACRO example.mac 2** in the command window.

```

PowerMill > MACRO example.mac 2
Ten green bottles sitting on the wall
Ten green bottles sitting on the wall
If one green bottle should accidentally fall
There will be nine green bottles sitting on the wall
Nine green bottles sitting on the wall
Nine green bottles sitting on the wall
If one green bottle should accidentally fall
There will be eight green bottles sitting on the wall
...
If 1 green bottle should accidentally fall
There will be 0 green bottles sitting on the wall
PowerMill >

```


This gives exactly the same output as the Returning values from macros (see page 19) example. It shows you an alternative way of creating the same output.

Pausing a macro for user interaction

You can pause a running macro to allow user input, such as the selection of surfaces or curves. The command to do this is:

```
MACRO PAUSE "User help instructions"
```

This displays a dialog containing the supplied text and a button to allow the user to **RESUME** the macro.

When the macro is paused, users can perform any actions within PowerMill, with the exception of running another macro. The current macro remains paused until the user clicks the **RESUME** button. If the user closes the dialog, by clicking the dialog close icon , this ends any currently running macros, including the paused macro.

For example:

```

GET EXAMPLES 'cowling.dgk'
ROTATE TRANSFORM TOP
CREATE TOOL ; BALLNOSED
EDIT TOOL ; DIAMETER 10
EDIT BLOCK RESET

```

```
CREATE BOUNDARY ; SELECTED
STRING Msg = "Select surfaces for boundary, and
press"+crlf+"RESUME when ready to continue"
EDIT BLOCK RESET
MACRO PAUSE $Msg
EDIT BOUNDARY ; CALCULATE
```

If you do not enter a string after `MACRO PAUSE` the macro pauses but does not display a `RESUME` dialog. To resume the macro either type `MACRO RUN` or provide another mechanism to continue the macro.

Variables in macros

You can create variables in macros just as you can in a PowerMill project. When you create a variable in a macro, it has the same properties as a PowerMill parameter, and can store either a value or an expression.



There are some restrictions on the use of macro variables.

- Variable names must start with an alphabetic character (a-z, A-Z) and may contain any number of subsequent alphanumeric characters (a-z, A-Z, 1-9, _). For example, you can name a variable **Count1** but not **1Count**.
- Variable names are case insensitive. For example, **Count**, **count**, and **CoUnT** all refer to the same variable.
- All variables must have a type, which can be:
 - INT** — Integer numbers. For example, 1, 21, 5008.
 - REAL** — Real numbers. For example, 201, -70.5, 66.0.
 - STRING** — A sequence of characters. For example, hello.
 - BOOL** — Truth values, either 0 (false) or 1 (true).
 - ENTITY** — A unique value that references an existing PowerMill entity.
 - Object** — A collection of parameters that PowerMill groups together, such as **Block**, or **Connections**.
- You must declare the variable type, for example:


```
INT Count = 5
REAL Diameter = 2.5
STRING Tapefile = "MyFile.tap"
```
- You can access any of the PowerMill parameters in variable declarations, expressions, or assignments.

- Any variables you create in a macro are only accessible from within the macro. When the macro has finished the variable is no longer accessible and cannot be used in expressions or other macros.
- If you need to create a variable that can be used at any time in a PowerMill project then you should create a **User Parameter**.

Assigning parameters

When you assign a value to a variable the expression is evaluated and the result is assigned, the actual expression is not retained. This is the same as using the **EVAL** modifier in the PowerMill parameter **EDIT PAR** command. These two statements are equivalent:

```
EDIT PAR "Stepover" EVAL "Tool.Diameter * 0.6"
$Stepover = Tool.Diameter * 0.6
```



*Variable and parameter names may optionally be prefixed with a \$ character. In most cases, you can omit the \$ prefix, but it **MUST** be used when you assign a value to either a variable or parameter within a macro.*

Inputting values into macros

An input dialog enables you to enter specific values into a macro.

The basic structure is:

```
$<variable> = INPUT <string-prompt>
```

This displays an input dialog with a specified prompt as its title which enables you to enter a value.

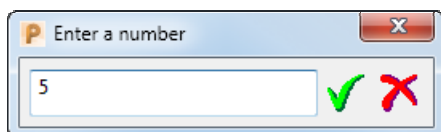


If you add an input dialog you should consider adding an error function to check the value entered is reasonable.

For example:

```
string prompt = "Enter a number"
$i = input $prompt
$error = ERROR i
}
```

produces this dialog:



You can also use **INPUT** in the variable definition.

For example:


```
REAL X = INPUT "Enter a number"
```

For more information see User selection of entities in macros (see page 31).

Asking a Yes/No question

A Yes/No query dialog is a very simple dialog.

Selecting **Yes** assigns **1** to the variable.

Selecting **No** assigns **0** to the variable.

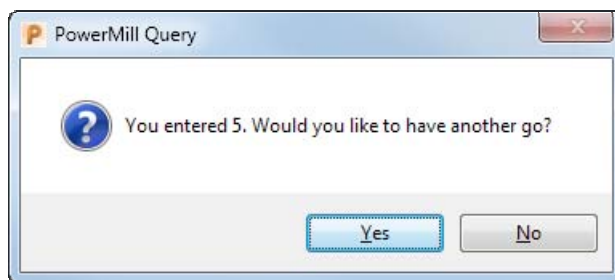
The basic structure is:

```
$<variable> = QUERY <string-prompt>
```

For example:

```
string yesnoprmt = "You entered 5. Would you like to have another go?"  
bool carryon = 0  
$carryon = query $yesnoprmt
```

produces this dialog:



Creating a message dialog

There are three types of message dialogs:

- Information dialogs
- Warning dialogs
- Error dialogs

The basic structure is:

```
MESSAGE INFO|WARN|ERROR <expression>
```

For example, an input dialog to enter a number into a macro:

```
real i = 3  
string prompt = "Enter a number"  
do {  
  bool err = 0  
  do {  
    $i = input $prompt  
    $err = ERROR i  
    if err {  
      $prompt = "Please 'Enter a number' "  
    }  
  }  
}
```

```

    } while err
    string yesnoprompt = "You entered " + string(i) + ".
    Would you like to have another go?"
    bool carryon = 0
    $carryon = query $yesnoprompt
} while $carryon
message info "Thank you!"

```

An example to find out if a named toolpath exists:

```

string name = ""
$name = input "Enter the name of a toolpath"
if pathname('toolpath',name) == "" {
    message error "Sorry. Couldn't find toolpath " + name
} else {
    message info "Yes! Toolpath " + name + " exists!"
}

```

Carriage return in dialogs

You can specify a carriage return to control the formatting of the text in a message dialog using `crlf`.

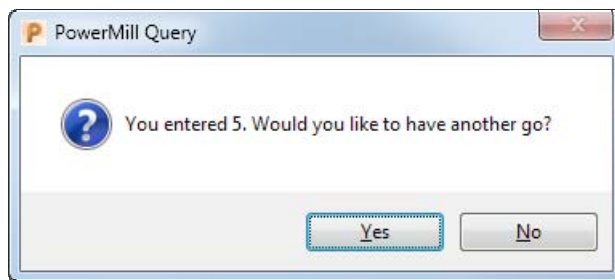
For example, looking at the input dialog to enter a number into a macro:

```

real i = 3
string prompt = "Enter a number"
do {
    bool err = 0
    do {
        $i = input $prompt
        $err = ERROR i
        if err {
            $prompt = "Please 'Enter a number'"
        }
    } while err
    string yesnoprompt = "You entered " + string(i) + "." +
    crlf + " Would you like to have another go?"
    bool carryon = 0
    $carryon = query $yesnoprompt
} while $carryon
message info "Thank you!"

```

produces this query dialog:



User selection of entities in macros

Use the **INPUT** command to prompt the user to select a specific entity in PowerMill, such as a toolpath or a tool. You can use this to:

- Display a list of available entities
- Prompt the user to select one of them.

For example, to list all the available tools and then ask the user to select one:

```
STRING ToolName = INPUT ENTITY TOOL "Please select a Tool."
```



This command returns the name of the tool the user selected.

This example creates two folders, creates two tool in each folder, then asks the user to select one of the tools:

```
// Create some tools in folders
CREATE FOLDER 'Tool' 'Endmills'
CREATE IN 'Tool\Endmills' TOOL 'End 20' ENDMILL
EDIT TOOL ; DIAMETER 20
CREATE IN 'Tool\Endmills' TOOL 'End 10' ENDMILL
EDIT TOOL ; DIAMETER 10
CREATE FOLDER 'Tool' 'Balls'
CREATE IN 'Tool\Balls' TOOL 'Ball 12' BALLNOSED
EDIT TOOL ; DIAMETER 12
CREATE IN 'Tool\Balls' TOOL 'Ball 10' BALLNOSED
EDIT TOOL ; DIAMETER 10
// Prompt user to pick one
STRING ToolName = ''
$ToolName = INPUT ENTITY TOOL "Please select a Tool."
```

You can also ask for the selection of a number of entities. The result is the list of entities selected, which can be assigned to either a list of strings, or list of entities.

```
ENTITY LIST $Selected_Toolpaths = INPUT ENTITY MULTIPLE
toolpath "which toolpaths do you want to check?"
STRING LIST ToolpathNames = INPUT ENTITY MULTIPLE
TOOLPATH "Select toolpaths to check"
```

You can then iterate over the user selection with a **FOREACH** loop:

```

FOREACH $tp in ToolpathNames {
    ACTIVATE TOOLPATH $tp.Name
    EDIT COLLISION APPLY
}

```

User selection from a list of options

You can use the **INPUT** command to prompt the user to select from a list of options that your macro supplies. The syntax for this is:

```
INT value = INPUT CHOICE <string-array> <prompt>
```

For example, suppose you have a machining macro where everything is setup except that you want to give the user the choice of cut direction to use. You can do this by using a CHOICE input as follows:

```

// Create an array of strings from the CutDirection
parameter
    STRING ARRAY Opts[] = values(CutDirection)
    INT C = INPUT CHOICE $Opts "Choose the Cut Direction
    you want"
    $CutDirection = $C
}

```

Or for another example, you can increase or decrease the number of options the user can select. You can limit the options available to only one, such as **Gouge Check** or **Collision Check** a toolpath, or you can increase the options available so the user can choose between the two options. To create this list, enter the following:

```

STRING ARRAY Opts[] = {"Gouge check only", "Collision
check only", "Gouge and Collision check"} INT C = INPUT
CHOICE $Opts "Pick an option"
SWITCH $C {
    CASE 0:
        MACRO "Gouge_Check.mac"
        BREAK
    CASE 2:
        MACRO "Gouge_Check.mac"
        // Intended fall through to next command
    CASE 1:
        MACRO "Collision_Check.mac"
        BREAK
}

```



*The above example uses the 'fall through' behavior of cases within a switch block (see page 58). If you are not used to using the switch statement you can use an **IFELSE** statement instead:*

```

IF $C==0 {
    MACRO "Gouge_Check.mac"
}

```

```

} ELSEIF $C==1 {
    MACRO "Collision_Check.mac"
} ELSEIF $C==2 {
    MACRO "Gouge_Check.mac"
    MACRO "Collision_Check.mac"
}

```

User selection of a file name

You can prompt your user for a filename with Use the **FILESELECT** command to prompt the user for a file name. This command displays an **Open** dialog which enables user to browse for a file.

For example:

```

STRING Filename = ''
$Filename = FILESELECT "Please select a pattern file"

```

Arrays and lists

Arrays

In addition to simple variables of type INT, REAL, or STRING you can also have arrays of these types. When you declare an array you must initialise all of its members using an initialisation list. When you have specified an array you cannot change its size. The syntax for an array is:

```

BASIC-TYPE ARRAY name[n] = {...}

```

For example, to declare an array of three strings:

```

STRING ARRAY MyArray[3] = {'First','Second','Third'}

```

All the items in the initialisation list must be the same **BASIC-TYPE** as the array.

You can access the items of the array by subscripting. The first item in the array is subscript 0. For example:

```

INT Index = 0
WHILE Index < size(MyArray) {
    PRINT MyArray[Index]
    $Index = Index + 1
}

```

Prints:

First

Second

Third

If you leave the size of the array empty, then PowerMill determines its size from the number of elements in the initialisation list. For example:

```
STRING ARRAY MyArray[] =
{'First','Second','Third','Fourth'}
PRINT = size(MyArray)
```

Prints:

4

Lists

PowerMill also has a LIST type. The main difference between a list and an array is that the list does not have a fixed size, so you can add and remove items to it. You can create lists:

- that are empty to start with
- from an initialisation list
- from an array.

```
// Create an empty list
STRING LIST MyStrings = {}
// Create a list from an array
STRING LIST MyList = MyArray
// Create a list using an initialisation list
STRING LIST MyListTwo = {'First','Second'}
```

You can use two inbuilt functions `add_first()` and `add_last()` to add items to a list.

For example using the inbuilt function `add_last()`:

```
CREATE PATTERN Daffy
CREATE PATTERN Duck
// Create an empty list of strings
STRING LIST Patterns = {}
FOREACH pat IN folder('Pattern') {
    // Add the name of the pattern to the list
    int s = add_last(Patterns, pat.Name)
}
FOREACH name IN Patterns {
    PRINT = $name
}
```

Prints:

Daffy

Duck

You can also add items to the front of a list by using the inbuilt function `add_first()`:

```
CREATE PATTERN Daffy
```

```

CREATE PATTERN Duck
// Create an empty list of strings
STRING LIST Patterns = {}
FOREACH pat IN folder('Pattern') {
    // Add the name of the pattern to the list
    int s = add_first(Patterns, pat.Name)
}
FOREACH name IN Patterns {
    PRINT = $name
}

```

Prints:

Duck

Daffy

Using lists

A list, like an array, contains multiple values. You can create a list with initial values:

```
INT LIST MyList = {1,2,3,4}
```



Unlike an ARRAY, you do not use the [] syntax.

You can specify an empty list:

```
INT LIST MyEmptyList = {}
```

You can use lists anywhere you might use an array. For instance, you can use a list in a FOREACH loop:

```

FOREACH i IN MyList {
    PRINT = i
}

```

or to initialise an array:

```
INT ARRAY MyArray[] = MyList
```

You can also use an array to initialise a list:

```
INT LIST MyList2 = MyArray
```

You can pass a list to macro functions that expect an array:

```

FUNCTION PrintArray(INT ARRAY MyArray) {
    FOREACH i IN Myarray {
        PRINT = i
    }
}

FUNCTION Main() {
    INT LIST MyList = {10,20,30,40}
    CALL PrintArray(MyList)
}

```

You can access the elements of a list with a FOREACH loop, or you can use the array subscripting notation:

```
INT Val =  MyList[2]
```

Adding items to a list summary

The main differences between a list and an array is that a list can have items added to it and removed from it.

To add an item to a list you can use either of the inbuilt functions `add_first()` or `add_last()`.

For example, to collect the names of all the toolpaths in a folder:

```
// Create an empty list
STRING LIST TpNames = {}

FOREACH tp IN folder('Toolpath\MyFolder') {
    INT Size = add_last(TpNames, tp.name)
}
```

For more information see Adding comments to macros (see page 88).

Removing items from a list summary

The main differences between a list and an array is that a list can have items added to it and removed from it.

To remove an item from a list you can use either of the inbuilt functions `remove_first()` or `remove_last()`.

For example, if you have a list of toolpath names some of which are batched and you want to ask the user whether they want them calculated now. You can use a function which removes calculated toolpaths from the list and creates a query message for the rest.

```
FUNCTION CalculateNow(STRING LIST TpNames) {
    // Cycle through the list
    FOREACH Name IN TpNames {
        IF entity('toolpath',Name).Calculated {
            // Toolpath already calculated so
            // remove name from list
            BOOL success = remove(TpNames,Name)
        }
    }
    // Do we have any names left
    IF size(TpNames) > 0 {
        // Build the prompt string
        STRING Msg = "These toolpaths are uncalculated"
        FOREACH name IN TpNames {
            $Msg = Msg + CRLF + name
        }
    }
}
```



```

$Msg = Msg + CRLF + "Do you want to calculate them
now?"
// Ask the user if they want to proceed
bool yes = 0
$yes = QUERY $msg
IF yes {
    // Loop through the toolpaths and calculate them
    WHILE size(TpNames) > 0 {
        STRING Name = remove_first(TpNames)
        ACTIVATE TOOLPATH $Name
        EDIT TOOLPATH ; CALCULATE
    }
}
}
}

```

You could use a FOREACH loop rather than a WHILE loop:

```

FOREACH Name IN TpNames {
    ACTIVATE TOOLPATH $Name
    EDIT TOOLPATH ; CALCULATE
}

```

PowerMill has an inbuilt function which enables you to remove duplicate items from a list: `remove_duplicates`. For example, to determine how many different tool diameters there are in your toolpaths you could add the tool diameters from each toolpath and then remove the duplicates:

```

REAL LIST Diameters = {}
FOREACH tp IN folder('toolpath') {
    INT s = add_first(Diameters, tp.Tool.Diameter)
}
INT removed = remove_duplicates(Diameters)

```

For more information, see Removing items from a list (see page 88) or Removing duplicate items in a list (see page 87).

Building a list

You can use the inbuilt `member()` function in a macro function to build a list of tool names used by toolpaths or boundaries without any duplicates:

```

FUNCTION ToolNames(STRING FolderName, OUTPUT STRING LIST
ToolNames) {

    // loop over all the items in FolderName
    FOREACH item IN folder(FolderName) {

        // Define local working variables
        STRING Name = ''
        INT size = 0
    }
}

```

```

// check that the item's tool exists
// it might not have been set yet
IF entity_exists(item.Tool) {
    // get the name and add it to our list
    $Name = item.Tool.Name
    IF NOT member(FolderName, Name) {
        $dummy = add_last(FolderName, Name)
    }
}

// Check whether this item has a reference tool
// and that it has been set
IF active(item.ReferenceTool) AND
entity_exists(item.ReferenceTool) {
    // get the name and add it to our list
    $Name = item.ReferenceTool.Name
    IF NOT member(FolderName, Name) {
        $dummy = add_last(FolderName, Name)
    }
}
}
}
}

```

As this function can work on any toolpath or boundary folder, you can collect all the tools used by the toolpaths in one list and all of the tools used by boundaries in another list. You can do this by calling the macro function twice:

```

STRING LIST ToolpathTools = {}
STRING LIST BoundaryTools = {}
CALL ToolNames('Toolpath',ToolpathTools)
CALL ToolNames('Boundary',BoundaryTools)

```

To return a list containing the items from both sets with any duplicates removed:

```

STRING LIST UsedToolNames = set_union(ToolpathTools,
BoundaryTools)

```

Subtract function

You can use the `subtract()` function to determine what happened after carrying out a PowerMill command. For example, suppose you to find out if any new toolpaths are created during a toolpath verification. If you get the list of toolpath names before the operation, and the list of names after the operation, and then subtract the 'before' names from the 'after' names you are left with the names of any new toolpaths.

```

FUNCTION GetNames(STRING FolderName, OUTPUT STRING LIST
Names) {
    FOREACH item IN folder(FolderName) {
        INT n = add_last(Names, item.Name)
    }
}

```

```

    }
}

FUNCTION Main() {

    STRING LIST Before = {}
    CALL GetNames('toolpath',Before)

    EDIT COLLISION APPLY

    STRING LIST After = {}
    CALL GetNames('toolpath',After)
    STRING LIST NewNames = subtract(After, Before)

    IF is_empty(NewNames) {
        PRINT "No new toolpaths were created."
    } ELSE {
        PRINT "The new toolpaths created are:"
        FOREACH item IN NewNames {
            PRINT = item
        }
    }
}

```

Entity variables

PowerMill has a special variable type **ENTITY**. You can use **ENTITY** variables to refer to existing PowerMill entities such as toolpaths, tools, boundaries, patterns, workplanes, and so on. You cannot use this command to create new entities.

For example:

```

// create an entity variable that references boundary
entity 'Duck'
ENTITY Daffy = entity('boundary','Duck')

```

The inbuilt functions, such as `folder()` return lists of entities so you can store the result of the call in your own list and array variables:

For example:

```

ENTITY List Toolpaths = folder('toolpath')

```

When looping over folder items in a **FOREACH** the loop variable that is automatically created has the type **ENTITY**. Therefore the following code is syntactically correct:

```

FOREACH tp IN folder('toolpath') {
    ENTITY CurrentTP = tp
    PRINT = CurrentTP.Name
}

```

You can also pass **ENTITY** variables to functions (and passed back from function) by using an **OUTPUT** argument:

For example:

```
FUNCTION Test(OUTPUT ENTITY Ent) {
    $Ent = entity('toolpath','2')
}

FUNCTION Main() {
    ENTITY TP = entity('toolpath','1')
    CALL Test(TP)
    PRINT = TP.Name
}
```

Additionally, you can use an **ENTITY** variable anywhere in PowerMill that is expecting a entity name.

For example:

```
ENTITY tp = entity('toolpath','1')
ACTIVATE TOOLPATH $tp
```

Object variables

PowerMill has a variable type called OBJECT which can hold any collection of variables that PowerMill pre-defines, such as Block or Connections.

For example:

```
// Get the current set of block parameters
OBJECT myObject = Block
// Activate a toolpath (this may change the block)
ACTIVATE TOOLPATH "Daffy"
// Reset the block to its old state
$Block = myObject
```

Whilst you cannot create an ARRAY of OBJECT you can create a LIST of OBJECTs:

For example:

```
OBJECT LIST myObjects = {Block,Connections}
FOREACH ob IN myObjects {
    PRINT PAR "ob"
}
```

As you can see from the above example, each object in a list may be different. It is the responsibility of the macro writer to keep track of the different types of OBJECT. PowerMill has an inbuilt function `get_typename()` to help with this.

For example:

```
OBJECT LIST myObjects = {Block,Connections}
FOREACH ob IN myObjects {
    PRINT = get_typename(ob)
}
```

```
}
```

Which prints:

Block

ToolpathConnections

As with all lists, you can also access the elements by index:

```
PRINT = get_typeof(myObjects[0])
```

```
PRINT = get_typeof(myObjects[1])
```

Objects can also be passed to and from macro **FUNCTIONs**.

For example:

```
FUNCTION myBlkFunction(OBJECT blk) {
  IF get_typeof(blk) != "Block" {
    MESSAGE ERROR "Expecting a Block object"
    MACRO ABORT
  }
  // Code that works on block objects
}
// Find block with maximum zrange
FUNCTION myZrangeBlockFunc(OUTPUT OBJECT Blk) {
  // The
  REAL zrange = 0
  FOREACH tp IN folder('toolpath') {
    // find zlength of this block
    REAL z = tp.Block.Limits.ZMax - tp.Block.Limits.ZMin
    IF z > zrange {
      // Copy if longer than previously
      $Blk = Block
      $zrange = z
    }
  }
}
```

Vectors and points

In PowerMill vectors and points are represented by an array of three reals.

PowerMill contains point and vector parameters, for example the **Workplane.Origin**, **Workplane.ZAxis**, **ToolAxis.Origin**, and **ToolAxis.Direction**. You can create your own vector and point variables:

```
REAL ARRAY VecX[] = {1,0,0}
REAL ARRAY VecY[] = {0,1,0}
REAL ARRAY VecZ[] = {0,0,1}
REAL ARRAY MVecZ[] = {0,0,-1}
```

```
REAL ARRAY Orig[] = {0,0,0}
```

For more information, see the inbuilt Vectors and points functions (see page 70)

Comparing variables

Comparing variables enables you to check information and defines the course of action to take when using **IF** (see page 55) statements and **WHILE** (see page 64) statements.

The result of a comparison is either true or false. When true the result is **1**, when false the result is **0**.

A simple comparison may consist of two variables with a relational operator between them:

Relational operator		Description
Symbol	Text	
==	EQ	is equal to
!=	NE	is not equal to
<	LT	is less than
<=	LE	is less than or equal to
>	GT	is greater than
>=	GE	is greater than or equal to



You can use either the symbol or the text in a comparison.

For example,

```
BOOL C = (A == B)
```

is the same as:

```
BOOL C = (A EQ B)
```

C is assigned **1** (true) if **A** equals **B** and . If **A** does not equal **B**, then **C** is **0** (false).



The operators = and == are different.

The single equal operator, =, assigns the value on the right-hand side to the left-hand side.

The double equals operator, ==, compares two values for equality.

If you compare the values of two strings, you must use the correct capitalisation.

For example, if you want to check that the tool is an end mill, then you must use:

```
Tool.Type == 'end_mill'
```

and not:

```
Tool.Type == 'End_Mill'
```

If you are unsure about the case of a string then you can use one of the inbuilt functions **lcase()** or **ucase()** to test against the lower case (see page 81) or upper case (see page 80) version of the string:

```
lcase(Tool.Type) == 'end_mill'  
ucase(Tool.Type) == 'END_MILL'
```

For example, comparing variables:

```
BOOL bigger = (Tool.Diameter+Thickness  
>=ReferenceToolpath.Tool.Diameter+ReferenceToolpath.Thick  
ness)
```

gives a result of **1** (true) when the **Tool.Diameter + Thickness** is greater than or equal to the **ReferenceToolpath.Tool.Diameter + ReferenceToolpath.Thickness** and a result of **0** (false) otherwise.

Logical operators

Logical operators let you to do more than one comparison at a time. There are four logical operators:

- AND
- OR
- XOR
- NOT



*Remember the result of a comparison is either true or false. When true, the result is **1**; when false, the result is **0**.*

Using the logical operator AND

The result is true (**1**) if all operands are true, otherwise the result is false (**0**).

Operand 1	Operand 2	Operand 1 AND Operand 2
true (1)	true (1)	true (1)
true (1)	false (0)	false (0)
false (0)	true (1)	false (0)
false (0)	false (0)	false (0)

Using the logical operator OR

The result is true (**1**) if at least one operand is true. If all the operands are false (**0**) the result is false.

Operand 1	Operand 2	Operand 1 OR Operand 2
true (1)	true (1)	true (1)
true (1)	false (0)	true (1)
false (0)	true (1)	true (1)
false (0)	false (0)	false (0)

Using the logical operator XOR

The result is true (1) if exactly one operand is true. If all the operands are false the result is false (0). If more than one operand is true the result is false (0).

Operand 1	Operand 2	Operand 1 XOR Operand 2
true (1)	true (1)	false (0)
true (1)	false (0)	true (1)
false (0)	true (1)	true (1)
false (0)	false (0)	false (0)

Using the logical operator NOT

The result is the inverse of the input.

Operand 1	NOT Operand 1
true (1)	false (0)
false (0)	true (1)

Advance variable options

Scratchpad variables

It is possible to create and manipulate variables in the command line window. These are called scratchpad variables as you can use them to test the results of parameter evaluation without having to write a macro.

For example, to test some code, in the command line window type:

```
STRING Test = Tool.Name
DEACTIVATE TOOL
ACTIVATE TOOL $Test
```

```
PowerMill >
PowerMill > STRING Test = Tool.Name
PowerMill > DEACTIVATE TOOL
PowerMill > ACTIVATE TOOL $Test
PowerMill >
```


To clear the scratchpad, in the command line window type:

```
RESET LOCALVARS
```

If you do not issue the `RESET LOCALVARS` command, the local variable, `Test`, remains defined until you exit from PowerMill.

Using variables and parameters in macro commands

You can substitute the value of a variable or parameter in a command wherever the command expects a number or a string. To do this, prefix the variable or parameter name with a \$.



*The **EDIT PAR** command only accepts `$variable` input when the `$variable` has a numeric value. You cannot use the `$variable` syntax for **STRING** parameters.*

For example, to create a tool with a diameter that is half that of the active tool.

```
// Calculate the new diameter and name of tool
REAL HalfDiam = Tool.Diameter/2
STRING NewName = string(Tool.Type) + " D-" +
string(HalfDiam)

// Create a new tool and make it the active one
COPY TOOL ;
ACTIVATE TOOL #

// Now rename the new tool and edit its diameter
RENAME TOOL ; $NewName
EDIT TOOL $NewName DIAMETER $HalfDiam
```

This creates a tool with half the diameter.

Scope of variables

A variable exists from the time it is declared until the end of the block of code within which it is declared. Blocks of code are macros and control structures (**WHILE**, **DO - WHILE**, **SWITCH**, **IF-ELSEIF-ELSE**, and **FOREACH**).

A variable, with a specific name, can only be defined once within any block of code.

For example,

```
// Define a local variable 'Count'
INT Count = 5
// Define a second local variable 'Count'
INT Count = 2
```

Gives an error since `Count` is defined twice.

However, within an inner block of code you can define another variable with the same name as a variable (already defined) in an outer block:

```
INT Count = 5
IF Count > 0 {
    // Define a new local variable 'Count'
    INT Count = 3
    // Print 3
    PRINT $Count
// The local Count is no longer defined
}
// Print 5
PRINT $Count
```

A variable defined within an inner block of code hides any variable declared in an outer block. This is also important if you use a name for a variable which matches one of PowerMill's parameters. For example, if the toolpath stepover is 5 and in your macro you have:

```
// 'Hide' the global stepover by creating your own
variable
REAL Stepover = 2
// Print Stepover
PRINT $Stepover
```

The value printed is **2** not **5**, and the toolpath stepover value is unchanged. To access the current toolpath's stepover parameter you must use **toolpath.Stepover**.

```
// 'Hide' the global stepover by creating your own
variable
REAL Stepover = 2
// Print 2
PRINT $Stepover
// Print the value of the toolpath's stepover - which is
5
PRINT $toolpath.Stepover
```



*As macro variables cease to exist at the end of a macro or block of code, you should not use a variable defined in a macro within a retained expression. You can use assignments, as the value is computed immediately. Do not use a macro variable in an **EDIT PAR** expression without **EVAL** as this causes an expression error when PowerMill tries to evaluate it.*

```
REAL Factor = 0.6
// The next two commands are OK as the expression is
evaluated immediately.
$Stepover = Tool.Diameter * Factor
EDIT PAR "Stepover" EVAL "Tool.Diameter * Factor"
```

```
// The next command is not OK because the expression is
retained.
EDIT PAR "Stepover" "Tool.Diameter * Factor"
```

The `Factor` variable ceases to exist at the end of the macro, so `Stepover` evaluates as an error.

Using expressions in macros

An arithmetic expression is a list of variables and values with operators which define a value. Typical usage is in variable declarations and assignments.

```
// Variable declarations
REAL factor = 0.6
REAL value = Tolerance * factor

// Assignments
$Stepover = Tool.Diameter * factor
$factor = 0.75
```



*When using an assignment you **MUST** prefix the variable name with a \$. So PowerMill can disambiguate an assignment from other words in the macro command language.*



In assignments, the expression is always evaluated and the resulting value assigned to the variable on the left of the = operand.

In addition to using expressions in calculations, you can use logical expressions to make decisions in macros. The decision making statements in PowerMill are **IF-ELSE_IF** (see page 56), **SWITCH** (see page 58), **WHILE** (see page 64), and **DO-WHILE** (see page 65). These execute the commands within their code blocks if the result of an expression is true (1). For example:

```
IF active(Tool.TipRadiused) {
    // Things to do if a tip radiused tool.
}
IF active(Tool.TipRadiused) AND Tool.Diameter < 5 {
    // Things to do if a tip radiused tool and the diameter
    // is less than 5.
}
```

You can use any expression to decide whether or not a block of code is performed.

Operators for integers and real numbers

The standard arithmetical operators are available for integers and real numbers.

Operator	Description	Examples
+	addition	3+5 evaluates to 8
-	subtraction	5-3 evaluates to 2
*	multiplication	5*3 evaluates to 15
/	division	6/2 evaluates to 3
%	modulus. This is the remainder after two integers are divided	11%3 evaluates to 2
^	to the power of	2^3 is the same as 2*2*2 and evaluates to 8

For a complete list of operators, see the HTML page displayed when you select **Help > Documentation > Parameters > Reference > Functions**.

Operators for strings

The concatenation (+) operator is available for string.

For example "abc"+"xyz" evaluates to abcxyz.

You can use this to build strings from various parts, for example:

```
MESSAGE "The Stepoever value is: " + string(Stepover)
```

Operator precedence

The order in which various parts of an expression are evaluated affects the result. The operator precedence unambiguously determines the order in which sub-expressions are evaluated.

- Multiplication and division are performed before addition and subtraction.

For example, $3 * 4 + 2$ is the same as $2 + 3 * 4$ and gives the answer 14.

- Exponents and roots are performed before multiplication and addition.

For example, $3 + 5 ^ 2$ is the same as $3 + 5^2$ and gives the answer 28.

$-3 ^ 2$ is the same as -3^2 and gives the answer -9.

- Use brackets (parentheses) to avoid confusion.

For example, $2 + 3 * 4$ is the same as $2 + (3 * 4)$ and gives the answer 14.

- Parentheses change the order of precedence as terms inside in parentheses are performed first.

For example, $(2 + 3) * 4$ gives the answer 20.

or, $(3 + 5) ^2$ is the same as $(3 + 5)^2$ and gives the answer 64.

- You must surround the arguments of a function with parentheses.

For example, $y = \text{sqrt}(2)$, $y = \text{tan}(x)$, $y = \text{sin}(x + z)$.

- Relational operators are performed after addition and subtraction.

For example, $a+b \geq c+d$ is the same as $(a+b) \geq (c+d)$.

- Logical operators are performed after relational operators, though parentheses are often added for clarity.

For example:

$5 == 2+3 \text{ OR } 10 \leq 3*3$

is the same as:

$(5 == (2+3)) \text{ OR } (10 \leq (3*3))$

but is normally written as

$(5 == 2+3) \text{ OR } (10 \leq 3*3)$.

Precedence

Order	Operation	Description
1	()	function call, operations grouped in parentheses
2	[]	operations grouped in square brackets
3	+ - !	unary prefix (unary operations have only one operand, such as, !x, -y)
4	cm mm um ft in th	unit conversion
5	^	exponents and roots
6	* / %	multiplication, division, modulo
7	+ -	addition and subtraction
8	< <= > >= (LT, LE, GT, GE)	relational comparisons: less than, less than or equal, greater than, greater than or equal

9	<code>== !=</code> (EQ, NE)	relational comparisons: equals, not equals
10	<code>AND</code>	logical operator AND
11	<code>NOT</code>	logical operator NOT
12	<code>XOR</code>	logical operator XOR
13	<code>OR</code>	logical operator OR
14	<code>,</code>	separation of elements in a list

Examples of precedence:

Expression	Equivalent
<code>a * - 2</code>	<code>a * (- 2)</code>
<code>!x == 0</code>	<code>(!x) == 0</code>
<code>\$a = -b + c * d - e</code>	<code>\$a = ((-b) + (c * d)) - e</code>
<code>\$a = b + c % d - e</code>	<code>\$a = (b + (c % d)) - e</code>
<code>\$x = y == z</code>	<code>\$x = (y == z)</code>
<code>\$x = -t + q * r / c</code>	<code>\$x = ((-t) + ((q * r) / c))</code>
<code>\$x = a % b * c + d</code>	<code>\$x = (((a % b) * c) + d)</code>
<code>\$a = b <= c d != e</code>	<code>\$a = ((b <= c) (d != e))</code>
<code>\$a = !b c & d</code>	<code>\$a = ((!b) (c & d))</code>
<code>\$a = b mm * c in + d</code>	<code>\$a = (((b mm) * (c in)) + d)</code>

Executing a macro string variable as a command using **DOCOMMAND**

The macro command, **DOCOMMAND**, executes a macro string variable as a command. This enables you to construct a command from string variables and then have that command run as a macro statement. Suppose you have a function to create a copy of a boundary and then fit arcs to the copy:

```
FUNCTION CopyAndArcfit(ENTITY Ent) {
  STRING $NewName = new_entity_name('boundary')
  COPY BOUNDARY $Ent
  EDIT BOUNDARY $NewName ARCFIT 0.01
}
```

If you then want to use the same function to copy a pattern and then fit arcs to the copy. You can replace all instances of 'boundary' with 'pattern' when you give the function a pattern entity. Unfortunately you cannot do this by using variables directly because the PowerMill command syntax does not allow variable substitution in network **KEYWORDS** for example you cannot use `$Type` like this:

```
COPY $Type $Ent
```

However, you can build the command as a string and then use **DOCOMMAND** to execute the resulting string as a command:

```
FUNCTION CopyAndArcFit(Entity Ent) {  
    STRING $NewName = new_entity_name(Ent.RootType)  
    STRING Cmd = "COPY "+ Ent.RootType + " " + Ent.name  
    DOCOMMAND $Cmd  
    $Cmd = "EDIT " + Ent.RootType + " " + Newname + "  
    ARCFIT 0.01"  
    DOCOMMAND $Cmd  
}
```

You can use this technique whenever you find that a variable value cannot be used in a particular point in a command.



Use this technique with caution as it can make your macros harder to understand.

Macro functions

When you run a macro you can use arguments, such as the name of a toolpath, tool, or a tolerance. You must structure the macro to accept arguments by creating a **FUNCTION** called **Main** (see page 52) then specify the arguments and their type.

For example, a macro to polygonise a boundary to a specified tolerance is:

```
FUNCTION Main(REAL tol) {  
    EDIT BOUNDARY ; SMASH $tol  
}
```

A macro to set the diameter of a named tool is:

```
FUNCTION Main(  
    STRING name  
    REAL diam  
)  
{  
    EDIT TOOL $name DIAMETER $dia  
}
```

To run these macros with arguments add the arguments in the correct order to the end of the **MACRO** command:

```
MACRO MyBoundary.mac 0.5
```

```
MACRO MyTool.mac "ToolName" 6
```

If you use **FUNCTION** in your macro, then all commands must be within a function body. This means that you must have a **FUNCTION Main**, which is automatically called when the macro is run.

```
FUNCTION CleanBoundary(string name) {  
    REAL offset = 1 mm  
    REAL diam = entity('boundary';name).Tool.Diameter  
    // Delete segments smaller than tool diameter  
    EDIT BOUNDARY $name SELECT AREA LT $diam  
    DELETE BOUNDARY $name SELECTED  
    //Offset outwards and inwards to smooth boundary  
    EDIT BOUNDARY $name OFFSET $offset  
    EDIT BOUNDARY $name OFFSET ${-offset}  
}  
FUNCTION Main(string bound) {  
    FOREACH bou IN folder(bound) {  
        CALL CleanBoundary(bou.Name)  
    }  
}
```

Within a function, you can create and use variables that are local to the function, just as you can within a **WHILE** loop. However, a function cannot access any variable that is defined elsewhere in the macro, unless that variable has been passed to the function as an argument.



*In the **CleanBoundary** function, `${-offset}` offset the boundary by a negative offset. When you want to substitute the value of an expression into a PowerMill command rather than the value of a parameter, use the syntax `${expression}`. The expression can contain any valid PowerMill parameter expression including: inbuilt function calls; mathematical, logical, and comparison operators.*



*As this macro requires an argument (the boundary name) you must run this from the command window. To run **Clean_Boundary.mac** macro on the **Cavity** boundary you must type `macro Clean_Boundary "Cavity"` in the command line.*

Main function

If a macro has any functions:

- It must have one, and only one, **FUNCTION** called **Main**.
- The **Main** function must be the first function called.

Function names are not case sensitive: **MAIN**, **main**, and **Main** all refer to the same function.

Running a macro where the **Main** function is called with either the wrong number of arguments or with types of arguments that do not match, causes an error. For example:

```
MACRO MyTool.mac 6 "ToolName"
```

generates an error since the macro expects a string and then a number, but is given a number and then a string.

If you want to repeat a sequence of commands at different points within a macro, you can use a **FUNCTION**.

For example, if you want to remove any small islands that are smaller than the tool diameter and smooth out any minor kinks after a boundary calculation. One solution is to repeat the same commands after each boundary calculation:

```
EDIT BOUNDARY ; SELECT AREA LT Boundary.Tool.Diameter
DELETE BOUNDARY ; SELECTED
EDIT BOUNDARY ; OFFSET "1 mm"
EDIT BOUNDARY ; OFFSET "-1 mm"
```

This is fine if you have a macro that creates one boundary, but if it creates a number of boundaries you end up with a macro with excessive repetition. However by using a **FUNCTION** you can define the sequence once:

```
FUNCTION CleanBoundary(string name) {
    REAL offset = 1 mm
    REAL diam = entity('boundary';name).Tool.Diameter
    // Delete segments smaller than tool diameter
    EDIT BOUNDARY $name SELECT AREA LT $diam
    DELETE BOUNDARY $name SELECTED
    //Offset outwards and inwards to smooth boundary
    EDIT BOUNDARY $name OFFSET $offset
    EDIT BOUNDARY $name OFFSET ${-offset}
}
```

Then call it whenever it is needed:

```
FOREACH bou IN folder('boundary') {
    CALL CleanBoundary(bou.Name)
}
CREATE BOUNDARY Shallow30 SHALLOW
EDIT BOUNDARY Shallow30 CALCULATE
CALL CleanBoundary('Shallow30')
```

Returning values from functions

There are two types of arguments to **FUNCTIONS**:

- Input variables (\$ *Input* arguments). If a parameter is an input then any changes to the parameter inside the function are lost when the function returns. This is the default.

- Output variables (\$ **Output** arguments) retain their value after the function returns.

When you call a function, PowerMill creates temporary copies of all the arguments to the function, these copies are removed when the function returns. However, if the macro contains an **OUTPUT** to an argument, then instead of creating a temporary copy of the variable, it creates an alias for the existing variable. Any changes that you make to the alias directly, change the actual variable.

In the example, the **Test** function has two arguments: **aInput** and **aOutput**. Within the Test function:

- The argument **aInput** is a new temporary variable that only exists within the function, any changes to its value only affect the temporary, and are lost once the function ends.
- The **aOutput** variable is an alias for the variable that was passed in the **CALL** command, any changes to its value are actually changing the value of the variable that was given in the **CALL** command.

```
FUNCTION Test(REAL aInput, OUTPUT REAL aOutput) {
    PRINT $aInput
    $aInput = 5
    PRINT $aOutput
    $aOutput = 0
    PRINT $aOutput
}

FUNCTION Main() {
    REAL Par1 = 2
    REAL Par2 = 1
    CALL Test(Par1, Par2)
    // Prints 2 - value is unchanged
    PRINT $Par1
    // Prints 0 - value has been changed
    PRINT $Par2
}
```

When the **CALL** command is executed in the **MAIN** function:

- 1 PowerMill creates a new **REAL** variable called **aInput**. It is assigned the value of **Par1**, and passed into **Test**.
- 2 PowerMill passes **Par2** directly into **Test** where it is known as **aOutput**.

Sharing functions between macros

You can share functions between macros by using the **INCLUDE** statement. You can put all your common functions in a file which you then **INCLUDE** within other macros. For example, if you put the **CleanBoundary** function into a file called **common.inc** you could rewrite the macro as:

```
INCLUDE common.inc
FUNCTION Main(input string bound) {
    FOREACH bou IN folder(bound) {
        CALL CleanBoundary(bou.Name)
    }
}
```

To call this macro from PowerMill:

```
// Clean all the boundaries
MACRO Clean 'boundary'
// Clean all the Roughing boundaries
MACRO Clean 'boundary\Roughing'
```

IF statement

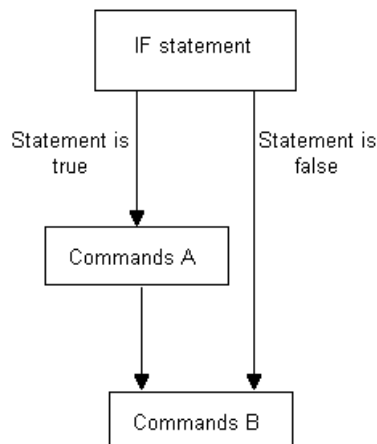
The **IF** statement executes a series of commands when a certain condition is met.

The basic control structure is:

```
IF <expression> {
    Commands A
}
Commands B
```

If **expression** is true then **Commands A** are executed, followed by **Commands B**.

If **expression** is false, then only **Commands B** are executed.



For example, if you want to calculate a toolpath, but do not want to waste time re-calculating a toolpath that has already been calculated:

```
// If the active toolpath has not been calculated, do so
now
IF NOT Computed {
    EDIT TOOLPATH $TpName CALCULATE
}
```

You must enclose **Commands A** in braces, {}, and the braces must be positioned correctly. For example, the following command is NOT valid:

```
IF (radius == 3) PRINT "Invalid radius"
```

To make this command valid, add the braces:

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



*The first brace must be the last item on the line and on the same line as the **IF**.*

The closing brace must be on a line by itself.

IF - ELSE statement

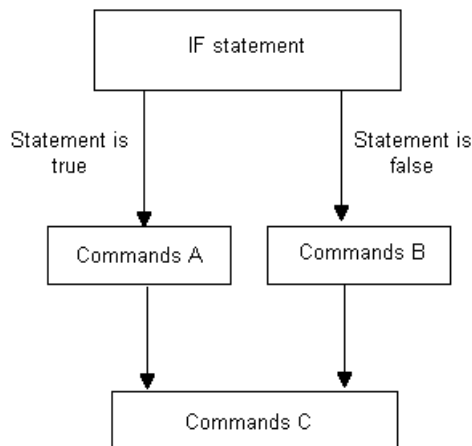
The **IF - ELSE** statement executes a series of commands when a certain condition is met and a different series of commands otherwise.

The basic control structure is:

```
IF <expression> {
    Commands A
} ELSE {
    Commands B
}
Commands C
```

If **expression** is true, then **Commands A** are executed followed by **Commands C**.

If **expression** is false, then **Commands B** are executed followed by **Commands C**.



```

// Set tool axis lead/lean if tip radiused tool
// Otherwise use the vertical tool axis.
IF active(Tool.TipRadius) OR Tool.Type == "ball_nosed" {
    EDIT TOOLAXIS TYPE LEADLEAN
    EDIT TOOLAXIS LEAD "5"
    EDIT TOOLAXIS LEAN "5"
} ELSE {
    EDIT TOOLAXIS TYPE VERTICAL
}
  
```

IF - ELSEIF - ELSE statement

The **IF - ELSEIF - ELSE** statement executes a series of commands when a certain condition is met, a different series of commands when the first condition is not met and the second condition is met and a different series of commands when none of the conditions are met.

The basic control structure is:

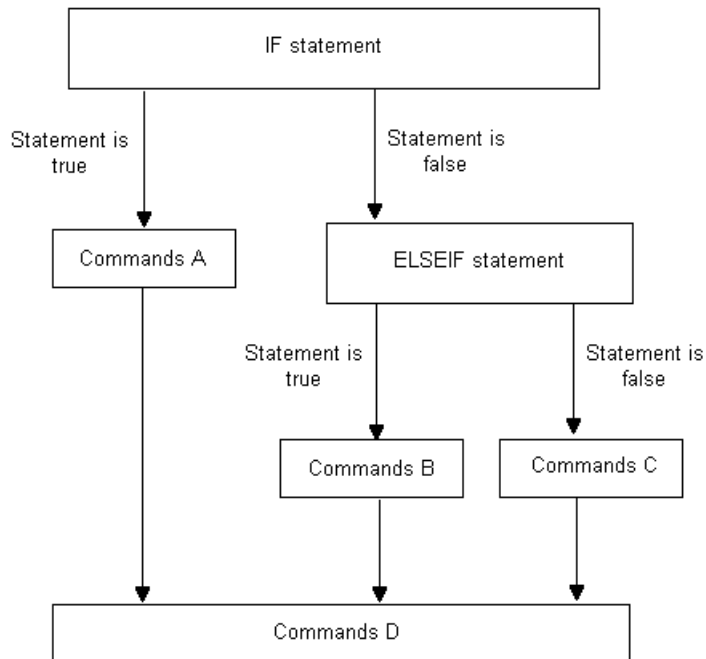
```

IF <expression_1> {
    Commands A
} ELSEIF <expression_2> {
    Commands B
} ELSE {
    Commands C
}
Commands D
  
```

If **expression_1** is true, then **Commands A** are executed followed by **Commands D**.

If **expression_1** is false and **expression_2** is true, then **Commands B** are executed followed by **Commands D**.

If **expression_1** is false and **expression_2** is false, then **Commands C** are executed followed by **Commands D**.



ELSE is an optional statement. There may be any number of **ELSEIF** statements in a block but no more than one **ELSE**.

```

IF Tool.Type == "end_mill" OR Tool.Type == "ball_nosed" {
    $radius = Tool.Diameter/2
} ELSEIF active(Tool.TipRadius) {
    $radius = Tool.TipRadius
} ELSE {
    $radius = 0
    PRINT "Invalid tool type"
}
  
```

This sets the variable **radius** to:

- Half the tool diameter if the tool is an end mill or ball nosed tool.
- The tip radius if the tool is a tip radiused tool.
- Displays **Invalid tool type** if the tool is anything else.

SWITCH statement

When you compare a variable with a number of possible values and each value determines a different outcome, it is advisable to use the **SWITCH** statement.

The **SWITCH** statement enables you to compare a variable against a list of possible values. This comparison determines which commands are executed.

The basic control structure is:

```

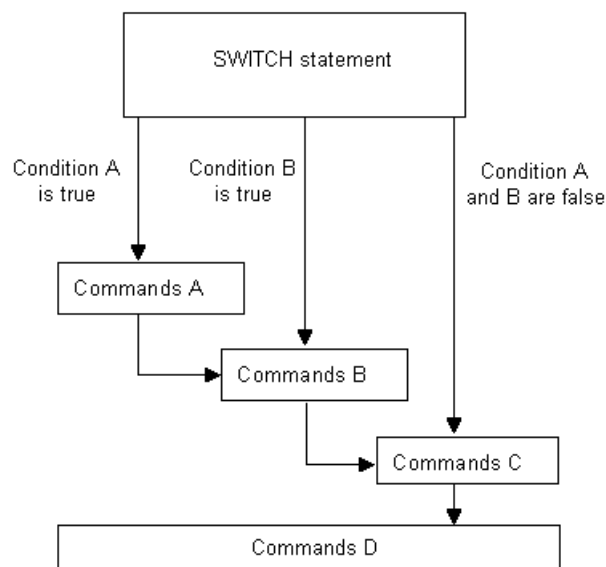
SWITCH variable {
CASE (constant_A)
    Commands A
CASE (constant_B)
    Commands B
DEFAULT
    Commands C
}
Commands D

```

If **condition_A** is true then **Commands A, B, C,** and **D** are executed.

If **condition_B** is true then **Commands B, C,** and **D** are executed.

If **condition_A** and **condition_B** are false then **Commands C,** and **D** are executed.



*When a match is found all the commands in the remaining **CASE** statements are executed. You can prevent this from happening by using a **BREAK** (see page 61) statement.*



*You can have any number of **CASE** statements, but at most one **DEFAULT** statement.*

This example makes changes to the point distribution based on the tool axis type. There are three options:

- 1 3+2-axis toolpaths to have an output point distribution type of **Tolerance and keep arcs** and a lead in and lead out distance of 200.
- 2 3-axis toolpaths to have an output point distribution type of **Tolerance and keep arcs**.
- 3 5-axis toolpaths to have an output point distribution type of **Redistribute**.



Because the **CASE 'direction'** block of code does not have a **BREAK** statement the macro also executes the code in the **'vertical'** block.

```
SWITCH ToolAxis.Type {  
  CASE 'direction'  
    EDIT TOOLPATH LEADS RETRACTDIST  "200.0"  
    EDIT TOOLPATH LEADS APPROACHDIST  "200"  
    // fall through to execute  
  CASE 'vertical'  
    // Redistribute points to tolerance and keep arcs  
    EDIT FILTER TYPE STRIP  
    BREAK  
  DEFAULT  
    // Redistribute points  
    EDIT FILTER TYPE REDISTRIBUTE  
    BREAK  
}
```


Running macros without displaying GUI items

The **NOGUI** statement enables you to use PowerMill modes without displaying GUI items, for example, toolbars, mode-toolbars, dialogs or graphics.

To use **Workplane** macros without displaying GUI items, enter **NOGUI** after **MODE**, for example:

```
MODE NOGUI WORKPLANE_EDIT START "1"
```

To use **Curve editor** macros without displaying GUI items, enter **NOGUI** after **CURVEEDITOR**, for example:

```
EDIT PATTERN ; CURVEEDITOR NOGUI START
```



*As the **NOGUI** command runs the equivalent of **GRAPHICS LOCK** when the mode starts, certain operations involving the selection of pattern segments may not work correctly.*

BREAK statement in a SWITCH statement

The **BREAK** statement exits the **SWITCH** statement.

The basic control structure is:

```
SWITCH variable {  
  CASE (constant_A)  
    Commands A  
    BREAK  
  CASE (constant_B)  
    Commands B  
    BREAK  
  DEFAULT  
    Commands C  
}  
Commands D
```

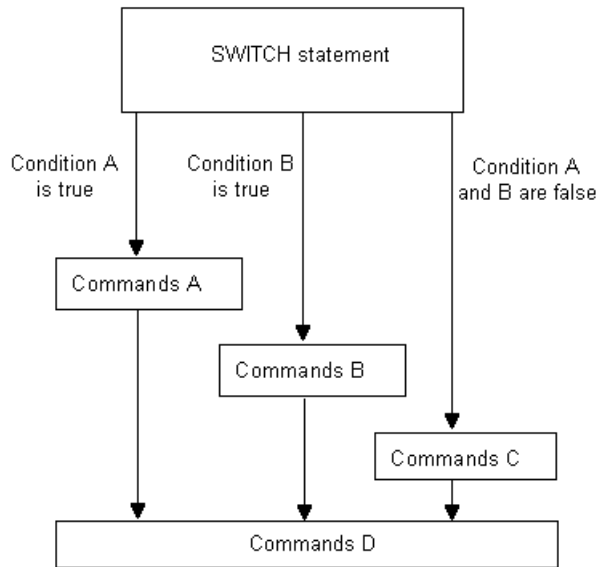
If **condition_A** is true then **Commands A** are executed followed by **Commands D**.



*Remember, if there is no **break** statements then commands **A**, **B**, **C**, and **D** are carried out.*

If **condition_B** is true then **Commands B** are executed followed by **Commands D**.

If **condition_A** and **condition_B** are false then **Commands C** are executed followed by **Commands D**.



Repeating commands in macros

If you want to repeat a set of commands a number of times, for example, creating a circle at the start of every line in the model, you can use loops.

For example, if you have two feature sets, **Top** and **Bottom**, which contain holes you want to drill from the top and bottom of the model respectively, use the macro:

```

STRING Fset = 'Top'
INT Count = 0

```

```

WHILE Count < 2 {
  ACTIVATE FEATURESET $Fset
  ACTIVATE WORKPLANE FROMENTITY FEATURESET $Fset
  IMPORT TEMPLATE ENTITY TOOLPATH "Drilling\Drilling.ptf"
  EDIT TOOLPATH $TpName CALCULATE
  $Fset = 'Bottom'
  $Count = Count + 1
}

```

There are three loop structures:

- **FOREACH (see page 63) loops** repeatedly execute a block of commands for each item in a list.
- **WHILE (see page 64) loops** repeatedly execute a block of commands until its conditional test is false.
- **DO - WHILE (see page 65) loops** executes a block of commands and then checks its conditional test.

FOREACH loop

A **FOREACH loop** repeatedly executes a block of commands for each item in a list or array.

The basic control structure is:

```
FOREACH item IN sequence{  
    Commands A  
}  
Commands B
```

where:

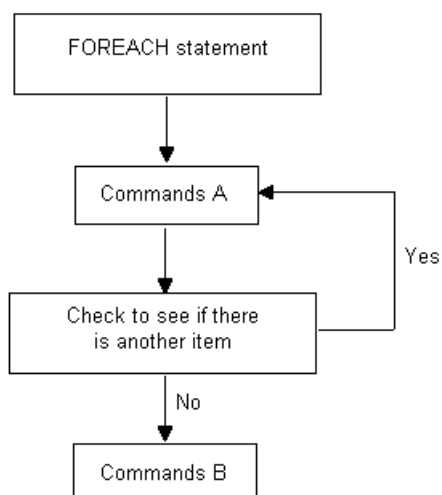
`item` is an automatically created variable that PowerMill initialises for each iteration of the loop;

`sequence` is either a list or an array.

Commands A are executed on the first item in the list.

Commands A are executed on the next item in the list. This step is repeated until there are no more items in the list.

At the end of the list, **Commands B** are executed.



For example,

```
FOREACH item IN folder("path") {  
    Commands A  
}  
Commands B
```

Where <path> is a folder in the Explorer such as, **Toolpath**, **Tool**, **Toolpath\Finishing**.

Within **FOREACH loops**, you can:

- Cancel the loop using the **BREAK** (see page 66) statement.
- Jump directly to the next iteration using the **CONTINUE** (see page 66) statement.

You cannot create your own list variables, there are some built in functions in PowerMill that return lists (see the parameter documentation for component, and folder).

You can use one of the inbuilt functions to get a list of entities, or you can use arrays to create a sequence of strings or numbers to iterate over. For example, use the inbuilt folder function to get a list of entities.

An example of using a **FOREACH loop** is to batch process tool holder profiles:

```
FOREACH ent IN folder('Tool') {  
    ACTIVATE TOOL $ent.Name  
    EDIT TOOL ; UPDATE_TOOLPATHS_PROFILE  
}
```



*The loop variable **ent** is created by the loop and destroyed when the loop ends.*

Another example is to renumber all the tools in a project:

```
INT nmb = 20  
FOREACH t IN folder('Tool') {  
    $t.number.value = nmb  
    $t.number.userdefined = 1  
    $nmb = nmb + 2  
}
```

To get the most out of these macro features, you should familiarise yourself with the inbuilt parameter functions detailed in **Help > Parameters > Reference**.

WHILE loop

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

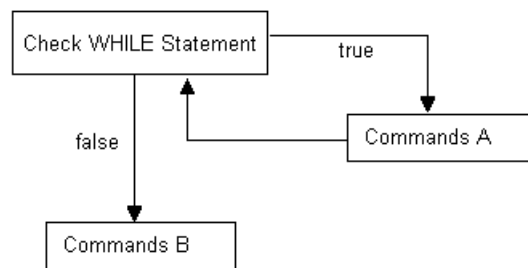
The basic control structure is:

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

If **condition** is true, then **Commands A** are executed.

While **condition** remains true, then **Commands A** are executed.

When **condition** is false, **Commands B** are executed.



Within **WHILE loops**, you can:

- Cancel the loop using the **BREAK** (see page 66) statement.
- Jump directly to the next iteration using the **CONTINUE** (see page 66) statement.

DO - WHILE loop

The **DO - WHILE loop** executes a block of commands and then performs its conditional test, whereas the **WHILE loop** checks its conditional test first to decide whether to execute its commands or not.

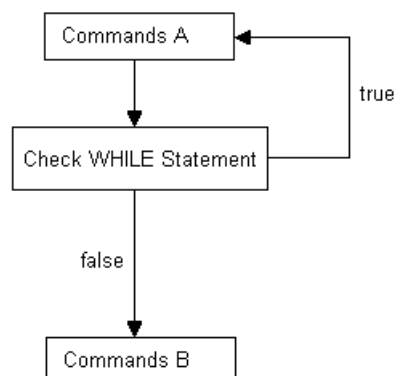
The basic control structure is:

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

Commands A are executed.

While **condition** remains true, then **Commands A** are executed.

When **condition** is false, **Commands B** are executed.



Within **DO - WHILE loops**, you can:

- Cancel the loop using the **BREAK** (see page 66) statement.
- Jump directly to the next iteration using the **CONTINUE** (see page 66) statement.

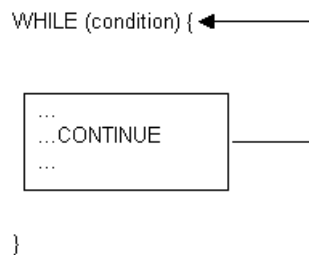
CONTINUE statement

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

This example, calculates and offsets, all unlocked boundaries, outwards and inwards.

```
FOREACH bou IN folder('Boundary') {  
  IF locked(bou) {  
    // This boundary is locked go get the next one  
    CONTINUE  
  }  
  REAL offset = 1 mm  
  EDIT BOUNDARY $bou.Name CALCULATE  
  EDIT BOUNDARY $bou.Name OFFSET $offset  
  EDIT BOUNDARY $bou.Name OFFSET ${-offset} }  
}
```

The **CONTINUE** statement enables the selection of the next boundary.

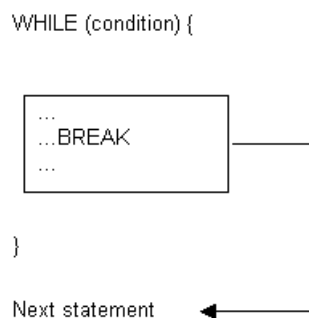


BREAK statement in a WHILE loop

The **BREAK** statement exits the **WHILE** loop.



Nested constructs can require multiple breaks.



Creating sequences

Use the `make_sequence` and `next_in_sequence` functions to create sequences. Sequences contain:

- a starting value. This defines the first element in the sequence.
- an incremental value. This returns the next element in the sequence.
- an optional padding value. This adds extra zeros before the returned elements .

To create a new sequence, use the `make_sequence` function. The basic structure is:

```
OBJECT make_sequence( int start_value, [, int increment,
int padding])
```

For example:

```
OBJECT seq = make_sequence(10, 5, 4)
//Returns a sequence which starts at 10, increments by 5
and is padded by 4 characters:
```

To increment a value in a specified sequence, use the `next_in_sequence` function. The basic structure is:

```
string next_in_sequence(object sequence)
```

For example:

```
OBJECT seq = make_sequence(10, 5, 4)
// Makes a new sequence starting from 10, incremented by
5 and padded to 4 characters
STRING n = next_in_sequence(seq)
// n = "0010"
STRING nn = next_in_sequence(seq)
// nn = "0015"
STRING nnn = next_in_sequence(seq)
// nnn = "0020"
```

RETURN statement

If a macro contains functions, the **RETURN** statement immediately exits the function. If the macro does not contain functions, the **RETURN** statement immediately terminates the current macro. This is useful if an error is detected and you do not want to continue with the remaining commands in the macro.

The basic control structure is:

```
EDIT TOOLPATH $tp.Name CALCULATE
IF NOT Computed {
    // terminate if toolpath did not calculate
    RETURN
}
```

To immediately exit from a function:

```

FUNCTION Calculate(String TpName) {

    IF NOT active(entity('toolpath',TpName).Tool.TipRadius)
    {
        // Error if toolpath does not use a tipradius tool
        PRINT "Toolpath does not have TipRadius tool"
        RETURN
    }

    EDIT TOOLPATH ; CALCULATE
}

FUNCTION Main() {

    FOREACH tp IN folder('Toolpath') {
        ACTIVATE TOOLPATH $tp.Name)
    }
}

```

Terminating macros

The command `MACRO ABORT` immediately terminates the current macro.

The command `MACRO ABORT ALL` terminates all the macros that are currently running. If you call `MACRO ABORT ALL` from within a macro that has been called by another macro, then both macros are terminated.

Printing the value of an expression

To print the value of a scalar expression or parameter use the syntax:

```
PRINT = expression
```

For example, to print the answer to a simple arithmetic expression:

```
PRINT = 2*5
```

When you run the macro, the command window displays the result, 10.

```

PowerMill >
10.0
PowerMill >

```

You can also apply an arithmetic expression to the value of a parameter. For example:

```

EDIT TOOL ; DIAMETER 10
PRINT = Tool.Diameter * 0.6

```

When you run the macro, the command window displays the result, 6.

Constants

PowerMill has a small number of useful constant values that you can use in expressions and macros these include:

REAL PI = 3.141593

REAL E = 2.718282

BOOL TRUE = 1

BOOL FALSE = 0

STRING CRLF = newline

Use these values to make your macros more readable. For example, use **CRLF** constant to build up multi-line messages and prompts:

```
STRING msg = "This is line one."+CRLF+"This is line two."  
MESSAGE INFO $msg
```

Displays the message:

This is line one.

This is line two.

Built-in functions

This section details all the built-in functions that you can use in your macros.

- General mathematical functions (see page 69).
- Trigonometrical functions (see page 70).
- Vector and point functions (see page 70).
- Workplane functions (see page 73).
- String functions (see page 73).
- List creation functions (see page 83).
- Path functions (see page 93) (Folder (see page 94), Directory (see page 94), Base (see page 95), and Project (see page 95) names).
- Conditional functions (see page 97).
- Evaluation functions (see page 97).
- Type conversion functions (see page 99).
- Parameter functions (see page 99).
- Statistical functions (see page 102).

General mathematical functions

The basic structure of the general mathematical functions are:

Description of return value	Function
Exponential	<code>real exp(real a)</code>
Natural logarithm	<code>real ln(real a)</code>
Common (base 10) logarithm	<code>real log(real a)</code>
Square root	<code>real sqrt(numeric a)</code>
Absolute (positive value)	<code>real abs(numeric a)</code>
Returns either -1, 0 or 1 depending on the sign of the value	<code>real sign(numeric a)</code>
Returns either 1 or 0 depending on whether the difference between a and b is less than or equal to tol	<code>real compare(numeric a, numeric b, numeric tol)</code>

Trigonometrical functions

The basic structure of the trigonometrical functions are:

Description of return value	Function
Trigonometric sine	<code>real sin(angle Ø)</code>
Trigonometric cosine	<code>real cos(angle Ø)</code>
Trigonometric tangent	<code>real tan(angle Ø)</code>
Trigonometric arcsine	<code>real asin(real a)</code>
Trigonometric arccosine	<code>real acos(real a)</code>
Trigonometric arctangent	<code>real atan(real a)</code>
Trigonometric arctangent of a/b, quadrant is determined by the sign of the two arguments	<code>real atan2(real a, real b)</code>

Vector and point functions

In PowerMill vectors and points are represented by an array of three reals.

PowerMill contains point and vector parameters, for example the **Workplane.Origin**, **Workplane.ZAxis**, **ToolAxis.Origin**, and **ToolAxis.Direction**. You can create your own vector and point variables:

```
REAL ARRAY VecX[] = {1,0,0}
REAL ARRAY VecY[] = {0,1,0}
REAL ARRAY VecZ[] = {0,0,1}
REAL ARRAY MVecZ[] = {0,0,-1}
```

```
REAL ARRAY Orig[] = {0,0,0}
```

Length

The `length()` function returns the length of a vector.

For example:

```
REAL ARRAY V[] = {3,4,0}
// Prints 5.0
PRINT = length(V)
```

The inbuilt function `unit()` returns a vector that points in the same direction as the input vector, but has a length of 1:

```
PRINT PAR "unit(V)"
// [0] (REAL) 0.6
// [1] (REAL) 0.8
// [2] (REAL) 0.0

// prints 1.0
PRINT = length(unit(V))
```

Parallel

The `parallel()` function returns **TRUE** if two vectors are either parallel (pointing in the same direction) or anti-parallel (pointing in the opposite direction) to each other.

For example:

```
// prints 0
PRINT = parallel(VecX,VecY)
// prints 1
PRINT = parallel(VecX,VecX)
Print = parallel(MVecZ,VecZ)
```

Normal

The `normal()` function returns a vector that is normal to the plane containing its two input vectors. If either vector is zero it returns an error. If the input vectors are either parallel or anti-parallel a vector of zero length is returned.

For example:

```
REAL ARRAY norm = normal(VecX,VecY)
```

Angle

The `angle()` function returns the signed angle in degrees between two vectors, providing that neither vectors have a zero length.

For example:

```
// Prints 90
PRINT = angle(VecX,VecY)
// Prints 90
PRINT = angle(VecY,VecX)
```

The `apparent_angle()` function returns the apparent angle between two vectors when looking along a reference vector. If a vector is parallel or anti-parallel to the reference vector, or if any of the vectors have a zero length it returns an error:

```
// prints 270
print = apparent_angle(VecX,VecY,MVecZ)
// prints 90
print = apparent_angle(VecY,VecX,MVecZ)
```

Setting

The `set_vector()` and `set_point()` functions return the value 1 if the vector or point is set.

For example:

```
REAL ARRAY Vec1[3] = {0,0,1}
REAL ARRAY Vec2[3] = {0,1,0}

// set vec1 to be the same as vec2
BOOL ok = set_vector(vec1,vec2)
// make a X-axis vector
$ok = set_vector(vec2,1,0,0)

REAL X = Block.Limits.XMax
REAL Y = Block.Limits.YMin
REAL Z = Block.Limits.ZMax
ok = set_point(ToolAxis.Origin, X,Y,Z)
```

Unit vector

The `unit()` function returns the unit vector equivalent of the given vector.

For example:

```
REAL ARRAY V[3] = {3,4,5}
PRINT PAR "unit(V)"
BOOL ok = set_vector(V,0,0,6)
PRINT PAR "unit(V)"
```

Workplane functions

You can use the inbuilt function `set_workplane()` to define the origin and axis of a workplane entity. You can call the function:

- with two workplanes, where the values from the second workplane are copied into the first:

```
bool ok =  
set_workplane(Workplane,entity('workplane','3'))
```

which sets the active workplane to have the same values as workplane **3**.

- with a workplane, two vectors, and an origin:

```
REAL ARRAY YAxis[] = {0,1,0}  
REAL ARRAY ZAxis[] = {0,0,1}  
REAL ARRAY Origin = {10,20,30}  
bool ok =  
set_workplane(entity('workplane','reference'), YAxis,  
Zaxis,Origin)
```

String functions

PowerMill parameters and variables can contain strings of characters. There are a number of inbuilt functions that you can use to test and manipulate strings.

The basic structure of string functions are:

Description of return value	Function
Returns the number of characters in the string. For more information see Length function in a string (see page 78).	<code>int length(string str)</code>
Returns the position of the string target from the start of the string str , or -1 if the target string is not found. If you use the optional argument start then scanning begins from that position in the string. For more information see Position function in a string (see page 78).	<code>int position(string str, string target[, numeric start])</code>

Replaces all occurrences of the **target** string with a **replacement** string. The original string is unchanged.

```
string replace( string  
str, string target,  
string replacement)
```

For more information see Replacing one string with another string (see page 79).

Returns part of the string. You can define where the substring starts and its length. The original string is unchanged.

```
string substring( string  
str, int start, int  
length)
```

For more information see Substrings (see page 80).

Returns an upper case string. The original string is unchanged.

```
string ucase( string str)
```

For more information see Upper case function in a string (see page 80).

Returns a lower case string. The original string is unchanged.

```
string lcase( string str)
```

For more information see Lower case function in a string (see page 81).

Returns the string without any leading whitespace.

```
string ltrim( string str)
```

Returns the string without any trailing whitespace.

```
string rtrim( string str)
```

Returns the string without any leading or trailing whitespace.

```
string trim( string str)
```

Splits a string into a list of the strings, separated by whitespace

```
list tokens( string str)
```

The first character of a string is always at index **0**. You can append (add) strings together use the **+** operator. For example:

```
STRING One = "One"  
STRING Two = "Two"  
STRING Three = "Three"
```

```
PRINT = One + ", " + Two + ", " + Three
```

When you run the macro, the command window displays the result, **One, Two, Three**.

```
PowerMILL >
PowerMILL > One, Two, Three
PowerMILL >
PowerMILL >
```

Another way of achieving the same result is:

```
STRING CountToThree = One + ", " + Two + ", " + Three
PRINT = CountToThree
```

When you run the macro, the command window displays the result, **One, Two, Three**.

Date and time functions

The following functions can be used to manipulate the date and time:

Function	Description
<code>time()</code>	The current system time. This is useful for coarse timing in macros and getting the actual time and date.
<code>local_time(int time)</code>	A DateTime object representing the local time given a number of seconds.
<code>utc_time(int time)</code>	A DateTime object representing the time in Coordinated Universal Time.

The DateTime object contains a number of string values, as follows:

String	String value
<code>String year</code>	The year (1900-9999)
<code>String month</code>	The month of the year (01-12)
<code>String day</code>	The day of the month (01-31)

String hour	The hour of the day (00-23)
String minute	The minute of the hour (00-59)
String second	The second of the minute (00-59)
String timestamp	The date and time — the two values in the string are separated by a hyphen (YYYY-mm-dd-HH-MM-SS).



In previous versions of PowerMill you could not create variable of type objects, so you may need to call the `local_time()` or `utc_time()` functions multiple times, like this: `string year = local_time(tm).Year` etc..

Example

The following example shows how to use the `time()` function to measure how long an activity takes:

```
INT old_time = time()
EDIT TOOLPATH ; CALCULATE
INT cumulative_time = time() - old_time
STRING msg = "Toolpath calculation took " +
(cumulative_time) + "secs"
MESSAGE INFO $msg
```

Example

Getting and formatting the current time:

```
INT tm=time()
STRING ARRAY $timestamp[] =
tokens(utc_time($tm).timestamp, "-")
STRING clock =
$timestamp[3] + ":" + $timestamp[4]
$clock = $clock + ":" +
$timestamp[5]
PRINT $clock
```

Returning the home directory

Use the `home()` function to return the pathname of the home directory, including its subdirectories. This can save you time when entering pathnames in macros and user menus.

To use the `home()` function:

```
PRINT PAR $HOME()
STRING H = HOME()
PRINT $H
```


Returning the user ID string

Use the `user_id()` function to return the user ID string. This can save you time when entering pathnames in macros and user menus.

To use the `user_id()` function:

```
PRINT PAR $USER_ID()  
STRING user = USER_ID()  
PRINT $user
```

Returning the path of the running macro

Use `macro_path` and `include_filename` to return the path of the running macro.

If the argument `include_filename` is true, PowerMill prints the macro file name.

If the argument `include_filename` is false, PowerMill does not print the macro file name.

For example, if you run `C:\Macros\macro.mac`:

```
macro_path(0)  
// Returns "C:\Macros"  
macro_path(1)  
// Returns "C:\Macros\macro.mac"
```

Converting a numeric value into a string

The `string` function converts a numeric value into a string value.

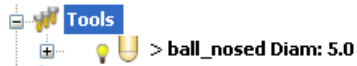
The basic structure is:

```
STRING string( numeric str )
```

This is useful when you want to append a number to a string. For example, to name tools so they contain the type of tool and the diameter, use:

```
CREATE TOOL ; BALLNOSED  
EDIT TOOL ; DIAMETER 5  
STRING TName = string(Tool.Type) + " Diam: " +  
string(Tool.Diameter)  
RENAME TOOL ; $TName  
PRINT = Tool.Name
```

When you run the macro, PowerMill creates a ball nosed tool with a diameter of 5 and gives the tool the name, **ball_nosed Diam: 5.0**.



The command window displays the result, **ball_nosed Diam: 5.0**.

```
PowerMill >  
PowerMill > ball_nosed Diam: 5.0  
PowerMill >
```

Length function in a string

The `length` function returns the number of characters in the string.

The basic structure is:

```
int length( string str )
```

For example:

```
STRING One = "One"  
PRINT = length(One)
```

The command window displays the result, **3**.

```
PowerMILL >  
PowerMILL > 3  
PowerMILL >
```

Another example:

```
STRING One = "One"  
STRING Two = "Two"  
STRING CountToTwo = One + ", " + Two  
PRINT = length(CountToTwo)
```

The command window displays the result, **8**.

```
PowerMILL >  
PowerMILL > 8  
PowerMILL >
```

Another way of producing the same result:

```
PRINT = length(One + ", " + Two )
```

The command window displays the result, **8**.

```
PowerMILL >  
PowerMILL > 8  
PowerMILL >
```

Position function in a string

The `position` string returns the position of the string **target** from the start of the string **str**, or **-1** if the **target** string is not found.

If you use the optional argument **start** then scanning begins from that position in the string.

The basic structure is:

```
int position( string str, string target [, numeric start]  
)
```

For example:

```
PRINT = position("Scooby doo", "oo")
```

The command window displays the result, **2**. PowerMill finds the first instance of **oo** and works out what its position is (**S** is position 0, **c** position 1 and **o** position 2).

```
position("Scooby doo", "oo", 4)
```

The command window displays the result, **8**. PowerMill finds the first instance of **oo** after position 4 and works out what its position is (**b** is position 4, **y** position 5, **"** is position 7 and **o** position 8).

```
position("Scooby doo", "aa")
```

The command window displays the result, **-1** as PowerMill cannot find any instances of **aa**.

You can use this function to check whether a substring exists within another string. For example, if you have a part that contains a cavity and you machined it using various strategies with a coarse tolerance and each of these toolpaths has **CAVITY** in its name. You have toolpaths with names such as, **CAVITY AreaClear**, **CAVITY flats**. To recalculate those toolpath with a finer tolerance use the macro commands:

```
// loop over all the toolpaths
FOREACH tp IN folder('Toolpath') {
    // if toolpath has 'CAVITY' in its name
    IF position(tp.Name, "CAVITY") >= 0 {
        // Invalidate the toolpath
        INVALIDATE TOOLPATH $tp.Name
        $tp.Tolerance = tp.Tolerance/10
    }
}
BATCH PROCESS
```

Replacing one string with another string

The `replace` function replaces all occurrences of the target string with a replacement string. The original string is unchanged.

The basic structure is:

```
string replace( string str, string target, string
replacement)
```

For example:

```
STRING NewName = replace("Scooby doo", "by", "ter")
PRINT = NewName
```

The command window displays the result, Scooter doo.

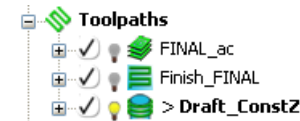
For example, whilst trying different values in the strategy dialogs you add **DRAFT** to the name each toolpath.



When you are satisfied with a particular toolpath you want to change **DRAFT** to **FINAL**. To save yourself from manually editing the toolpath name, you could use a macro to rename the active toolpath:

```
FOREACH tp IN folder('Toolpath') {  
    ACTIVATE TOOLPATH $tp.Name  
    STRING NewName = replace(Name, 'DRAFT', 'FINAL')  
    RENAME TOOLPATH ; $NewName  
}
```

This macro renames the toolpaths to:



*Any instance of **DRAFT** in the toolpath name is changed to **FINAL**. However, the macro is case sensitive, so instances of **Draft** are not changed.*

Alternatively, you could write a macro to rename a toolpath name without activating the toolpath:

```
FOREACH tp IN folder('Toolpath') {  
    STRING NewName = replace(tp.Name, 'DRAFT', 'FINAL')  
    RENAME TOOLPATH $tp.Name $NewName  
}
```

Substrings

The `substring` function returns part of the string. You can define where the substring starts and its length. The original string is unchanged.

The basic structure is:

```
string substring( string str, int start, int length)
```

For example:

```
PRINT = substring("Scooby doo", 2, 4)
```

The command window displays the result, **ooby**.

Upper case function in a string

The `upper case` function converts the string to upper case. The original string is unchanged.

The basic structure is:

```
string ucase( string str)
```

For example:

```
PRINT = ucase("Scooby doo")
```

The command window displays the result, **SCOOBY DOO**.

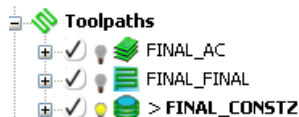
In the Replace one string with another (see page 79) example instances of **DRAFT** are replaced with **FINAL**, but instances of **Draft** are not.



The `ucase` statement replaces instances of **Draft**, **draft**, **dRAft** with **DRAFT**. The rest of the macro replaces **DRAFT** with **FINAL**.

```
FOREACH tp IN folder('Toolpath') {  
    // Get the upper case version of the name  
    STRING UName = ucase(tp.Name)  
    // check if the name contains 'DRAFT'  
    if position(UName, 'DRAFT') >= 0 {  
        // replace DRAFT with FINAL  
        STRING NewName = replace(UName, 'DRAFT', 'FINAL')  
        RENAME TOOLPATH $tp.Name $NewName  
    }  
}
```

This macro renames the toolpaths to:



Previously Draft_ConstZ was not renamed, but it is this time. All the toolpath names are now upper case.

Lower case function in a string

The `lower case` function converts the string to lower case. The original string is unchanged.

The basic structure is:

```
string lcase( string str)
```

For example:

```
PRINT = lcase("SCOOBY DOO")
```

The command window displays the result, **scooby doo**.

In the Replace one string with another (see page 79) example instances of **DRAFT** are replaced with **FINAL**, but instances of **Draft** are not.

In the Upper case function in a string (see page 80) example instances of **Draft**, **draft**, **dRAft** are replaced with **DRAFT**.

The `lcase` statement changes the upper case toolpath names to lower case. It replaces instances of **Draft**, **draft**, **dRAft** are replaced with **draft**.

```
FOREACH tp IN folder('Toolpath') {
    // Get the upper case version of the name
    STRING UName = ucase(tp.Name)
    // check if the name contains 'DRAFT'
    if position(UName, 'DRAFT') >= 0 {
        // replace DRAFT with FINAL
        STRING NewName = replace(UName, 'DRAFT', 'FINAL')
        RENAME TOOLPATH $tp.Name $NewName
    }
    // Get the lower case version of the name
    STRING LName = lcase(tp.Name)
    RENAME TOOLPATH $tp.Name $LName
}
```

This macro renames the toolpaths to:



All the toolpath names are now lower case

Whitespace in a string

Use the following functions to remove whitespace from a string:

- `ltrim()` — Removes leading whitespace.
- `rtrim()` — Removes trailing whitespace.
- `trim()` — Removes leading and trailing whitespace.



The original string is unchanged.

For example:

```
STRING Original = "  What's up Doc!"
STRING Trimmed = ltrim(Original)
print = Original
print = Trimmed
```

Where:

`print = Original` displays " **What's up Doc!**" in the command window.

`print = Trimmed` displays "**What's up Doc!**" in the command window.

Splitting a string

The **tokens()** function splits a string into a list of strings that were separated by the separator characters. By default the separator characters are spaces and tabs.

For example:

```
STRING InputStr = "One Two Three"
STRING LIST Tokens = tokens(InputStr)
FOREACH Tok IN Tokens {
    PRINT = Tok
}
```

You can also give the **tokens()** function an optional extra argument that changes the separator character.

For example:

```
STRING InputStr = "10:20:30:40"
STRING LIST Tokens = tokens(InputStr, ':')
FOREACH Tok IN Tokens {
    PRINT = Tok
}
```

List functions

List functions control the content of a list or array.

The basic structure of list functions are:

Description

Function

Returns the components (see page 84) of another object.

```
list components( entity
entity )
```

Returns a list of specified size containing elements with specified value.

```
list fill (int size, object
value )
```

Returns a list of all the entities in the folder (see page 85).

```
list folder( string folder
)
```

Determines if the list has any content (see page 86).

```
is_empty()
```

Determines if the list contains a specific value (see page 86).

```
member()
```

Adding (see page 87) a list or array to another list or array

```
+
```

Removes duplicate (see page 87) items from a list.

```
remove_duplicates()
```

Creates a list by compiling the contents of two lists (can contain duplicate naming)	<code>set_union()</code>
Creates a list containing items that are present in two lists (see page 88).	<code>intersection()</code>
Creates a list by subtracting (see page 88) from the first list those items that are present in the second list.	<code>subtract()</code>
Returns a sorted list of numerics or strings (see page 92).	<code>list sort(list list)</code>
Returns a sorted list of objects or entities (see page 92) sorted on a field name.	<code>list sort(list list, string field)</code>
Returns a list where the items have been reversed (see page 92).	<code>list reverse(list list)</code>

List components

The inbuilt components function returns the components of another object.



*Currently **NC Program** and **Group** entity parameters are supported.*



The components function returns a list of all items regardless of type. You must check the type of the variable of each item, in the list.

The basic structure is:

```
list components( entity entity )
```

For example if you want to batch process tool holder profiles for the tools in a group that contains toolpaths, boundaries, and tools:

```
FOREACH ent IN components(entity('Group', '1')) {
  IF lcase(ent.RootType) == 'tool' {
    EDIT TOOL $ent.Name UPDATE_TOOLPATHS_PROFILE
  }
}
```


An example, to ensure that all the area clearance toolpaths in an NC program have flood coolant turned on and that mist is set for all the others:

```
FOREACH item IN components(entity('ncprogram','')) {
  // only check nctoolpath items
  IF lcase(item.RootType) == 'nctoolpath' {
    // If the area clearance parameter is active then use
    flood
    IF active(entity('toolpath',item.Name).AreaClearance)
    {
      $item.Coolant.Value = "flood"
    } else {
      $item.Coolant.Value = "mist"
    }
  }
}
```

List fill

The `fill()` function returns a list of a specified size, which contains elements of a specified value.

The basic structure is:

```
list fill(int size, object value)
```

For example, if you wanted to create a list in which `abc` was repeated three times:

```
STRING ARRAY str_arr[] = fill(3, "abc")
// str_arr = {"abc", "abc", "abc"}
```

If you wanted to create a list in which `5` was repeated five times:

```
INT LIST int_ls = fill(5, 5)
// int_ls = {5, 5, 5, 5, 5}
```

List folder

The `folder()` function returns a list of all entities within a folder, including those in subfolders.

The basic structure is:

```
list folder( string folder )
```

The names of the root folders are:

- MachineTool
- NCProgram
- Toolpath
- Tool

- Boundary
- Pattern
- Featureset
- Workplane
- Level
- Model
- StockModel
- Group



*The name of the folder is case sensitive, so you must use **Tool** and not **tool**.*

You can use a **FOREACH** loop to process all of the entities within a folder. For example, to batch process tool holder profiles:

```
FOREACH tool IN folder ('Tool'){
    EDIT TOOL $tool.Name UPDATE_TOOLPATHS_PROFILE
}
```

An example, to batch process all the boundaries in your project:

```
FOREACH bou IN folder('Boundary') {
    EDIT BOUNDARY $bou.Name CALCULATE
}
```

Empty list

The `is_empty()` function queries a list to determine whether it is empty or not.

```
REAL LIST MyList = {}
IF is_empty(MyList) {
    PRINT "Empty List"
}
```

List member

The `member()` function returns **TRUE** if the given value is one of the items in the list. For example, to check that a toolpath does not occur in more than one NC program, you can loop over all NCProgram and check that each toolpath is only seen once. Do this by building a list of toolpath names and checking that each time you add a name you have not already seen it.

```
// Create an empty list
STRING List names = {}
// Cycle through the NC programs
FOREACH ncp IN folder('NCProgram') {
    // loop over the components in the nc program
    FOREACH item IN components(ncp) {
```

```

// Check that it is a toolpath
IF item.RootType == 'nctoolpath' {
  // Use MEMBER to check that we have not seen this
  name before
  IF NOT member(names, item.Name) {
    bool ok = add_last(names, item.Name)
  } else {
    // We have already added this name
    STRING msg = "Toolpath: "+item.Name+crlf+" in
more than one NCProgram"
    MESSAGE ERROR $msg
    MACRO ABORT
  }
}
}
}
}

```

The `is_empty()` function queries a list to determine whether it is empty or not.

```

REAL LIST MyList = {}
IF is_empty(MyList) {
  PRINT "Empty List"
}

```

Adding lists

The `+` function adds a list or array to another list or array. For example, you can add two lists together to get a list of all the tools used by the toolpaths and boundaries:

```

STRING LIST UsedToolNames = ToolpathTools + BoundaryTools

```

Removing duplicate items in a list

The `remove_duplicates()` function removes duplicate values. For example, a tool may be used in both a toolpath and a boundary, so the `UsedToolNames` list may contain duplicate values.

To remove the duplicate values:

```

INT n = remove_duplicates(UsedToolNames)

```

The `set_union()` function returns a list containing the items from both sets, removing any duplicates. So you can create the `UsedToolNames` list using:

```

STRING LIST UsedToolNames = set_union(ToolpathTools,
BoundaryTools)

```

Intersecting items in lists

The inbuilt function `intersection()` returns a list containing the items present in both lists or arrays. To obtain the names of the tools that are used in both toolpaths and boundaries use:

```
STRING LIST TP_Bound_Names = intersection(ToolpathTools,
BoundaryTools)
```

Items present in one list, but not the other

The inbuilt function `subtract()` returns the items that are in the first list, but not in the second list.

```
STRING UnusedToolNames = subtract(AllToolNames,
UsedToolNames)
```

Adding items to a list

You can add items to the start or end of a list.

Adding items to the start of a list

The inbuilt function `add_first(list, item)` adds an item to the start of a list. It returns the number of items in the list after the addition.

For example, to add the name of a pattern to the start of a list:

```
STRING LIST Patterns = {}
FOREACH pat IN folder('Pattern') {
    // Add the name of the pattern to the start of the list
    int s = add_first(Patterns, pat.Name)
}
```

Adding items to the end of a list

The inbuilt function `add_last(list, item)` adds an item to the end of a list. It returns the number of items in the list after the addition.

For example, to add the name of a pattern to the end of a list:

```
STRING LIST Patterns = {}
FOREACH pat IN folder('Pattern') {
    // Add the name of the pattern to the end of the list
    int s = add_last(Patterns, pat.Name)
}
```

Removing items from a list

You can remove items from the start or end of a list.

Removing items from the start of a list

The inbuilt function `remove_first(list)` removes an item from the start of a list. It returns the removed item.

For example, to print the names of patterns in a list:

```
// Print the names of the Patterns in reverse order
// Create a list of the pattern names
STRING LIST Patterns = {}
FOREACH pat IN folder('Pattern') {
    // Add the name of the pattern to start of the list
    int s = add_first(Patterns, pat.Name)
}
// Keep taking the first item from the list until
// there are no more
WHILE size(Patterns) > 0 {
    STRING name = remove_first(Patterns)
    PRINT $Name
}
```

If you have three patterns in the Explorer:



The `FOREACH` loop adds each item to the start of the list. As the `add_first` command adds the next pattern to the start of the list.

So you end up with a list

```
{"Pattern_3", "Pattern_2", "Pattern_1"}.
```

The `WHILE` loop takes the first item in the list, removes it from the list and prints it. This gives:

Pattern_3

Pattern_2

Pattern_1

Removing items from the end of a list

The inbuilt function `remove_last(list)` removes an item to the end of a list. It returns the removed item.

For example, to print the names of patterns in a list:

```
// Print the names of the Patterns in reverse order
// Create a list of the pattern names
STRING LIST Patterns = {}
FOREACH pat IN folder('Pattern') {
    // Add the name of the pattern to end of the list
    int s = add_first(Patterns, pat.Name)
}
// Keep taking the last item from the list until
// there are no more
WHILE size(Patterns) > 0 {
    STRING name = remove_last(Patterns)
    PRINT $Name
}
```

If you have the same three patterns in the Explorer:



The `FOREACH` loop adds each item to the end of the list. As the `add_last` command adds the next pattern to the end of the list. So you end up with a list `{"Pattern_1", "Pattern_2", "Pattern_3"}`.

The `WHILE` loop takes the last item in the list, removes it from the list and prints it. This gives:

Pattern_3

Pattern_2

Pattern_1

To end up with the patterns in the same order as they are in the Explorer either:

- In the `FOREACH` loop use the `add_last` command and in the `WHILE` loop use the `remove_first` command; or
- In the `FOREACH` loop use the `add_first` command and in the `WHILE` loop use the `remove_last` command.

Finding values in a list

The inbuilt function `remove(list, value)` returns true if the value is in the list and false if it is not. If the value is in the list, it also removes the first instance of the value from the list.

For example, to print a list of tool diameters and the number of toolpaths using each tool:

```
// Print a list the tool diameters and the
// number of Toolpaths using each unique diameter.
```

```
REAL LIST Diameters = {}
FOREACH tp IN folder('Toolpath') {
  INT s = add_last(Diameters, tp.Tool.Diameter)
}
// Create a list with just the unique diameters
REAL LIST UniqueD = Diameters
INT n = remove_duplicates(UniqueD)
// Loop over the unique diameters
FOREACH diam = UniqueD {
  // set a counter
  INT Count = 0
  DO {
    $Count = Count + 1
  } WHILE remove(Diameters, diam)
```

```

    STRING Msg = "There are "+Count+" toolpaths using
    "+diam+" tools"
    PRINT $msg
}

```

Extracting data from lists

The inbuilt function `extract(list, par_name)` returns a list containing `par_name` parameters extracted from the input list.

For example, to get the names of all the toolpaths in a project:

```
STRING LIST names = extract(folder('toolpath'),'name')
```

The result could have been achieved with a `FOREACH` loop that builds up the list of names item by item, however, the function allows for a more succinct expression, and it also lets .NET programs to interact with lists without having to use the PowerMill-control-flow statements.



In the above case, the list of toolpaths returned from the inbuilt function `folder()` is directly used as the list argument to `extract`.

Another example is finding the maximum block zheight of the toolpaths:

```
REAL maxz =
max(extract(folder('toolpath'),'Block.Limits.ZMax'))
```

Filtering data from lists

The inbuilt function `filer(list, expression)` returns a sub-list of the original list. The returned list only contains the items in the original list that match the expression you have specified. For example, suppose you want to obtain a list of raster toolpaths:

```
ENTITY LIST rasters = filter(folder('toolpath'),
"strategy=='raster'")
```

Suppose that your toolpaths may contain the UserParameter `'laser'` and you want to change something on just the toolpaths that contain the parameter. You can determine whether a toolpath has the `'laser'` parameter with the expression

`'member(UserParameters._keys,'laser')`. This works because each OBJECT has a special parameter `'_keys'`, which is a list of the immediate sub-parts of the object. So to just get the toolpaths that have the `'laser'` parameter, we can use the following code:

```
// create a string for the
// expression to help readability
STRING expr = "member(UserParameters._keys,'laser')"
ENTITY LIST laser_tps = filter(folder('toolpath'),$expr)
```

The `filter()` function can also be combined with the `extract()` function to build complex expressions within your macros. For example, to obtain the list of tools used by raster toolpaths:

```
ENTITY LIST tools =  
extract(filter(folder('toolpath'), "strategy=='raster'"), '  
tool')
```

A special variable called '`this`' has been added to help with the `filter()` function. The '`this`' variable can be used to refer to the element that the `filter()` function is examining. For example suppose you have a list of numbers and only want the numbers that are greater than 10:

```
REAL LIST numbers = {1.0, 10.2, 3.5, 20.4, 11.0, 2.8}  
REAL LIST numbs = filter(numbers, "this > 10.0")
```

The above returns the list {10.2, 20.4, 11.0}.

Sorted list

The sort function sorts lists and arrays of scalars (numerics or strings) or objects and entities. By default, the functions sort a list in ascending order. If you want to sort the list in descending order, you can apply the reverse function to the list.

If you are sorting a list of objects and entities, you must provide a field name for the sort.

The following examples sort lists of scalar values (numerics and strings):

```
STRING LIST l1 = {"The", "Twelfth", "Night"}  
$l1 = sort(l1)  
// returns the list {"Night", "The", "Twelfth"}  
REAL ARRAY a1 = {23, 12, 4, 52, 32}  
$a1 = sort(a1)  
// Returns the list {4, 12, 23, 32, 52}
```

When sorting non-scalar values, such as entities or objects, you must provide a sort field that is a scalar value:

```
CREATE TOOL ; BALLNOSED  
EDIT TOOL ; DIAMETER 2.0  
CREATE TOOLPATH 'bbb' RASTER  
CREATE TOOL ; BALLNOSED  
EDIT TOOL ; DIAMETER 1.0  
CREATE TOOLPATH 'ccc' RASTER  
CREATE TOOL ; BALLNOSED  
EDIT TOOL ; DIAMETER 1.5  
CREATE TOOLPATH 'aaa' RASTER
```

For example:

```
ENTITY LIST ents = sort(folder('toolpath'), 'name')  
// Returns the list of toolpath {'aaa', 'bbb', 'ccc'}
```



```
ENTITY LIST ents_diam =
sort(folder('toolpath'),'Tool.Diameter')
// Returns the list of toolpath {'ccc','aaa','bbb'}
```

You can reverse the order of a list by using the inbuilt function `reverse()`. The example above sorts the toolpaths based on tool diameter and returns the entries in ascending order, with the smallest diameter listed first. To sort the list in descending order, you can reverse the results.

```
ENTITY LIST ents_diam =
reverse(sort(folder('toolpath'),'Tool.Diameter'))
// Returns the list of toolpaths {'bbb','aaa','ccc'}
```

Path functions

The path functions returns part of the pathname of the entity,

The basic structure of the path functions are:

Description of return value	Function
The Folder name (see page 94) function returns the full folder name of the entity, or an empty string if the entity does not exist.	<code>string pathname(entity ref)</code>
The Folder name (see page 94) function can also be used to return the full folder name of the entity.	<code>string pathname(string type, string name)</code>
The Directory name (see page 94) function returns the directory prefix from the path.	<code>string dirname(string path)</code>
The Base name (see page 95) function returns the non-directory suffix from the path.	<code>string basename(string path)</code>
The Project name (see page 95) functions returns the pathname of the current project on disk.	<code>project_pathname(bool basename)</code>
The Active folder (see page 96) functions returns folder names of active entities.	<code>String active_folder()</code>
The Folder list (see page 96) functions returns the names of folders in the PowerMill project.	<code>String folder_list(folder_name)</code>

Folder name

The `pathname` function returns the full folder name of the entity, or, if the entity does not exist, an empty string.

The basic structure is:

```
string pathname( entity ref )
```

Also,

```
string pathname( string type, string name)
```

Returns the full folder name of the entity.

For example, if you have a **BN 16.0 diam** tool in a **Ballnosed tool** folder, then:

```
pathname('tool', 'BN 16.0 diam')
```

returns the string **Tool\Ballnosed tools\BN 16.0 diam**.



If the entity does not exist it returns an empty string.

You can use this function in conjunction with the **dirname()** (see page 94) function to process all the toolpaths in the same folder as the active toolpath.

```
STRING path = pathname('toolpath',Name)
// check that there is an active toolpath
IF path != '' {
    FOREACH tp IN folder(dirname(path)) {
        ACTIVATE TOOLPATH tp.Name
        EDIT TOOLPATH ; CALCULATE
    }
} ELSE {
    MESSAGE "There is no active toolpath"
    RETURN
}
```

Directory name

The `dirname` function returns the directory prefix from the path.

The basic structure is:

```
string dirname( string path)
```

For example you can use this to obtain the argument for the inbuilt `folder()` function.

```
STRING folder = dirname(pathname('toolpath',Name))
```

Base name

The base name function returns the non-directory suffix from the path.

The basic structure is:

```
string basename( string path)
```

Usually `basename(pathname('tool',tp.Name))` is the same as `tp.Name`, but you can use this in conjunction with **dirname** (see page 94) to split apart the folder names.

For example, suppose your toolpaths are split in folders:

Toolpath\Feature1\AreaClear

Toolpath\Feature1\Rest

Toolpath\Feature1\Finishing

Toolpath\Feature2\AreaClear

Toolpath\Feature2\Rest

Toolpath\Feature2\Finishing

You can rename all your toolpaths so that they contain the feature name and the area clearance, rest, or finishing indicator.

```
FOREACH tp in folder('Toolpath') {  
    // Get the pathname  
    STRING path = pathname(tp)  
    // Get the lowest folder name from the path  
    STRING type = basename(dirname(path))  
    // get the next lowest folder name  
    STRING feat = basename(dirname(dirname(path)))  
    // Get the toolpath name  
    STRING base = basename(path)  
    // Build the new toolpath name  
    STRING NewNamePrefix = feat + "-" + type  
    // Check that the toolpath has not already been renamed  
    IF position(base,NewNamePrefix) < 0 {  
        RENAME TOOLPATH $base ${NewNamePrefix+" " + base}  
    }  
}
```

Project name

The project pathname function returns the pathname of the current project on disk.

The basic structure is:

```
project_pathname(bool basename)
```

The argument `dirname_only` gives a different result if it is true to if it is false.

- If true, returns the name of the project.
- If false returns the full path of the project.

For example if you have opened a project called: **C:\PmillProjects\MyProject**

`project_pathname(0)` returns "**C:\PmillProjects\MyProject**."

`project_pathname(1)` returns **MyProject**.

A PowerMill macro example is:

EDIT BLOCKTYPE TRIANGLES

STRING \$ARBLOCK = project_pathname(0) + '\' + 'block_test.dmt'

GET BLOCK \$ARBLOCK

Active folder name

Use this to determine the folder names of currently active entities, for example the name of the active toolpath or workplane folder.

To display all of the folders in the toolpath branch:

```
STRING LIST MyToolpaths = GET_FOLDERS('toolpath')
```

To display a list of all of the subfolders below a given path:

```
STRING LIST MyFolderToolpaths =  
GET_FOLDERS('toolpath\Rough')
```

To display the name of the active folder:

```
STRING MyFolder = ACTIVE_FOLDER()
```

An empty list is returned if there are no folders, or if there are no active folders.

To find out if the given folder path exists or not, use:

```
document folder_exist()
```

This returns true or false depending on whether the path exists or not. For example, `BOOL ok =`

```
folder_exists('toolpath\areaclearance')
```



Use `document folder_exist()` to interrogate PowerMill Explorer folders. To interrogate folders on disk, use the `dir_exists()` functions.

Stock model states

Use the `PRINT PAR` function to print the parameters of each stock model state. You can use this to display the:

- toolpaths that are applied to the block
- block size
- active state

The basic structure is:

```
PRINT PAR "entity('stockmodel','stockmodel name').States"
```

This can be extended to print specific parameters, for example:

```
PRINT PAR "entity('stockmodel','stockmodel  
name').States[0].Locked"  
PRINT PAR "entity('stockmodel','stockmodel  
name').States[1].References[1]"
```

Conditional functions

The basic structure of conditional functions are:

Description of return value	Function
Returns the value of expression 1 if the conditional expression is true, otherwise it returns the value of expression 2.	<pre>variant select(conditional-expression; expression1;expression2)</pre>



Both expressions must be of the same type.

This example obtains either the tool radius or its tip radius, if it has one.

You can use an IF block of code:

```
REAL Radius = Tool.Diameter/2  
  IF active(Tool.TipRadius) {  
    $Radius = Tool.TipRadius  
  }
```

Or you can use the inbuilt select function:

```
REAL Radius = select(active(Tool.TipRadius),  
Tool.TipRadius, Tool.Diameter/2)
```



*To assign an expression to a parameter, you must use the **select()** function.*

Evaluation functions

The evaluation function evaluate a string argument as an expression.

For example:

```
print = evaluate("5*5")  
prints 25.
```

You can use evaluate to provide a different test at runtime.

This example provides a bubble sort for numeric values. By changing the comparison expression you can get it to sort in ascending or descending order.

```
FUNCTION SortReals(STRING ComparisonExpr, OUTPUT REAL
LIST List) {
    // Get number of items.
    INT Todo = size(List)
    // Set swapped flag before we start
    Bool swapped = 1
    // Repeat for number of items
    WHILE Todo > 1 AND Swapped {
        // start at the beginning
        INT Idx = 0
        // Signal that nothing has been done yet
        $Swapped = 0
        // loop over number of items still to do
        WHILE Idx < Todo-1 {
            // swap if they are out of sequence
            // Uses user supplied comparison function to
            // perform test
            IF evaluate(ComparisonExpr) {
                REAL swap = List[Idx]
                $List[Idx] = List[Idx+1]
                ${List[Idx+1]} = swap
                // signal that we've done something
                $Swapped = 1
            }
            // look at next pair
            $Idx = Idx + 1
        }
        // reduce number of items
        $Todo = Todo - 1
    }
}

FUNCTION Main() {
    /Set up some data
    REAL ARRAY Data[] = {9,10,3,4,1,7,2,8,5,6}
    // Sort in increasing value
    CALL SortReals("List[Idx] > List[Idx+1]", Data)
    PRINT PAR "Data"
    REAL ARRAY Data1[] = {9,10,3,4,1,7,2,8,5,6}
    // Sort in decreasing order
    CALL SortReals("List[Idx] < List[Idx+1]", Data1)
    PRINT PAR "Data1"
}
```

Type conversion functions

The type conversion functions enable you to temporarily convert a variable from one type to another within an expression.

The basic structure of the type conversion functions are:

Description of return value	Function
Convert to integer value.	<code>int int(scalar a)</code>
Convert to real value.	<code>real real(scalar a)</code>
Convert to boolean value.	<code>bool bool(scalar a)</code>
Convert to string value	<code>string string(scalar a)</code>

Normally you would use inbuilt `string()` conversion function to convert a number to a string when building up a message:

```
STRING $Bottles = string(Number) + " green bottles ..."
```

In other cases, you may want convert a real number to an integer, or an integer to a real number:

```
INT a = 2
INT b = 3
REAL z = 0
$z = a/b
PRINT $z
```

This prints **0.0**.

If you want the ratio then you have to convert either **a** or **b** to **real** within the assignment expression.

```
INT a = 2
INT b = 3
REAL z = 0
$z = real(a)/b
PRINT $z
```

This prints **0.666667**.

Parameter functions introduction

All of the PowerMill parameters have an active state which determines whether the parameter is relevant for a particular type of object or operation.

The basic structure of the parameter functions are:

Description of return value	Function
Evaluates the active expression of par .	<code>bool active(par)</code>

Returns whether the parameter can be changed or not.

```
bool locked( par )
```

Returns the number of sub-parameters that **par** contains.

```
int size( par )
```

Returns a list of string descriptions for a enumerator type.

```
string list  
values(par)
```

Returns the parameter one level above in the parameter tree.

```
par parent(par)
```

Evaluate the active expression

For example, the **Boundary.Tool** parameter is not active for a block or sketch type boundaries. You can test whether a parameter is active or not with the inbuilt **active()** function. This can be useful in calculations and decision making.

The basic control structure is:

```
IF active(...) {  
    ...  
}
```

Check if you change a parameter

You can test whether a particular parameter is locked or not with the inbuilt **locked()** function. You cannot normally edit a locked parameter because its entity is being used as a reference by another entity. If you try to edit a locked parameter with the **EDIT PAR** command, PowerMill displays a query dialog asking for permission to make the change. You can suppress this message using the **EDIT PAR NOQUERY** command. The **locked()** function enables you to provide your own user messages and queries that are tailored to your application.

For example:

```
IF locked(Tool.Diameter) {  
    BOOL copy = 0  
    $copy = QUERY "The current tool has been used do you  
    want to make a copy of it?"  
    IF NOT copy {  
        // cannot proceed further so get out  
        RETURN  
    }  
    COPY TOOL ;  
}  
$Tool.Diameter = 5
```


Check the number of sub-paramters

The inbuilt **size()** function returns the number of immediate items in the parameter. You can use this to determine the number of toolpaths in a folder:

```
PRINT = size(folder('Toolpath\Cavity'))
```

Enumerator parameter

The **values()** function returns a list of display names for an enumerator parameter, such as Tool.Type, CutDirection, or Strategy. The names are translated into the current language that a user is working in. This list can be used to gather input from the user with the CHOICE dialog. For example, to ask the user which cut direction they would like to use, you can use the following code:

```
// Get names for the choices the user can make for this
parameter
STRING ARRAY Opts[] = values(CutDirection)

// Get the user input
INT C = INPUT CHOICE $Opts "Choose the Cut Direction you
want"

// Use the returned value to set the direction
$CutDirection = $C
```

Parent parameter

The **parent()** function enables you to access and specify machine tool parameters in a more user friendly way. For example:

```
//To change the opacity of a machine tool/Robot table
with AXIS ADDRESS T, the following syntax can be used
EDIT PAR "parent(machine_axis('T')).ModelList.Opacity"
"25"

//Check the way the 'parent' function works...
CREATE TOOL ; BALLNOSED
EDIT TOOL "1" DIAMETER "20"
real error = 0
$error = ERROR parent(Tool)
print $error

//returns 1.0 as it fails finding the parent of a root -
'TOOL' is the root
$error = ERROR parent(Tool.Diameter)
print $error

//returns 0 as it finds the parent Tool of Tool.Diameter
```

Automate a sequence of edits or actions

Use the following functions to automate a sequence of edits or actions to a number of files and directories:

```
// return list of file and/or directory names
list file_list(<type>, directory, filespec)
// <type> == "all" returns both the files and directories
// <type> == "files" just returns the files
// <type> == "dirs" just returns the directories
// a '+' suffix to the type (eg "files+") will recurse
down the directories

// get the current directory
string pwd()

// check whether a file exists
bool file_exists(path)

// check whether a directory exists
bool dir_exists(path)
```

Statistical functions

The statistical functions enable you to return the minimum and maximum values of any number of numeric arguments.

The basic structure of the statistical functions are:

Description of return value	Function
Returns the largest value in a list of numbers.	<code>real max(list numeric a)</code>
Returns the smallest value in a list of numbers.	<code>real min(list numeric a)</code>

This example finds the maximum and minimum block height of the toolpaths in the active NC program.

```
REAL maxz = -100000
REAL minz = abs(maxz)
FOREACH item IN components(entity('ncprogram','')) {
  IF item.RootType == 'nctoolpath' {
    $maxz = max(maxz,entity('toolpath',item.Name))
    $minz = min(minz,entity('toolpath',item.Name))
  }
}
MESSAGE "Min = " + string(minz) + ", max = " +
string(maxz)
```

Entity based functions

These functions work on specific entities.

Command	Description
<code>entity_exists()</code>	Returns true if the named entity exists (see page 105).
<code>geometry_equal()</code>	Compares two tools, or two workplanes for geometric equivalence.
<code>new_entity_name()</code>	Returns the name (see page 105) assigned to the next entity.
<code>set_workplane()</code>	Sets the vectors and origin of a workplane (see page 73).
<code>segments()</code>	Returns the number of segments in a toolpath, boundary or pattern.
<code>stockmodel_visible_volume()</code>	Returns the stock model volume, based on the stock model drawing option used to show the material.
<code>limits()</code>	Returns the XYZ limits of an entity.
<code>toolpath_cut_limits()</code>	Returns the XYZ limits of the toolpath's cutting moves.

Equivalence

You can use the inbuilt function `geometry_equal()` to test whether two tools, or two workplanes are geometrically equivalent. For a tool the test is based on the cutting geometry of the tool.

Number of segments

The inbuilt function `segments()` returns the number of segments in a pattern or boundary:

```
IF segments(Pattern) == 0 {  
    PRINT "The pattern is empty"  
}
```

To return the number of segments in a toolpath, use:

```
function toolpath_component_count(toolpath,type)
```

For example:

```
print par ${toolpath_component_count('toolpath', '1',  
    'links')}  
print par ${toolpath_component_count('toolpath', '1',  
    'leads')}  
print par ${toolpath_component_count('toolpath', '1',  
    'segments')}
```

```
// Returns the number of toolpath segments, links and leads moves
in toolpath named '1'
```

Stock model volume

The inbuilt function `stockmodel_visible_volume()` returns the stock model volume, based on the stock model drawing option used to show the material. The drawing options are:

- Show all material
- Show rest material
- Show removed material

For example:

```
real volume =
stockmodel_visible_volume(entity('stockmodel','SM'))
// Assigns the volume of the stock model named "SM" to
variable "volume".
```

Limits

The inbuilt function `limits()` returns an array of six elements containing the XYZ limits of the given entity. The supported entities are: pattern, boundary, toolpath, feature set, or model.

```
REAL ARRAY Lims[] = limits('model','MyModel')
```

The values in the array are:

```
REAL MinX = Lims[0]
REAL MaxX = Lims[1]
REAL MinY = Lims[2]
REAL MaxY = Lims[3]
REAL MinZ = Lims[4]
REAL MaxZ = Lims[5]
```

Toolpath cut limits

The inbuilt function `toolpath_cut_limits()` returns an array of 6 real values that hold the XYZ limits of the toolpath's cutting moves. The entity parameter must be a calculated toolpath. The returned array can be used to initialize, or assign to, another array or list.



The string parameter must be the name of a calculated toolpath, otherwise an error will be returned.

```
REAL ARRAY lims[] = toolpath_cut_limits(Toolpath)
$lims = toolpath_cut_limits('my toolpath')
```

The values in the array are:

```
array[0] = Minimum X value.
array[1] = Maximum X value.
array[2] = Minimum Y value.
```

```
array[3] = Maximum Y value.  
array[4] = Minimum Z value.  
array[5] = Maximum Z value
```

Does an entity exist?

The inbuilt function `entity_exists()` returns true if the entity exists. You can call this function with:

- an entity parameter such as `entity_exists(Boundary)`, `entity_exists(ReferenceTool)`, or `entity_exists(entity('toolpath',''))`.
- two parameters that specify the entity type and name such as `entity_exists('tool','MyTool')`.

For example:

```
IF entity_exists(Workplane) {  
    PRINT "An active workplane exists"  
} ELSE {  
    PRINT "No active workplane using world coordinate  
system."  
}  
  
IF NOT entity_exists(entity('toolpath','')) {  
    PRINT "Please activate a toolpath and try again."  
    MACRO ABORT ALL  
}
```

New entity name

The inbuilt function `new_entity_name()` returns the next name that PowerMill gives to a new entity of the given type. You can supply an optional basename argument to obtain the name that PowerMill uses when creating a copy or clone of an entity.

This example shows you how to determine the name of a new entity.

```
CREATE WORKPLANE 1  
CREATE WORKPLANE 2  
CREATE WORKPLANE 3  
  
// Prints 4  
PRINT = new_entity_name('workplane')  
  
DELETE WORKPLANE 2  
  
// Prints 2  
PRINT = new_entity_name('workplane')
```

```
CREATE WORKPLANE ;

// Prints 2_1
PRINT = new_entity_name('workplane', '2')
```

Improving entity-specific macros

You can use parameters to write macros that instruct PowerMill to execute specific lines of code before and after it processes the first and last entity in a loop. The parameters enable PowerMill to:

- identify the last selected entity; and
- display the total number of selected entities and the entity PowerMill is currently processing.

The parameters are:

- `powermill.Status.MultipleSelection.Last`
This enables PowerMill to identify the last selected entity.
- `powermill.Status.MultipleSelection.Count`
This enables PowerMill to display the entity it is processing, for example, 'Checking toolpath 6 for collisions'.
- `powermill.Status.MultipleSelection.Total == 0`
This enables PowerMill to display the total number of selected entities.
- `powermill.Status.MultipleSelection.First`
This enables PowerMill to identify the first entity in a loop.

Creating an entity-specific macro

The example shows how to create a macro that uses a user-defined clearance value to collision check the select toolpaths. The macro:

- asks the user to enter the holder clearance PowerMill uses to collision check the toolpaths.
- displays the name of the toolpath it is processing.
- displays a message when collision checking is complete.

```
Function Main(
STRING $Selected_Toolpath
)
{
    // Create new project parameter to store the holder
    clearance
    BOOL $chk = 0
    $chk = ERROR $project.clearance
```

```

if $chk {
    // Project variable does not exist. Create it and set
    // it to 5 degrees
    EDIT PAR CREATE REAL "clearance"
}

// Before checking the first toolpath, PowerMILL should
// ask the user to enter the holder clearance
❶ IF ($powermill.Status.MultipleSelection.First) OR
$powermill.Status.MultipleSelection.Total == 0 {
    ❷ $project.clearance = INPUT ${"Enter the holder
    clearance PowerMILL uses when checking the " +
    $powermill.Status.MultipleSelection.Total + "
    selected toolpaths for collisions "}
}

// Now collision check toolpath with entered clearance
// Set the clearance:
EDIT COLLISION HOLDER_CLEARANCE $project.clearance
// Now check the toolpath for collisions
EDIT COLLISION TYPE COLLISION
PRINT = "Processing toolpath " +
$powermill.Status.MultipleSelection.Count
EDIT COLLISION APPLY

// Tell user all selected toolpaths have been checked
IF ($powermill.Status.MultipleSelection.Last) {
    MESSAGE INFO "All toolpaths have been checked "
}
}

```



Enter line ❶ and line ❷ without line breaks. The lines only appear to include line breaks because of the limited page width.

Model hierarchy

Model Component Functions

```

INT select_components(DATA components)
INT deselect_components(DATA components)

```

These functions select or deselect all of the components in the passed-in data parameter. The data parameter must store a ModelCompList or ModelCompSet. The return value is numeric, but carries no information.

```
INT number_selected(DATA components)
```

This function returns the number of the components in the passed-in data parameter that are currently selected. The data parameter must store a ModelCompList or ModelCompSet.

Model Hierarchies

Some CAD systems store models in a hierarchical structure. When PowerMILL reads these CAD files it creates a parameterised representation of this structure. This structure can be navigated as a tree, and there are two helper functions, one to retrieve a node from the hierarchy by its path, and the other to retrieve the hierarchy (or a subsection of the hierarchy) as a list that can be filtered or iterated over.

Nodes

The hierarchy of a model is made up of nodes. These are maps with typename "ModelHierarchyElement". They have the following properties:

Property	Type	Description
Name	STRING	The name associated with the node in the hierarchy. The root node's name will be the same as the model's name.
Path	STRING	The path to the node. It consists of the names of the node's ancestors, starting with the root node, separated by backslashes. It includes the node's name.
Parent	MAP (typename: ModelHierarchyElement)	The parent of this node in the hierarchy. The root node's Parent is always an error. For all other nodes, it will be a map of this type.
Children	ARRAY of MAPs (typename: ModelHierarchyElement)	A list of the children of the node in the hierarchy. Each child node is a map of this type.
Components	DATA {ModelCompList}	A list of the model components associated with the node.
Parameters	MAP (typename: ModelMetadata)	Key-Value pairs associated with the node.

**SelectedIn
GUI**

BOOL

This parameter is not currently used, and will always return false.

The root node of a model's hierarchy is accessible through the "Hierarchy" property on the model entity parameter.

Walking the hierarchy

If you want to select all components associated with nodes containing "Hole" in their name, for instance, you could use a macro like this:

```
FUNCTION Main() {  
    ENTITY mod = entity("model", "1")  
    CALL SelectHoles(mod.Hierarchy)  
}  
  
FUNCTION SelectHoles(OBJECT node) {  
    // Select the components associated with this node if  
    // its name contains "Hole"  
    IF (position(node.Name, "Hole") > -1) {  
        INT i = select_components(node.Components)  
    }  
    // Recursively call this function with each child node  
    FOREACH child IN node.Children {  
        CALL SelectHoles(child)  
    }  
}
```

This is a basic template for working with a hierarchy: a function that takes a node as an argument, does something with it, and then recursively calls the function with each of its child nodes.

This template can be built on to give more complex functionality. For example, the operation on the node could depend on extra passed-in arguments, several operations could be performed on the node, or a conditional check on each child node could be placed within the FOREACH loop to skip certain branches of the tree.

Getting a Node by its Path

```
OBJECT model_tree_node(ENTITY model[, STRING path])  
OBJECT model_tree_node(STRING model_name[, STRING path])
```

The first argument should be a model entity or the name of a model entity. The second argument is an optional path into that model's hierarchy. The function returns the node with the given path or the root node if the path is omitted.

The following example gets the node "group1", which is a child of the "part" node, which is a child of the root node "1". It then stores how many of the components associated with the node are currently selected:

```
OBJECT node = model_tree_node("1", "1\part\group1") {  
  INT count = number_selected (node.Components)
```

Getting the Hierarchy as a List

```
OBJECT LIST model_tree_nodes(ENTITY model[, STRING path)  
OBJECT LIST model_tree_nodes(STRING model_name[, STRING  
path])
```

This function takes the same arguments as `model_tree_node()`. It returns a list containing the node that would be returned by `model_tree_node()` if it were sent the same arguments, and all of its descendants.

The example below performs the same operation as the macro in the "Walking the Hierarchy" section above, selecting all geometry associated with nodes that contain "Hole" in their name.

```
FOREACH node IN model_tree_nodes(entity("model", "1")) {  
  IF position(node.Name, "Hole") > -1 {  
    BOOL b = select_components(node.Components)  
  }  
}
```

As well as being a more concise method, for operations that are to be performed on every node in the hierarchy, this will generally execute quicker than walking the hierarchy using the recursive method.

Model metadata

Exchange can extract the metadata from 3rd party CAD files and communicate them to PowerMill during the translation process. Metadata is accessible through parameters which are associated with groups, workplanes and geometries. These are used to store information relevant to the machining of an item.

Metadata on geometry

The `components()` parameter function is extended to include a data parameter which stores a list of model components, including surfaces, and returns a list of the components in a parameterised form.

The parameterized components are objects with the following properties:

Name — A string that contains the name of the component.

Model — A string that contains the name of the model containing the component.

Parameters — An object which stores the key-value pairs associated with the component.

Metadata on workplanes

There are parameters to represent workplanes in the hierarchy. These are maps with the following properties:

Name — The name of the workplane.

Origin — The origin of the workplane. This is an array of 3 REALs.

XAxis — The X axis of the workplane. This is an array of 3 REALs.

YAxis — The Y axis of the workplane. This is an array of 3 REALs.

ZAxis — The Z axis of the workplane. This is an array of 3 REALs.

Parameters — A map of the metadata associated with the workplane.

If a workplane in an imported model has the same name as an existing workplane in PowerMill but a different origin or orientation, the parameter will store the original name.



If models are exported, all of their metadata is lost. However, if you have a model in a project, it retains its metadata.

Feature Parameters

You can use the `inbuilt components()` function to loop over the features within a feature set. Each feature has the following parameters:

Name — Name of Feature.

ID — Id of Feature.

RootType — 'feature' as a string.

num_items — Number of sub-holes.

Type — Type of feature (pocket, slot, hole, boss).

Top — Top of feature, z-value relative to Origin.

Bottom — Bottom of feature, z-value relative to Origin.

Depth — Depth of feature, from top to bottom.

Diameter — Diameter of feature.

Draft — Draft angle.

Axis — Z axis of Feature.

For example:

```
// Print out the diameter of each hole
FOREACH f in components(entity('featureset','1')) {
  IF f.Type == "hole" {
    PRINT = f.name + " has a diameter of " + f.Diameter
  }
}
```

You can also use the `components()` function to iterate over the elements of compound holes, as follows:

```
ENTITY fset = entity('FeatureSet','')
PRINT = "Feature Set '" + fset.Name + "' has " +
fset.num_items + "
Features"
FOREACH feat IN components(fset) {
  IF feat.num_items > 0 {
    PRINT = "Feature '" + feat.Name + "' is a compound
hole'"
    FOREACH sub IN components(feat) {
      PRINT = "Sub-hole '" + sub.Name + "' has diameter "
+ sub.Diameter
    }
  } ELSE {
    PRINT = "Feature '" + feat.Name + "' is a " + feat.Type
  }
}
```



You cannot edit feature parameters in the macro language. You must use the normal PowerMill commands to edit features. However, the parameters will give you all the values you need to make the appropriate edits.

Working with files and directories

PowerMill contains a number of commands and functions for creating and manipulating files on disc. The following commands can be used to delete and copy files and directories:

```
DELETE FILE <filename>
```

```
DELETE DIRECTORY <directory>
```

```
COPY FILE <source-file> <destination file>
```

```
COPY DIRECTORY <source directory> <destination-directory>
```

The command `CD` changes the working directory:

```
// change working directory to "C:\temp"
CD "C:\temp"
```

```
// change working directory back to where PowerMill
started from
CD
```

The command `MKDIR` will create a directory path:

```
MKDIR "C:\temp\output\pm_project"
```

The command will create all directories in the path if they do not exist already.

File and directory functions

PowerMill contains a number of parameter functions that can be used to examine the file structure of the disc:

- `string pwd()` — Returns the current working directory path.
- `bool file_exists(filepath)` — Returns true if filepath is an existing file.
- `bool dir_exists(dirpath)` — Returns true if dirpath is an existing directory.
- `list list_files(string flags,string directory[, string filetype])` — Returns a list of files that match the flags and optional filetype. The flags parameter can be either 'all', 'files', or 'dirs' with an additional '+' suffix. If the '+' suffix is given then all subdirectories are listed.

Example

```
// get a list of all the files in the working directory
STRING LIST files = list_files("files",pwd())

// get all the stl files in the C:\temp directory
$files = list_files("files","c:\temp",".stl")

// get all the directories and subdirectories in the
working directory
$files = list_files("dirs+", pwd())
```

File reading and writing in macros

PowerMill has a number of commands that can be used to read information from a file, or to write information to a file.

Use the following commands:

- `FILE OPEN` — Open a file for reading or writing and associate it with a file handle.

- **FILE CLOSE** — Close a file and free-up its file handle so you can re-use it later.
- **FILE WRITE** — Write the contents of an existing variable to an open file.
- **FILE READ** — Read values from one or more lines from an open file into an existing variable.



A file handle is the name used internally to refer to the file.

FILE OPEN command

Before you can use a file, it must be opened for either reading or writing, and given an internal name (file handle) by which you will later refer to it.

The syntax for opening a file is:

```
FILE OPEN <pathname-of-file> FOR <access-type> AS
<handle>
```



*The <access-type> can be **READ**, **WRITE**, or **APPEND**, and <handle> is a short string used to refer to the file.*

For example, to open the file **fred.txt** for writing, use the command:

```
FILE OPEN "d:\my-files\fred.txt" FOR WRITE AS output
```

To open a file for reading you might use the command:

```
FILE OPEN "d:\my-files\fred.txt" FOR READ AS input
```

To open a file and append more information to the end of it, use the command:

```
FILE OPEN "d:\my-files\fred.txt" FOR APPEND AS input
```

You cannot use the same <handle> for more than one file at a time.

FILE CLOSE command

When you have finished with a file it is good practise to close it so that you can reuse the handle and release system resources.

For example:

```
FILE CLOSE output
FILE CLOSE input
```



To reuse a closed file you need to reopen it.

FILE WRITE command

Use the **FILE WRITE** command to write data from a variable to a file that has been opened for writing or appending.

Variables are written line by line. If the variable to be written is a scalar (INT, BOOL, REAL, or STRING) then a single line is written (unless a string containing new lines is written).

If the variable to be written is an array or list then every element from the source variable is written one line at a time to the file. Individual elements can be written using sub-scripts.

The syntax for the command is:

```
FILE WRITE $<variable> TO <handle>
```

For example:

```
FILE OPEN "test.txt" FOR WRITE AS "out"
STRING LIST greeting = {"Hello, ", "World!"}
INT ARRAY errors[5] = {1,2,3,4,5}
```

```
FILE WRITE $greeting TO "out"
FILE WRITE $errors TO "out"
FILE WRITE $PI TO "out"
FILE WRITE $greeting[1] TO "out"
FILE CLOSE "out"
```

```
// Append an error message to a log file
FILE OPEN "errorlog.txt" FOR APPEND AS log_file
INT error = 2
STRING time = "14:57"
STRING date = "July 1st, 2012"
STRING log_entry = "Error (" + error_code + ") occurred
at " + time + " on " + date
FILE WRITE $log_entry TO log_file
FILE CLOSE log_file
```

FILE READ command

The `FILE READ` command is used to read data from a file opened for reading into an existing variable.

If the variable is a scalar then a single line is read and the string is converted to the required variable type using standard conversion rules.

If the variable is an array then one line is read for each element in the array, with values being stored in the array (starting from index 0). If the end of the file is reached before the array is reached, the data in the remaining elements remain unchanged.

If the variable to be read is a list then all remaining lines in the file are read with existing list elements being over-written and the list being extended as necessary. Again, if the number of lines remaining to be read in the file are fewer than the number of elements currently in the list, then data in the remaining elements is unchanged.

For example:

```
FILE OPEN "values.txt" FOR READ AS input
STRING product_name = ""
INT ARRAY vers[2] = {0, 0}
REAL tol = 0.0
STRING LIST rest_of_file = {}

FILE READ $prod FROM input
FILE READ $version FROM input
FILE READ $tol FROM input
FILE READ $rest_of_file FROM input

PRINT ="Tolerance from " + prod + " v" + vers[0] + "." +
vers[1] + ": " + tol
PRINT ="Comments:"
FOREACH line IN rest_of_file {
    PRINT $line
}
```

Frequently asked questions

How do I loop through all the toolpath entities?

The `folder()` function returns all the items that are in the Explorer folders, for example **Machine Tools**, **Toolpaths**, **Tools**, **Boundaries**, **Patterns** etc.. The easiest way to loop through all the items is to use the `FOREACH` statement:

```
FOREACH tp IN folder('Toolpath') {
    PRINT = tp.name
}
```

The `folder` function returns all the items in the specified folder and in any subfolders. You can limit the number of items returned by specifying a specific subfolder to loop through:

```
FOREACH tp IN folder('Toolpath\semi-finishing') {
    PRINT = tp.name
}
```

How do I only loop over the items in a parent folder (and exclude any subfolders)?

As described above, the `folder()` function returns items from a parent folder and any subfolders. If you only want to loop over the items in the parent folder, you need to filter the results. For example, if you have the folder 'Semi-finishing' and the subfolder 'Not used', which contains temporary or alternative toolpaths, then to access the toolpaths only in the 'Semi-finishing' folder, you need to use the `pathname` and `dirname` functions:

```
STRING fld = 'Toolpath\semi-finishing'
```



```

FOREACH tp IN folder($fld) {
    IF dirname(pathname(tp)) == $fld {
        PRINT = tp.name
    }
}

```

You can also achieve the same result as above by using a complex expression for the FOREACH loop. In some cases this may make your code easier to understand and in other cases much harder. In the following example, the results of the folder() function are filtered so the IF statement can be removed.

```

STRING fld = 'Toolpath\semi-finishing'
STRING filt = 'dirname(pathname(this)) == fld'
FOREACH tp IN filter(folder($fld), $filt) {
    PRINT = tp.name
}

```



Note the use of 'this' in the \$filt expression: when used in the filter function, 'this' is an alias for the current list item that is being filtered. In cases where you need to explicitly use the list item, such as the one above, you should refer to it as 'this' in the expression.

How do I loop over the items in the active folder?

The inbuilt function active_folder() returns the name of the folder that is currently active in the Explorer.



Check the correct folder in the Explorer is active.

```

STRING fld = active_folder()
IF fld == "" {
    // No active folder use the root instead
    $fld = 'Boundary'
} ELSEIF position(fld, 'Boundary\') != 0 {
    MESSAGE "Active folder isn't a boundary folder"
    RETURN
}

```

How can I tell if a toolpath has been calculated?

The **Toolpath's** parameter 'Computed' is be true if it has been calculated. Sometimes a toolpath may have been calculated but contain no cutting segments. If this is an issue then you should check the number of segments as well:

```

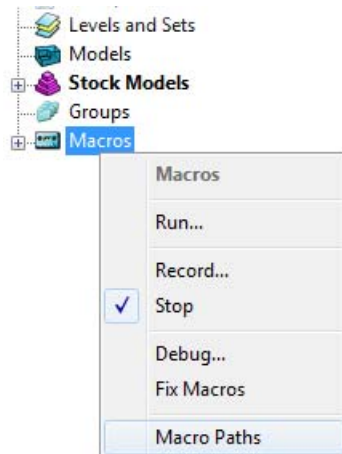
IF tp.Calculated AND segments(tp) > 0 {
    PRINT "Toolpath is calculated and has segments"
}

```

Organising your macros

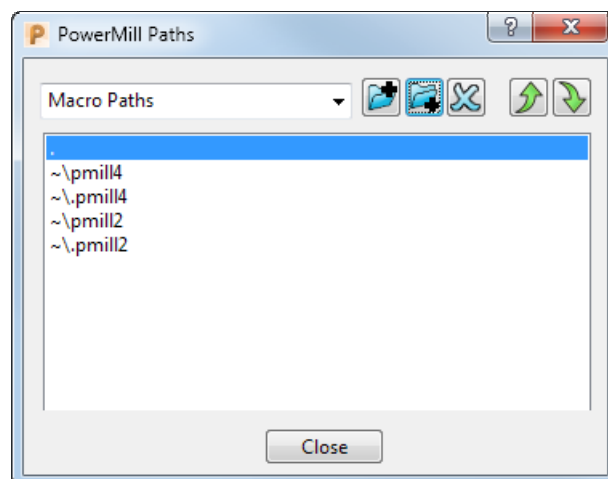
Recorded macros are listed in the Explorer under the **Macros** branch. This example shows you how to manage the macro paths.

- 1 From the **Macros** menu select **Macro Paths**.




*Alternatively, from the **Tools** menu, select **Customise Paths > Macro Paths**.*



The **PowerMill Paths** dialog is displayed showing you all the default macro paths. PowerMill automatically searches for macros located within these folders, and displays them in the Explorer.



The period (.) indicates the path to the local folder (currently, the folder to which the project is saved).

*The tilde (~) indicates your **Home** directory.*

- 2 To create a macro path, click , and use the **Select Path** dialog to select the desired location. The path is added to the top of the list.

- 3 To change the path order, select the path you want to move, and use the  and  buttons to promote or demote the path.
- 4 Click **Close**.
- 5 To load the new paths into PowerMill, expand the **Macros** branch in the Explorer.



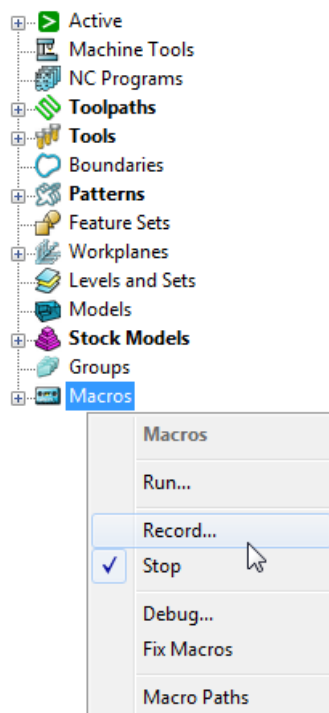
Only the paths that contain at least one macro are shown.

For more information, see [Displaying Macros in the Explorer](#).

Recording the pmuser macro

The **pmuser.mac** is automatically run whenever you start PowerMill providing you with your preferred settings.


- 1 From the **Tools** menu, select **Reset Forms**. This ensures that PowerMill uses the default parameters in the dialogs.
- 2 From the **Macros** context menu, select **Record**.



- 3 Browse to **pmill4** folder in your **Home** area. In the **Select Record Macro File** dialog, enter **pmuser** in the **File name** field, and click **Save**.




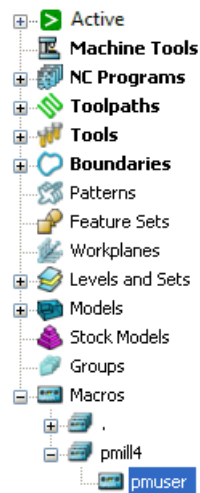
*If you are asked whether you want to overwrite the existing file, select **Yes**.*

The macro icon  changes to red to show recording is in progress.



All dialog options that you want to include in your macro must be selected during its recording. If an option already has the desired value, re-enter it.

- 4 Set up your preferences. For example:
 - a From the **NC Programs** menu, select **Preferences**.
 - b In the **NC Preferences** dialog, select a **Machine Option File** (for example, **heid.opt**).
 - c Click **Open**.
 - d Click **Accept**.
 - e Click  on the **Main** toolbar to display the **Safe area** tab of the **Toolpath connections** dialog.
 - f Enter a **Safe Z** of **10** and **Start Z** of **5**.
 - g Click **Accept**.
- 5 From the **Macros** context menu, select **Stop** to finish recording.
- 6 Expand the **Macros** node. The **pmuser.mac** macro is added under **pmill4**.



- 7 Close and then restart PowerMill to check that the settings from the **pmuser** macro are activated.

Turning off error and warning messages and locking graphic updates

Error and warning messages

PowerMill displays error and warning messages that you must respond to. For example, PowerMill displays an error message if you attempt to activate a toolpath that does not exist.

Normally you should avoid writing a macro that generates error or warning messages, but sometimes it is not possible. In such cases, you can suppress the messages using the following:

```
DIALOGS MESSAGES OFF
```

```
DIALOGS ERROR OFF
```

To turn the error and warning messages back on, type:

```
DIALOGS MESSAGE ON
```

```
DIALOGS ERROR ON
```

Graphics

When you run a macro, PowerMill updates the screen every time a change is made. If PowerMill updates the screen frequently, this amount of screen activity can look unpleasant. Use the following to instruct PowerMill not to update the screen while the commands are in progress, and instead to update the screen (just the once) after the commands are complete.

```
GRAPHICS UNLOCK
```

```
GRAPHICS LOCK
```



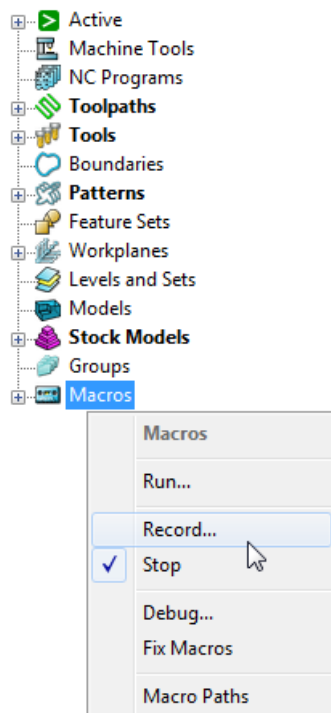
When a macro finishes, PowerMill restores the message and graphic settings to what they were before the macro started. This ensures the messages and graphics are not accidentally turned off permanently.

Recording a macro to set up NC preferences


This example records a macro that sets up NC preferences for Heid400 machine controllers.

- 1 From the **Tools** menu, select **Reset Forms**. This ensures that PowerMill uses the default parameters in the dialogs.

- 2 From the **Macros** context menu, select **Record**.



- 3 Browse to **pmill** folder in your **Home** area in the **Select Record Macro File** dialog, enter **h400_prefs** in the **File Name** field, and click **Save**.

The macro icon  **Macros** changes to red to show recording is in progress.



All dialog options that you want to include in your macro must be selected during its recording. If an option already has the desired value, re-enter it.

- 4 From the **NC Programs** context menu, select **Preferences**.
- 5 In the **NC Preferences** dialog, select the **Heid400.opt** in the **Machine Option File** field on the **Output** tab.
- 6 Click the **Toolpath** tab, and select **Always** in the **Tool Change** field.
- 7 Click **Accept**.
- 8 From the **Macros** context menu, select **Stop** to finish recording.

Tips for programming macros

This section gives tips to help you record macros.

- Macros record any values you explicitly change in a dialog, but do not record the current default values. For example, if the default tolerance is **0.1** mm and you want a tolerance **0.1** mm, you must re-enter **0.1** in the tolerance field during recording. Otherwise PowerMill uses the current tolerance value, which is not necessarily the value you want.
- From the **Tools** menu, select **Reset Forms**. This ensures that PowerMill uses the default parameters in the dialogs.
- When debugging a macro it is important to have the macrofixer turned off. Use the command:

```
UNSET MACROFIX
```

This ensures all syntax and macro errors are reported by PowerMill directly. You can use `SET MACROFIX` to turn it back on.

- If you get a syntax error in a loop (**DO-WHILE**, **WHILE**, **FOREACH**) or a conditional statements (**IF-ELSEIF-ELSE**, **SWITCH**) check you have a space before any opening braces (`{`). For a **DO-WHILE** loop make sure the closing brace (`}`) has a space after it and before the **WHILE** keyword.
- To exit a loop, press **ESC**.
- Your code blocks must have matching braces. They must have the same number of opening braces (`{`) as closing braces (`}`).
- The **ELSEIF** keyword does not have a space between the **IF** and the **ELSE**.
- If you encounter expression errors check you have balanced parentheses, and balanced quotes for strings.
- Decimal points in numbers must use a full stop (.) and not a comma (,).
- The variable on the left of the = sign in assignments must have a \$ prefix. So:

```
$myvar = 5
```

is correct, but:

```
myvar = 5
```

is wrong as it is missing the \$ prefix.

- Local variables override PowerMill parameters. If your macro contains:

```
REAL Stepover = 10
```

then during the execution of the macro any use of **Stepover** uses the value **10** regardless of what the value specified in the user interface. Also the command:

```
EDIT PAR "Stepover" "Tool.Diameter*0.6"
```

changes the value of this local **Stepover** variable NOT the PowerMill **Stepover** parameter.

Index

A

- Active folder name • 95
- Adding items to a list • 35, 87
- Adding lists • 86
- Adding macro loops • 11
- Adding macro variables • 10
- Arguments in macros • 51
 - Function values • 53
 - Running macros with arguments • 12
 - Sharing functions • 54
- Arrays • 33
 - Lists • 33
 - Points • 41
 - Using arrays • 25
 - Vectors • 41
- Automate a sequence of edits or actions • 101

B

- Base name • 94
- Basic macro • 9
 - Adding macro loops • 11
 - Adding macro variables • 10
 - Decision making in macros • 14
 - Returning values from macros • 18
 - Running macros with arguments • 12
 - Using a FOREACH loop • 22
 - Using arrays • 25
 - Using functions in macros • 16
- BREAK statement • 60, 65
- Building a list • 37
- Built-in functions • 68

C

- Calling from other macros • 4
- Carriage returns in dialogs • 30
- Comparing variables • 41, 43
- Components
 - List components • 83
- Compound macros • 4
- Conditional functions • 96
- Constants • 68
 - Euler's number • 68
 - Pi • 68
- Converting numerics to strings • 76
- Creating macros • 1
 - Basic macro • 9
- Creating variables (macros) • 27

D

- Date and time functions • 74
- Decision making in macros • 14
- Decisions in macros
 - BREAK statement • 60, 65
 - IF - ELSE statement • 55
 - IF - ELSEIF - ELSE statement • 56
 - IF command • 54
 - SWITCH statement • 57
- Decrease options available to user • 31
- Delete files or directories • See Working with files and directories
- Dialogs in macros • 28
 - Carriage returns in dialogs • 30
- Directory name • 93
- DO - WHILE loop • 64
 - Decision making in macros • 14

DOCOMMAND • 50

E

- Editing macros • 3
 - Editing • 3
- Empty list • 85
- Enter values into macros • 28
- Entities in macros • 30
- Entity based functions • 102
 - Entity variables • 39
 - Equivalence • 102
 - Limits • 102
 - New entity name • 104
 - Number of segments • 102
 - Workplane origin • 72
- Entity variables • 39
- Equivalence • 102
- Error and warning messages, turn off • 119
- Euler's number • 68
- Evaluation functions • 96
- Example of programming language • 8
- Existing file or directory • See Delete files or directories
- Exiting a function • 66
- Exponential • 68
- Expressions in macros • 46
 - Order of operations • 48
 - Precedence • 48
- Extracting data from lists • 90

F

- Feature parameters • 110
- File name in macros • 32
- Filtering data from a list • 90
- Folder
 - List folder • 84
- Folder name • 93
- FOREACH loop • 62
 - Using a FOREACH loop • 22
- Function values • 53
- Functions in macros • 68
 - Arguments in macros • 51
 - Conditional functions • 96
 - Entity based functions • 102
 - Evaluation functions • 96
 - Exiting a function • 66

- Function values • 53
- Introduction • 98
- List components • 83
- List folder • 84
- List functions • 82
- Main function • 52
- Path functions • 92
- Point functions • 69
 - Setting a point • 69
- Returning function values • 53
- Sharing functions • 54
- Statistical functions • 101
- STRING function • 72
- Type conversion functions • 98
- Using functions in macros • 16
- Using the SWITCH statement • 17
- Vector functions • 69
 - Angle between vectors • 69
 - Length of a vector • 69
 - Normal vectors • 69
 - Parallel vectors • 69
 - Point functions • 69
 - Setting a vector • 69
 - Unit vector • 69

I

- IF - ELSE statement • 55
 - Decision making in macros • 14
- IF - ELSEIF - ELSE statement • 56
- IF command • 54
- Increase options available to user • 31
- Inputting values into macros • 28
 - Entities in macros • 30
 - Options in macros • 31
 - File name in macros • 32
- Intersecting items in lists • 87
- Items in one list • 87

L

- Length of a string • 77
- Limits • 102
- List components • 83
- List folder • 84
- List functions • 82
 - Adding items to a list • 35, 87
 - Adding lists • 86
 - Empty list • 85
 - Extracting data from lists • 90

- Finding values in a list • 87
- Intersecting items in lists • 87
- Items in one list • 87
- List components • 83
- List folder • 84
- List member • 85
- Removing duplicate items • 86
- Removing items from a list • 36, 87
- List member • 85
- Lists • 33
 - Adding items to a list • 35, 87
 - Arrays • 33
 - Building a list • 37
 - Removing items from a list • 36, 87
 - Using lists • 34
- Logarithm • 68
- Loops
 - Adding macro loops • 11
 - Decision making in macros • 14
 - DO - WHILE loop • 64
 - FOREACH loop • 62
 - WHILE loop • 63

M

- Macro comments • 7
- Macro statement • 6
 - Adding macro loops • 11
 - Arguments in macros • 51
 - BREAK statement • 60, 65
 - DO - WHILE loop • 64
 - FOREACH loop • 62
 - IF - ELSE statement • 55
 - IF - ELSEIF - ELSE statement • 56
 - IF command • 54
 - Macro statement • 6
 - RETURN statement • 66
 - SWITCH statement • 57
 - Using the SWITCH statement • 17
 - WHILE loop • 63
- Macro statements
 - Adding macro loops • 11
 - Arguments in macros • 51
 - BREAK statement • 60, 65
 - DO - WHILE loop • 64
 - FOREACH loop • 62
 - IF - ELSE statement • 55
 - IF - ELSEIF - ELSE statement • 56
 - IF command • 54
 - Macro statement • 6

- RETURN statement • 66
- SWITCH statement • 57
- Using a FOREACH loop • 22
- Using the SWITCH statement • 17
- WHILE loop • 63

Macros

- Calling from other macros • 4
- Compound macros • 4
- Creating macros • 1
- Editing macros • 3
- Expressions in macros • 46
- Macro comments • 7
- Macro statement • 6
- NC preference macro • 120
- pmuser macro • 4, 118
- Recording macros • 2, 118, 120
- Repeating commands in macros • 61
- Running macros • 3
- Setting paths • 117
- Variables in macros • 27
- Writing macros • 5

Main function • 52

Mathematical functions • 68

- Exponential • 68
- Logarithm • 68
- Mathematical functions • 68
- Natural logarithm • 68
- Square root • 68

N

- Natural logarithm • 68
- NC preference macro • 120
- New entity name • 104
- Normal vectors • 69
- Number of segments • 102

O

- Object variable • 40
- Operators • 47
 - Logical operators • 43
 - Relational operator • 41
- Order of operations • 48

P

- Parameter functions

- Automate a sequence of edits or actions • 101
- Introduction • 98
- Path functions • 92
 - Active folder name • 95
 - Base name • 94
 - Directory name • 93
 - Folder name • 93
 - Path name • 93
 - Project name • 94
- Path name • 93
- Pausing macros • 26
- Pi • 68
- pmuser macro • 4, 118
- Point functions • 69
 - Setting a point • 69
- Position of a string • 77
- Precedence • 48
- Print
 - Print the values of an expression • 67
- Programming language example • 8
- Project name • 94

R

- Reading a file • 112
- Recording macros • 2, 118, 120
- Relational operator • 41
- Removing duplicate items • 86
- Removing items from a list • 36, 87
- Repeating commands in macros • 61
 - BREAK statement • 60, 65
 - DO - WHILE loop • 64
 - FOREACH loop • 62
 - WHILE loop • 63
- Replacing strings • 78
- RETURN statement • 66
- Returning function values • 53
- Returning values from macros • 18
- Running macros • 3
- Running macros with arguments • 12

S

- Scratchpad variables • 44
- Selecting a file name in macros • 32
- Selecting entities in macros • 30
- Setting paths • 117
- Setting up your working directories • 117

- Sharing functions • 54
- Splitting a string • 82
- Square root • 68
- Statistical functions • 101
- Stopping macros • 66
- STRING function • 72
 - Converting numerics to strings • 76
 - Data and time • 74
 - Length of a string • 77
 - Position of a string • 77
 - Replacing strings • 78
 - Splitting a string • 82
 - String variables • 50
 - Substrings • 77, 79
 - Upper case function • 79
 - Whitespace in a string • 81
- String variables • 50
- Substrings • 77, 79
- SWITCH statement • 57
 - Using the SWITCH statement • 17

T

- Tips for programming macros • 121
- Trouble shooting macros • 121
- Turning off error and warning messages • 119
- Type conversion functions • 98

U

- Upper case function • 79
- Using a FOREACH loop • 22
- Using arrays • 25
- Using functions in macros • 16
- Using lists • 34
- Using the SWITCH statement • 17
- Using variables (macros) • 28

V

- Variable scope (macros) • 45
- Variables in macros • 27
 - Adding macro variables • 10
 - Comparing variables • 41, 43
 - Creating variables (macros) • 27
 - DOCOMMAND • 50
 - Entity variables • 39
 - Logical operators • 43

- Object variables • 40
- Operators • 47
- Order of operations • 48
- Precedence • 48
- Relational operator • 41
- Returning values from macros • 18
- Scratchpad variables • 44
- String variables • 50
- Using variables (macros) • 28
- Variable scope (macros) • 45
- Vector functions • 69
 - Angle between vectors • 69
 - Length of a vector • 69
 - Normal vectors • 69
 - Parallel vectors • 69
 - Point functions • 69
 - Setting a vector • 69
 - Unit vector • 69

W

- Warning and error messages, turn off • 119
- WHILE loop • 63
- Whitespace in a string • 81
- Working with files and directories • 111
- Workplane origin • 72
- Writing information to files • 112
- Writing macros • 5