AVEVA

We'll take you there™

AVEVA™ E3D Design (3.1) PML Macros and Functions

TG1031-CC0-C01

Training Guide

www.aveva.com

This page is intentionally left blank.

Revision Log

DATE	REVISION	DESCRIPTION OF REVISION	AUTHOR	REVIEWED	APPROVED
17/05/22	0.1	Issued for Review	PW	SF	
05/08/22	0.2	Issued for Review	PW	SF	
29/10/22	0.3	Reviewed	PW	SF	
27/11/22	0.4	Issued for Review	PW	SF	
30/11/22	0.5	Reviewed	PW	SF	
07/12/22	1.0	Approved for Training AVEVA™ Design 3.1	PW	SF	ST

Template version: V5.1.5

Updates

Change highlighting will be employed for all revisions. Where new or changed information is presented, section headings will be highlighted in Yellow.

Suggestion / Problems

If you have a suggestion about this manual or the system to which it refers please report it to AVEVA Training & Product Support at CSE@aveva.com

This manual provides documentation relating to products to which you may not have access or which may not be licensed to you. For further information on which products are licensed to you please refer to your licence conditions.

Visit our website at https://www.aveva.com

Disclaimer

Copyright Notice

Trademark Notice

Table of Contents

This page is intentionally left blank.

AVEVA

CHAPTER 1

1 Introduction

This manual is designed to introduce to the AVEVA Plant Programming Macro Language. There is no intention to teach software programming but only provide instruction on how to customise **AVEVA™ E3D Design** using Programmable Macro Language (PML) in AVEVA Plant.

This training guide is supported by the reference manuals available within the products installation folder. References will be made to these manuals throughout the guide.

1.1 Target Audience / Course Type

Target personnel – All Designer and Engineer

Course Type – Dual

Aim

The aim of this training guide is to provide the basic knowledge of the programmable macro language that provide the Macro customization for the **AVEVA™ E3D Design** Model module.

The following points need to be understood by the trainees:

- Understand how PML can be used to customise **AVEVA™ E3D Design**.
- Understand how to create Functions, Forms and Objects.
- Understand how to use the built-in features of **AVEVA™ E3D Design**.
- Understand the use of Add-ins to customise the environment.

Objectives

At the end of this training, the trainees will have a:

- Broad overview of Programmable Macro Language (PML).
- Basic coding practices and conventions.
- How PML can interact with the Design model.
- How Forms and Menus can be defined with PML.

Prerequisites

The participants must have a familiarity with **AVEVA™ E3D Design 3.1** or previous version.

Previous experience of programming is of benefit – but is not necessary.

This course requires the installation of **AVEVA™ Training Setup (EBU) 3.0.6.0** or later.

Course Structure

Training will consist of oral and visual presentations, demonstrations, worked examples and set exercises. Each trainee will be provided with some example files to support this guide. Each workstation will have a training project, populated with model objects. This will be used by the trainees to practice their methods and complete the set exercises.

Using this Guide

Certain text styles are used to indicate special situations throughout this document, here is a summary:

- Menu pull-downs and button click actions are indicated by **bold blue** text
- Information that needs to be entered into the software will be in **bold red** text
- System prompts will be ***bold italic black*** text
- Example files or inputs will be in the `courier new` font.
- Products, Applications, Modules, Toolbars, Explorers and other significant software elements will be in **bold black** text

Other areas in this Training Guide will be presented with italic blue text and an accompanying icon to classify the type of additional information. Other styles include:

Additional Information

Refer to other documentation

Warning

- *Why*

The following icons will be used to identify industry or discipline specific content:

Plant - Content specific to the Plant industry

● *Marine - Content specific to the Marine industry*

Other icons may be used if necessary.

AVEVA

CHAPTER 2

2 PML Overview

Programmable Macro Language (PML) is the customisation language used by **AVEVA™ E3D Design** and it provides a mechanism for users to add their own functionality to the AVEVA software family. This functionality could be as simple as a re-naming macro, or as complex a complete user-defined application (and everything in between).

PML is a coding language specific to AVEVA products based on the command syntax that is used to drive **AVEVA™ E3D Design**. As the product develops, PML is also improved providing new functionality and bringing it closer to other object-orientated programming languages, while still retaining the powerful command syntax. Although it is one language, there are three distinct parts:

- PML 1 the first version of PML based on command syntax. String based and provided IF statement, loops, variables & error handling.
- PML 2 object-oriented language extending the ability of PML. Use of functions, objects and methods to process information (also covered in *TG1031-CC1-C01 AVEVA™ E3D Design (3.1) PML Form Design*).
- PML .NET provides the platform in PML to display and use objects created in other .NET languages (covered in *TG1031-CC1-C01 AVEVA™ E3D Design (3.1) PML Form Design*)

2.1 PML 1 – String Based Command Syntax

When PML is written as command syntax, AVEVA™ E3D Design processes it as individual lines of command and runs them in sequence - as if they had been typed directly into the command window.

A simple macro is likely to be written completely in command syntax and allows users to re-run popular commands. Saved as an ASCII file, the macro can be run in AVEVA™ E3D Design through the command window (by entering **$m/FILENAME** or by dragging and dropping the file, where filename contain full path of the file).

2.1.1 Example of Simple Command Syntax Macro

The following is an example of a simple macro that can be used to create an Equipment element. As each line is run in order, the same piece of equipment will be created each time the macro is run.

```
NEW EQUIP /ABCD
NEW BOX XLEN 300 YLEN 400 ZLEN 600
NEW CYL DIA 400 HEI 600
CONN P1 TO P2 OF PREV
```

Macros can be extended to include IF statements, DO loops, variables and error handling (explained further later in the course). If additional information is enter onto the same line as the call for the macro, then these become input parameters and are available for use within the macro.

For example, an existing macro could be called by using the command **$M/FILEPATH\BUILDBOX 100 200 300**.

In this example, the three numbers (100, 200 & 300) become the three parameters supplied to the macro.

2.1.2 Examples of Command Syntax

The following are examples of command syntax that can be enter directly into the command window (or used within a macro)

Working with elements and attributes:

- To find out attribute information about the current element, enter **Q ATT**
- To create a new element (for example, BOX), enter **NEW BOX**
- To find out a specific attribute (for example NAME), enter **Q NAME**
- To print the variable value, enter **$P** at the start of define variable.
- To set a value to an attribute on the current element (for example XLEN), enter **XLEN 300**

Working with comments in statements

- To comment the pml code line, enter two hyphens (– –) at the start of the code line.
- The text after **$*** is a comment (used to comment out part of a line).
- To add multi line comment, enter **$(** at the start of comment line and enter **$)** at the end of comment.

Manipulating the draw list in DESIGN:

- To add the current element to the draw list, enter **ADD CE**
- To add a specific element below a piece of equipment, enter **ADD ONLY /E1301-S1**
- To remove all the elements from the draw list, enter **REM ALL**

Annotating elements in DESIGN:

- To label the current element with its name, enter **MARK CE**
- To label the element /PIPE with its name, enter **MARK /100-B-1-B1**
- To remove the label from the current element, enter **UNMARK CE**
- To remove all labels, enter **UNMARK ALL**
- To mark all branch elements with their names, enter **MARK WITH (NAME) ALL BRAN**
- To mark all large valve elements with their specification reference, enter **MARK WITH (NAME OF SPREF) ALL VALV WHERE CPAR [1] GT 10**

Adding colour to the 3D Model:

- To apply the standard, enhance colour to the current element, enter **ENHANCE CE**
- To apply colour 10 to element /PIPE, enter **ENHANCE /PIPE COL 10**
- To remove all colour from the 3D model, enter **UNENHANCE ALL**

Using Aid Graphics:

- To draw an unnumbered graphic aid line, enter **AID LINE E0N0U0 TO E0N1000U1000**
- To draw a sphere aid (aid number 1000), enter **AID SPHERE NUM 1000 E0N0U0 DIAM 200**
- To remove all unnumbered graphical aid lines, enter **AID CLEAR LINE UNN**
- To remove all number 1000 sphere aids, enter **AID CLEAR SPHERE 1000**
- To remove all graphical aids, enter **AID CLEAR ALL**

Position and Orientation of elements:

- To move the current element 1000mm in a N45E direction, enter **MOVE N45E DIST 1000**
- To move the current element E, until it is in line with /BOX, enter **MOVE E THRU /BOX**
- To rotate the current element around its origin (pointing up) by 45, enter **ROTATE BY 45 ABOUT U**
- To relatively move the current element east by 1000mm, enter **BY E 1000**
- To explicitly position the current element, enter **AT E0 N1000 U2000**
- To position & orientate the current element based on /EQUIP-N1, enter **CONNECT P2 TO P0 OF /EQUIP-N1**
- To position & orientate the current element based on the previous, enter **CONNECT P3 TO P1 OF PREV**

2.1.3 Syntax Graphs

- Syntax graphs are read from top left to bottom right. The start point is shown by >, and the user can follow any path through the graph until the exit point, shown by >, is reached.
- Points marked with a plus sign (+) are **option junctions** which allow the user to input any one of the commands to the right of the junction. For example:

```
>----+--- ABC -----.
     |             |
     |--- PQR -----|
     |             |
     `-------------+--->
```

- means the user can type in ABC *or* PQR *or* just press Enter to get the default option.
- Text in angle brackets <. . . > is the name of another syntax graph. This convention is used for syntax which occurs in many places. For example:

```
>--- +---ABC ----.
     |            |
     |---PQR -----|
     |            |
     |---<dia> ---|
     |            |
     `------------+--->
```

- means the user can type in ABC or PQR or any command allowed by the syntax given in diagram <dia> or just press Enter to get the default option.
- Points marked with an asterisk (*) are **loop back junctions**. Command options following these may be repeated as required. For example:

```
      .-----<-------.
     /              |
>---*--- option1 ---|
    |               |
    |--- option2 ---|
    |               |
    `--- option3 ---+--->
```

- means that the user can enter any combination of option1 *and/or* option2 *and/or* option3, where the options can be commands, other syntax diagrams, or command arguments.

Many examples of command syntax are provided in the various reference manuals within the installation folder of AVEVA™ E3D Design. For each a syntax graph is provided to show the various combinations of syntax available:

```
>-- ADD --+-- Only --+   .----<-------.
          |          |  /             |
          `----------*-- <selatt>  ---+-- COlour <colno> -->
                                      |
                                      `-->
```

The above syntax graph is for the ADD syntax (used to add elements to the draw list). From the graph it can be seen that only two words are required (e.g., ADD /PIPE) but others can be included (e.g., ADD ONLY /PIPE1 /PIPE2 COL 3)

The **$Q** syntax can also tell you the next allowable part of the syntax. For example, **ADD $Q-10** would print the 10 next available words that can follow the command **ADD**.

2.2 PML 2 – Object Orientated Programming

PML 2 is almost an object-oriented language, and it provides most features of other Object-Orientated Programming (OOP) languages. Operators and methods are polymorphic, and overloading of methods is supported. However, there is no inheritance and no concept of public or private variables.

PML 2 provides for classes of built-in, system-defined and user-defined object types. Objects have members (their own objects) and methods (their own functions). All PML variables instances of (1) built-in, (2) system-defined or (3) user-defined object.

Through the use of method concatenation, it is possible to achieve multi-operations in a single line of code. This means that PML 2 methods are typically shorter and easier to read than the PML 1 equivalent. While most PML 1 macros will still run within AVEVA™ E3D Design, PML 2 brings many new features that were previously unavailable. If used together with command syntax it must be expressed as a string before use.

2.2.1 Features of PML 2

The main features of PML 2 are:

- Available built-in variable types - STRING, REAL, BOOLEAN, ARRAY
- Built in Methods for commonly used actions
- Global Functions supersede old style macros
- User Defined Object Types
- PML Search Path (PMLLIB)
- Dynamic Loading of Forms, Functions and Objects (no need for synonyms)

2.2.2 Examples of Object-Orientated PML

Declaration of objects (variables):

- To declare a local variable as a REAL 3, enter **!realVariable = 3**
- To declare a global BOOLEAN variable as false, enter **!!booleanVariable = FALSE**
- To set the third value in an ARRAY as Fred, enter **!arrayVariable[3] = |Fred|**
- To declare a variable as an empty position object, enter **!position = object position()**

Finding out information about objects (variables)

- To find out a variable, enter **q var !exampleObject**
- To find out the object type of a variable, enter **q var !exampleObject.objectType()**
- To find out what members an object has, enter **q var !exampleObject.attributes()**

Methods available on objects:

- To find out what methods are available on an object, enter **q var !exampleObject.methods()**
- To find out is an object has been given a value, enter **q var !exampleObject.set()**
- To find out how large an ARRAY variable is, enter **q var !arrayVariable.size()**
- To find out how long a STRING variable is, enter **q var !stringVariable.length()**

Clearing an object:

- To empty an object, enter **!exampleObject.clear()**
- To delete an object that is no longer needed, enter **!exampleObject.delete()**

2.2.3 Software Customization Reference Manual

Many examples of the major objects available in AVEVA™ E3D Design are provided in the Software Customisation Reference Manual within the products installation folder. For each object, the members and methods are listed and explained.

- For each of the members the name, enter and purpose are provided.
- For each method the name, argument types, returned object type and purpose are provided.

2.3 PML Objects

Every variable defined with PML has an object type. This type is set when the variable is defined and is fixed for the life of the variable. When assigning a value to a variable, the object type needs to be considered, otherwise an error may occur. There are three groups of object types available:

- Built-in (e.g. String, Real, Boolean and Array)
- System-defined (e.g. Position, Orientation)
- User-defined

When a variable is declared as a specific object type, it is given all the members (attributes) and methods of the object definition. This means standard groupings can be setup to store data and that code repetition can be avoided.

A user-defined object provides an opportunity to group data together for a specific purpose. Once grouped as an object, it can be assigned to a variable and used as any other object. The following are examples of two user-defined objects:

```
define object FACTORY
          member .name is STRING
          member .workers is REAL
          member .output is REAL
endobject

define object PRODUCT
          member .productCode is STRING
          member .total is REAL
          member .site is FACTORY
endobject
```

These objects would be defined as two separate object files (**.pmlobj**) and loaded into AVEVA™ E3D Design. You will notice that the object **PRODUCT** is able to have a member which is another user-defined object. This means that the **PRODUCT** object has access to all the members and methods of a **FACTORY** object.

2.3.1 Creating Variables (Instances of object)

There are two types of variables in PML (1) Local and (2) Global

The difference between the two is that global variables last for the whole AVEVA™ E3D Design session and can be referenced directly from other PML routines. Local variables are only available within the routine which defined them. The variables a declared with a single '**!**' for local and a double '**!!**' for global.

- **!localVariable** a local variable
- **!!globalVariable** a global variable

A variable object-type can be implied from the assigned value:

`!name = |Fred|`	To create a LOCAL, STRING variable
`!!answer = 42`	To create a GLOBAL, REAL variable
`!!flag = TRUE`	To create a GLOBAL, BOOLEAN variable

It is also possible to define a variable as an object-type without an initial value. The value is therefore **UNSET**

`!name = STRING()`	To create a LOCAL, UNSET STRING variable
`!!answer = REAL()`	To create a GLOBAL, UNSET REAL variable
`!array = BOOLEAN()`	To create a LOCAL, UNSET ARRAY variable

2.3.2 Naming Conventions

It is common practice to follow a naming convention when defining objects and variables. Using upper case for the name of the object type and mixing upper and lower case for the variable is a good practice to follow.

For example, the type might be **WORKERS** while the name of the variable might be **numberOfWorkers**.

Notice that to make the variable name more meaningful, full words are used with a mixture of upper and lower case letters to make it readable.

Variable names should not start with a number or contain any spaces or full stops (full stops are used in PML2 to indicate methods and members – explained later)

AVEVA uses a CD prefix on most of its global variables. Newer functionality does not use a prefix so all new PML must be checked for name clashes. Using your own prefix could help avoid this.

2.3.3 Using the Members of an Object

When a variable is declared as an object-type, it is also given all the objects members and methods. For example, to a local variable as a factory object:

```
!newPlant = object FACTORY()
```

After being declared as above (using the OBJECT keyword and ending with a double bracket, the local variable **!newPlant** is now a **FACTORY** object and has the same members as the **FACTORY** object. These members are available to store information and can be assigned values in the following way:

```
!newPlant.name = |ProcessA|
!newPlant.workers = 451
!newPlant.output = 200
```

Notice the use of a dot between the variable and its member. This works as long as the word after the dot is a valid member of the variable object-type.

Once assigned a value, this value is available for use and can be recalled. For example:

```
!numberOfWorkers = !newPlant.workers
```

This creates a new local real variable and assigns it the value 451.

2.3.4 Special Objects used in E3D Design

In a standard E3D DESIGN session, there are number of specialised objects which are loaded and used by standard product. These objects should not be deleted or overwritten but are available for use. The particularly useful objects are

!!CE	- a global DBREF object which tracks and represents the current element
!!ERROR	- a global ERROR object which holds information about the last error

!!PML	-	Used to obtain file path strings through the .getPathName() method
!!ALERT	-	Used to provide popup feedback to user
!!AIDNUMBERS	-	Used to manage aid graphics
!!APPDESMAIN	-	The form which represents the main MODEL interface
!!FMSYS	**-**	Provides a variety of system-oriented tasks such as !!FMSYS.SHOWFORMS() returns an ARRAY of all shown forms

2.4 PML Functions and Methods

Functions and methods provide the actions of PML. When called, a function or method will run through the lines of PML it contains in order – just like a PML 1 style macro. Functions and methods may be passed arguments and even return values. A function which does not return a value is typically referred to as a PML Procedure.

A function is a global method (stored in its own file) and can be called directly on the command line (e.g. **call !!exampleFunction()**) while a method is local to the object it is defined within (e.g. **!exampleObject.exampleMethod()**).

Arguments become local variables within the function/method and the object-types need to be declared within the definition. The returned object-type is also defined. For example:

```
define function !!area ( !length is REAL, !width is REAL) is REAL
    !area = !length * !width
    Return !area
endfunction
```

In this example, the function **!!area** is expecting two real arguments. The two arguments are expressed as local variables which are multiplied together to calculate the local variable **!area**. Using the return keyword, the variable **!area** is then returned. If the function was called in following way, the variable !area will have a value of 2400:

```
!area = !!area(12, 200)
```

2.4.1 Arguments of type ANY

When defining an argument within a method or function, you may declare it as an **ANY** object. This means that the argument can be of any object-type, allowing any variable to be supplied to the function. As no argument check is carried out, the function needs to be robust enough to cope with any argument. For this reason, **ANY** should be used with special consideration:

```
define function !!argumentType(!argument is ANY) is STRING
    !type = !argument.objecttype()
    return !type
endfunction
```

2.5 PML Forms

For users of **AVEVA™ E3D Design** the concept of a form should already be familiar. In terms of PML, a form is a global variable (for example, **!!exampleForm**). A form is capable of owning members and methods (like any other object) but there are a set of predefined members which can be used to put gadgets on the form (buttons, lists, options etc). These gadgets are objects in their own right and can be given callbacks, which can activate a standard command or call a function or run a method defined within the form.

```
setup form !!nameCE
    !this.formTitle = |Name CE|
    Button .Button |Print Name Of CE| call |!this.print()|
exit
define method .print()
    !name = !!ce.flnn
    $p Name of Current Element = $!name
Endmethod
```

The above example is the definition of a form called **nameCE**. Saved within one file (nameCE.pmlfrm), it defines two form members (a predefined member for the title and a new button gadget) and a form method. **!this** is a special local variable and using it replaces the need to reference the owning object directly. For example, to call the method **.print()** from within the form, the call is **!this.print()** and to call the method from anywhere else, it's **!!nameCE.print()**.

2.6 PMLUI Environment Variable

All PML1 Macros are typically stored a directory structure pointed at by the **AVEVA™ E3D Design** environment variable **PMLUI**. The PMLUI environment variable is set in the .bat or .init file that is used to load **AVEVA™ E3D Design** (typically within evars.bat or evars.init file). To set the PMLUI environment variable, the following line is needed in the .bat file:

```
set PMLUI=C:\AVEVA\Customisation\PMLUI
```

The purpose of an environment variable is to reduce the length of the command used to call a macro. This means that **$M/%PMLUI%\DES\PIPE\MPIPE** can be entered instead of the full file path.

In standard product, this process is shortened further as all PML1 macros and forms are called using synonyms. For example, the macros associated with piping are called using the synonym **CALLP**:

```
$S CALLP=$M/%PMLUI%/DES/PIPE/$s1
CALLP MPIPE
```

*If all synonyms are killed, then **AVEVA™ E3D Design** will cease to function as normal.*

2.7 PMLLIB Environment Variable

All PML 2 objects and functions are in a directory structure pointed at by the **AVEVA™ E3D Design** environment variable **PMLLIB**. As with the PMLUI variable, this is set in the .bat or .init file which runs **AVEVA™ E3D Design**. To set the environment variable, the following line is needed in the .bat file:

```
set PMLLIB=C:\AVEVA\Customisation\PMLLIB
```

The PMLLIB environment variable differs because it can be searched dynamically. This means that the individual files do not need to be referenced directly and can be called by name. This is possible because **AVEVA™ E3D Design** compiles a **pml.index** file which sits in the PMLLIB folder and provides the path to all suitable files within it. For example, to load and show a form the command is **show !!exampleForm**, there is no need to reference the file path at all.

If a new file is created or the PMLLIB variable is changed then there is a need to update the pml.index file. This can be done by entering **PML REHASH** onto the command window. If there are multiple paths that need updating, enter **PML REHASH ALL**

If a PML object has already been loaded into **AVEVA™ E3D Design**, but the file definition has changed then the object needs to be killed and reloaded before the changes can be seen. This can be done by entering either **pml reload form !!exampleForm** or **pml reload object EXAMPLEOBJECT**.

2.8 Modifications to the PMLUI and PMLLIB

As PML is developed, it is normal practice to store it in a parallel file structure outside the standard installation directory. This parallel file structure can then be reference by the .bat files. There are a couple of reasons why this is a good idea:

- It keeps the customisation separate from the standard install, so as subsequent versions are installed there will not be a need to move the customised files.
- If any standard files are modified, then the originals are still available if required.
- If the customisation fails, the standard installation is available to go back to.
- Many local **AVEVA™ E3D Design** installations can reference the same customisation from a network address.

*Any changes to AVEVA standard product may cause **AVEVA™ E3D Design** to function inappropriately.*

It is possible to get **AVEVA™ E3D Design** to look in different places for PML files and this is done by setting the environment variables to multiple paths. This allows the standard install to be kept separate from user and company customisation.

The **evars.init** file is known as product initialization file and it is stored in the local installation directory of the product and all modifications will be available on locally.

C:\Program Files (x86)\AVEVA\Everything3D3.1\evars.init

The **custom_evars.bat** file is known as custom initialization file, and it is stored on company's fileserver where all projects are listed. In this case it allows client to maintain companies' specific PML environment dependencies available to everyone across the company working with projects.

C:\Users\Public\Documents\AVEVA\Projects\Everything3D3.1\custom_evars.bat

This is done by updating the variable to include another path, for example:

```
set PMLUI=C:\Temp\pmlui;%pmlui%
```

```
set PMLLIB=C:\Temp\pmllib;%pmllib%
```

This will put the additional file path in front of the standard (which would have already been defined in the .bat file). This change can also be checked in an **AVEVA™ E3D Design** session by entering **q evar PMLUI** or **q evar PMLLIB** onto the command window.

Exercise 1 Updating the Environment Variables

Perform the following tasks:

- Extract the provided files into the folder a suitable folder.
- In the Training environment, C:\Users\Public\Documents\Aveva\Training3.0\USERDATA\PMLEX directory will be used to store PML samples and user PML files.
- Enter into the below path C:\Users\Public\Public Documents\AVEVA\Training3.0\Projects directory and open the **custom_evars.bat** file in text editor.
- Check the file contains the line **set PMLLIB=C:\Users\Public\Documents\Aveva\Training3.0\USERDATA\PMLEX;%PMLLIB%**
- Save and close the file.
- Start AVEVA™ E3D Design
- Enter the following data

  Project: **Training (TRA)**

  User: **A.TRAINEE**

  Password: **A**

  MDB: **A-MDB**

  Click the **Model** tile.
- Query **PMLLIB** environment variable.

CHAPTER 3

3 Macros, Synonyms and Control Logic

A macro is a group of **AVEVA™ E3D Design** commands written to a file (in sequence) that can be run together. This removes the need for user to have to enter every line of code separately and those repetitive processes can be run as a macro.

3.1 A Simple Macro

The following example is a macro which creates an EQUI element which owns a BOX and CYLI. To test the example, copy the code from guide and paste as macro into the command window.

When running this macro in, ensure the Current Element (CE) is a ZONE or below.

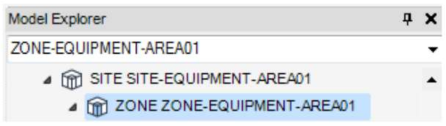
```
NEW EQUIP /ABCD
NEW BOX XLEN 300 YLEN 400 ZLEN 600
NEW CYL DIA 400 HEI 600
CONN P1 TO P2 OF PREV
```

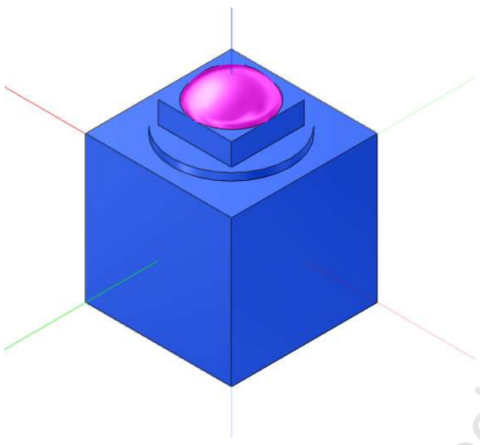
To run the macro into **AVEVA™ E3D Design**, either drag and drop the file into the command window or enter **$M/** followed by the full file path of the saved macro.

Exercise 2 Creating a centre of Handwheel

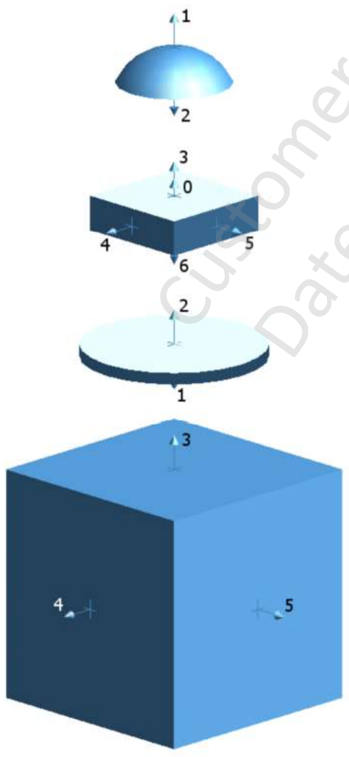
Perform the following tasks:

- Write a macro that will produce the centre of a handwheel. Create it under ZONE-EQUIPMENT-AREA01.
- It shall be constructed from 4 primitives: 2 boxes, a cylinder and a dish and should create the following hierarchy:


Model Explorer
ZONE-EQUIPMENT-AREA01
SITE SITE-EQUIPMENT-AREA01
ZONE ZONE-EQUIPMENT-AREA01



- The macro shall create the primitive with the following fixed sizes:
    - FIRST BOX: XLEN=100, YLEN=100, ZLEN= 100
    - CYLI: DIA= 80mm, HEI= 5 mm
    - SECOND BOX: XLEN= 50, YLEN = 50, ZLEN= 15
    - DISH: DIA= 50mm, HEI= 15mm



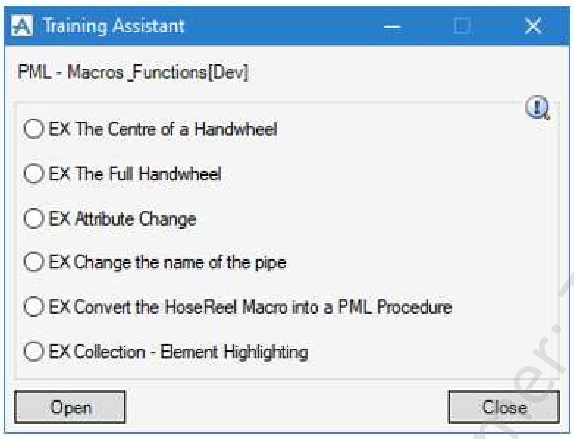
- Let the first BOX be created without specifying a position or orientation.
- Use the CONN syntax to position the next primitive to the first BOX. For example:

  CONN P1 TO P1 OF PREV

- This will connect PPoint 1 (P1) of the current primitive to P1 of the primitive created previously.
- To help with connecting the primitives, refer to the adjacent diagram showing an exploded version of the equipment, with the PPoints identified.

- The macro can be extended with the following commands:
  - To add the Handwheel to the drawlist
    ADD /HandWheel
  - To set the limits of the view to the HandWheel
    AUTO /HandWheel
  - To completely clear the drawlist
    REM ALL
- Save the file as C:\Users\Public\Documents\AVEVA\ Training3.0\USERDATA\PMLEX\ex2.mac and run it from the command line by entering **$m/%PMLLIB%\ex2.mac** or by dragging and dropping the file into the command window.

An example of the completed macro can be found in the Training Assistant



- EX The Centre of a Handwheel
- EX The Full Handwheel
- EX Attribute Change
- EX Change the name of the pipe
- EX Convert the HoseReel Macro into a PML Procedure
- EX Collection - Element Highlighting

- On the ***Training*** tab, in the ***Tools*** group, clicking the **Assistant** button displays the ***Training Assistant*** form.

3.2 Finding Examples of Command Syntax

While developing a macro there are four main techniques of deriving the command syntax needed:

• DB listing utility	create the required elements in **AVEVA™ E3D Design** using the standard appware and use the DB Listing utility to output the information ***Manage > History*** **> DB Listing**
• $Q syntax	if part of a command is known, the syntax which follows it can be found by entering **$Q** after the command e.g. **NEW $Q**
• Standard product	standard **AVEVA™ E3D Design** is supplied with numerous PML files which can be searched for keywords and used for inspiration
• Reference Manuals	the documentation and reference manuals are available online at AVEVA help desk which is available by clicking F1 in **AVEVA™ E3D Design 3.1**

As macros develop in complexity and use, it is likely that a mixture of the above will be used.

3.3 Communicating with AVEVA Products in PML

All commands need to be supplied to the command processor as STRINGs. This is important when working with variables as they may have another object-type. If a **$** symbol is put in front of a variable, **AVEVA™ E3D Design** will expand the contents of the variable as a string and then read the line. For example:

```
!componentType = |BOX|
 !xLength = 5600
NEW $!componentType XLEN $!xLength
```

This is the equivalent of entering **NEW BOX XLEN 5600**

3.4 Parameterized

Macros can be parameterised. This means instead of hard coding the values in the macro, arguments can be passed to it and used instead. This allows the values to be varied, making the macro flexible. As an example, simpleMac.mac could be parameterised in the following way:

```
NEW EQUIP /$1
NEW BOX
XLEN $2 YLEN $3 ZLEN $4
NEW CYL DIA $3 HEI $4
CONN P1 TO P2 OF PRE
```

To run this Macro, parameters need to be passed to it. The four parameters are defined by including the values of the parameters after the macro call:

e.g. **$M/C:\Users\Public\Documents\AVEVA\Training3.0\USERDATA\PMLEX\parameterMac.mac**

ABCDE 300 400 60

If this macro is dragged and dropped into the command window, the parameters would be undefined and the macro would fail. To avoid this, default parameter values can be set at the top of the macro using the **$d=** syntax. For example:

```
$d1=ABCDEF
$d2=300
$d3=400
$d4=600
```

If defined, the default values will only be used if no parameters are passed to the macro.

Macros may have up to nine parameters separated by space. In the below example, ABC, DEF and GHK are seen as separate strings and therefore different parameters:

e.g. **$M/%PMLUI%\examples\nineParameter.mac ABC DEF GHK 55 66 77 88 99 0**

If a text string is required as a single parameter, it can be entered by placing a **$<** before and a **$>** after the string. $< $> act as delimiters and anything in between is interpreted as a single parameter.

e.g. **$M/%PMLUI%\examples\sevenParameter.mac $<ABC DEF GHK$> 55 66 77 88 99 00**

3.5 Synonyms

Synonyms are abbreviations of longer commands. They are created by assigning a command to a synonym variable. To call the command held by the synonym, just enter the name of the synonym.

e.g. **$SNewBox=NEW BOX XLEN 100 YLEN 200 ZLEN 300** called by entering **NewBox**

It is also possible to parameterise a synonym:

e.g. **$SNewBox=NEW BOX XLEN $S1 YLEN $S2 ZLEN $S3** called by entering **NewBox 100 200 300**

A synonym can call itself and if it uses the **$/** syntax (return character) it can be used to loop through elements. For example, a new XLEN value can be applied to all elements for level in the hierarchy.

e.g. **$SXChange=XLEN 1000 $/ NEXT $/ Xchange**

Synonyms can be turned off and on with the **$S-** and **$S+** syntax. To kill a synonym, enter **$SXXX=** and to kill all synonyms **$SK**. Standard **AVEVA™ E3D Design** relies on synonyms to function.

If a required synonym is killed, the product will no longer function properly (requiring a restart)

3.6 Defining Variables

There are number of different methods for defining variables in **AVEVA™ E3D Design**:

3.6.1 Numbered Variables

Numbered variables are set by entering a number then value after the command VAR. Examples of this are below:

```
var 1 name
var 2 |hello|
var 3 (99)
var 4 (99 * 3 / 6 + 0.5)
var 117 pos in site
var 118 (name of owner of owner)
var 119 |Hello| + |world| + |how are you|
```

The value of variable 1 can be obtained by entering **q var 1** into the command window. The available variable numbers only go to 119 (there is no 120) and they are module dependant.

This technique is no longer commonly used and has been included for completeness.

3.6.2 PML 1 Style Variables

The following are some examples of setting variables using the PML 1 style syntax:

`VAR !NAME NAME`	Takes the current element's (CE) name attribute
`VAR !POS POS IN WORLD`	Takes CE position attribute relative to the world
`VAR !x |NAME|`	Sets the variable to the text string |NAME|
`VAR !temp (23 * 1.8 + 32)`	Calculate a value using the expression
`VAR !list COLL ALL ELBO FOR CE`	Makes a string array of database references

This style of defining variables is still valid and is useful for command syntax that returns a value. It has been included within this guide primarily for information for when old code is upgraded.

3.6.3 PML 2 Style Variables

The following are some examples of setting variables using the PML 2 style syntax:

`!name = !!ce.name`	Takes the current element's (CE) name attribute
`!pos = !!ce.pos.wrt(/*)`	Takes CE position attribute relative to the world
`!x = |NAME|`	Sets the variable to the text string |NAME|
`!temp = 23 * 1.8 + 32`	Calculate a value using the expression

These examples show that there are equivalents to the PML 1 style, but because PML 2 is object-based this can be incorporated into the declaration of the variable.

The equivalent PML2 collection will be explained later in the training guide.

3.7 Expressions

Expressions are calculations using PML variables. This can be done in a PML 1 or PML 2 style

```
VAR !Z ( |$!X| + |$!Y| )          PML 1
!Z = !X + !Y                      PML 2
```

In the PML1 example, !Z is set as a STRING variable. In the PML2 example, !Z is returned as a REAL, if !X and !Y are REAL

3.7.1 Expression Operators

There are a number of expression operators which are available for use within PML. The numeric operators and functions are the same for both PML 1 and PML 2 styles. For comparison and logic operators there are new PML 2 methods available:

Numeric operators:	**+ - / ***
Numeric functions:	**SIN COS TAN SQR POW NEGATE ASIN ACOS ATAN LOG ALOG ABS INT NIN**
PML 1 style Comparison operators:	**LT GT EQ NE LE GE**
PML 1 style Logic operators:	**NOT AND OR**
PML 2 style Comparison methods:	**.lt() .gt() .eq() .neq() .leq() .geq()**
PML 2 style Logic methods:	**.not() .and() .or()**

The PML 2 comparison methods are available on any object-type variable, providing it is compared to a variable of the same type. The PML 2 logic methods are available on the BOOLEAN object

*For further examples, refer to **AVEVA™ E3D Design** Software Customisation Reference Manual:*

Some examples of expressions in use:

```
!s = 30 * sin(45)
!t = pow(20,2)              (raise 20 to the power 2 (=400))
!f = (match(name of owner, |LPX|) gt 0)
```

3.7.2 Operator Precedence

When **AVEVA™ E3D Design** reads an expression, there is a precedence applied to it. This should be considered when writing expressions. The precedence order is as follows:

( )

*** /**

+ -

EQ NE GT LT GE LE

NOT

AND

OR

e.g 60 * 2 / 3 + 5 = 45

60 * (2 / (3 + 5)) = 15

3.7.3 PML 2 Expressions

PML 2 expressions may be of any complexity and may derive the required values from PML Functions and Methods and include Form gadget values, object members and methods. For example:

```
!newValue = !!myFunc(!OldVal) * !!form.gadget.val / !myArray.method()
```

3.8 Arrays

An ARRAY variable can contain many values, each of which is an ARRAY ELEMENT.

An array is created when an array element is defined, or it can be initialised as an empty array (i.e. **!x = ARRAY()**). If an ARRAY ELEMENT is itself an ARRAY, this will create a Multi-dimensional ARRAY. For example, enter the following onto the command window to define an array:

```
!x[1] = |ABCD|
!x[2] = |DEFG|
!y[1] = |1234|
!y[2] = |5678|
!z[1] = !x
!z[2] = !y
```

To query the information about !z, enter **q var !z**. This will return the following information:

```
<ARRAY>
   [1]  <ARRAY> 2 Elements
   [2]  <ARRAY> 2 Elements
```

To find out more information about the elements within the Multi-dimensional array, enter **q var !z[1]** or **q var !z[2][1]**

3.9 Concatenation Operator

Values are automatically converted to STRING and concatenated when the '**&**' operator is used. Enter the following onto the command window:

```
!a = 64
!b = 32
!m = |mm|
!c = !a & !b & !m
q var !c
```

Compare this against the results of entering **!d = !a + !b & !m**

3.10 Do Loop

A **DO** loop is a way of repeating PML, allowing pieces of code to be run more than once. This is useful as it allows code to be reused and reduces the overall number of lines. As an example, check the below code.

DO Loops with BREAK

```
DO !loopCounter TO 10
    !value = !loopCounter * 2
    q var !loopCounter !value
ENDDO
```

In the above example, as the loop runs, values of !loopCounter and !value with be printed to the command line for the full range of the defined loop. The step of the loop can be altered by adding FROM and BY to the loop definition. As an example, check the below code.

```
DO !loopCounter FROM 5 TO 10 BY 2
    !value = !loopCounter * 2
    q var !loopCounter !value
ENDDO
```

3.10.1 DO Loops with BREAK

If you need a loop to run until a certain condition is reached, a **BREAK** command will exit the current loop. This can be used in conjunction with an infinite loop or with a loop with a range. As an example, try the below code

```
DO !n
    !value = POW(!n, 2)
    q var !value
    BREAK IF (!value GT 1000)
ENDDO
```

The loop in the example will run until the BREAK condition is met. If the condition is never reached, then the code will run indefinitely!

*It is good practice to give a loop a range, even if very large (i.e. 1 to 100000). This ensures that the loop has an end and won't require **AVEVA™ E3D Design** to be crashed to exit it.*

The BREAK command can also be called from within a normal IF statement. This is typically done if multiple break conditions need to be considered or if additional code needs running. For example:

```
IF (!value GT 1000) THEN
     !!alert.message(|1000 reached|)
    BREAK
ENDIF
```

3.10.2 DO Loops with SKIP

It is possible to skip part of the DO loop using the **SKIP** command. This could be useful if parts of a number sequence need to be missed. As an example, try the below code.

```
DO !n FROM 1 TO 25
    SKIP IF (!n LE 5) OR (!n GT 15)
    q var !n
ENDDO
```

The SKIP command can also be called within a normal IF statement (as with the BREAK command).

3.10.3 DO INDEX and DO VALUES

DO INDEX and **DO VALUES** are ways of looping through arrays. This is an effective method for controlling the values used for the loops. Typically, values are collected into an ARRAY variable then looped through using the following:

`DO !X VALUES !ARRAY`	!X takes each ARRAY element as its value
`DO !X INDEX !ARRAY`	!X takes an incremental number from 1 to !ARRAY size

As an example, try the below code.

```
VAR !zones COLL ALL ZONES FOR SITE
VAR !names EVAL NAME FOR ALL FROM !zones

q var !names
$P
$P ************************************

DO !x VALUES !names
    q var !x
ENDDO
$P
$P ************************************

DO !x INDEX !names
    q var !names[!x]
ENDDO
```

3.11 IF STATEMENTS

An **IF** statement is a construct for the conditional execution of commands. The commands within the statement will only be run if the conditions are met. In the following example, the code within the IF construct is only run if the expression is **TRUE** (i.e. !number is real and less than 0)

```
IF ( !number LT 0 ) THEN
    !negative = TRUE
ENDIF
```

The expression can be written in any form, providing the answer is **BOOLEAN**. For example, the following is the same as above but written in a PML 2 style:

```
IF ( !number.lt(0) ) THEN
```

If the variable itself is already BOOLEAN, a comparison does not need to be made. For example:

```
!booleanVariable = TRUE
IF ( !booleanVariable ) THEN
```

or

`IF ( !!exampleFunction() ) THEN`, where !!exampleFunction() returns a BOOLEAN result.

3.11.1 IF, ELSEIF and ELSE Statements

An **IF** statement is extended by adding additional conditions to it and this is done by using **ELSEIF** or **ELSE** statements. When an IF statement is encountered, **AVEVA™ E3D Design** will evaluate its first condition. If the condition is **FALSE**, **AVEVA™ E3D Design** will look to the next ELSEIF condition.

Once a condition is found to be **TRUE**, that part of the code will be run and then the IF statement is complete. If an ELSE condition is added, this portion of code will only be run if no other conditions are met. This is a way of ensuring some code runs as part of the construct. As an example, copy the below code to the file and save it in the path C:\Users\Public\Documents\AVEVA\Training3.0\USERDATA\PMLEX\numCheck.mac and run the macro with one real parameter for the comparison.

```
$d1=1
IF ($1 EQ 0) THEN
    $P Your value is zero

ELSEIF ($1 LT 0) THEN
    $P Your value is less than zero

ELSE
    $P Your value is greater than zero
ENDIF
```

The ELSEIF and ELSE commands are optional, but there can only be one ELSE command in an IF construct.

3.12 Branching

PML provides a way of jumping from one part of a macro to another.

Syntax **LABEL /LABEL_NAME** defines a label
. . .
Some PML code
. . .
Syntax **GOLABEL /LABEL_NAME** defines a command to jump to the label

The next line to be executed after **GOLABEL /LABEL_NAME** will be the line following **LABEL /LABEL_NAME**, which could be before or after the **GOLABEL** command. The name of the label can be up to 16 characters (excluding the leading slash)

The use of this method should be limited as it can make code hard to read and therefore to debug.

3.12.1 Conditional Branching

It is possible to add a condition to the GOLABEL syntax such that the jump will only be made if the condition is TRUE. As an example, enter the below code and save it as

C:\Users\Public\Documents\AVEVA\Training3.0\USERDATA\PMLEX\conditional.mac:

```
DO !A
    $P Processing $!A
    DO !B TO 10
        !C = !A * !B
        GOLABEL /finished if ( !C GT 100 )
        $P Product $!C
    ENDDO
ENDDO

LABEL /finished
$P Finished with processing = $!A Product = $!C
```

If the expression **!C GT 100** is **TRUE** there will be a jump to label /**finished** and PML execution will continue with the **$P** command. If the expression is **FALSE**, PML execution will continue with the command: **$P Product $!C** and go back through the DO loop.

3.13 Error Handling

An error condition can occur when a command could not complete successfully. This is because of a mistake in the executed macro or function. An error normally has three effects:

- An Alert box appears which the user must acknowledge.
- An error message is outputted to the command line together with a trace back to the error source.
- Any current running PML macros and functions are abandoned.

3.13.1 Error Codes

When an error occurs, an error code is returned to the user along with a message. The following example error is caused when trying to create an EQUI element in the wrong part of the hierarchy.

(41,8) ERROR – Cannot create an EQUI at this level

Where 41 is the program section which identified the error and 8 is the error code itself.

3.13.2 Error Handling Using the HANDLE Syntax

If the input line which caused the error was part of a PML macro or function, the error may optionally be **HANDLED**. This allows the designer of the macro to limit the errors the user will experience.

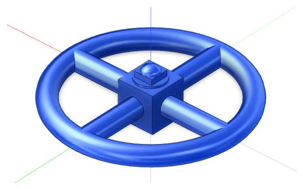
As an example, copy the below code to the file C:\Users\Public\Documents\AVEVA\Training3.0\USERDATA\PMLEX\errorTest.mac. First run the macro at a **SITE** element, then at a **ZONE** element and then again at the same ZONE. Compare the return printed lines in the command window.

```
NEW EQUI /ABCD

HANDLE (41, 8)
    $p Need to be at a ZONE or below

ELSEHANDLE (41, 12)
    $p That name has already been used. Names must be unique

ELSEHANDLE ANY
    $p Another error has occurred

ELSEHANDLE NONE
    $p Everything OK. EQUI created

ENDHANDLE
```

Notice how the HANDLE syntax can differentiate between specific error codes and how it is possible to capture all errors or only run if no error occurs.

Exercise 3 Creating a complete Handwheel

Perform the following tasks:



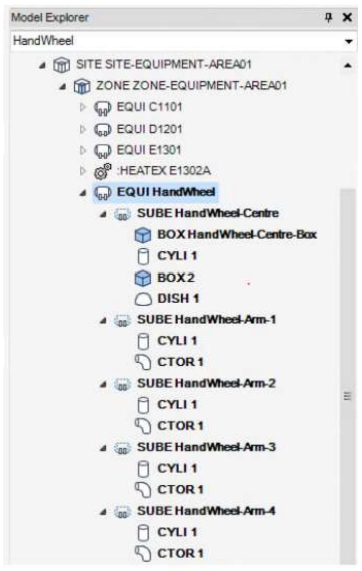
- Extend the previous macro to build the remaining parts of the Handwheel (shown below).
- Information about the primitives can be found in Appendix A
- It is possible to build the remaining primitives individually, but as there is a rotational centre, a copy-rotate syntax could be used.
- To copy a SUBE element (where /XXXX is another SUBE.

  NEW SUBE /XXXX COPY PREV

- To rotate the CE (where AA is angle and BB is direction)

  ROTATE BY AA ABOUT BB

- To help with organization of the primitives of the EQUI, you may wish to add additional SUBE elements. This could help with the copy-rotate methods.
- An example hierarchy is shown to the left but this will depend on your macro code.
- Turn the macro into a parameterized macro. Pass it two parameters (1) the name of the EQUI and (2) the outside diameter of the HandWheel.
- Think about how the sizes of the primitives will relate to the outside diameter. You may need to do some calculations before the value can be used.
- As parameters are used, set the default values.



```
Model Explorer
HandWheel
◢ SITE SITE-EQUIPMENT-AREA01
  ◢ ZONE ZONE-EQUIPMENT-AREA01
    ▷ EQUI C1101
    ▷ EQUI D1201
    ▷ EQUI E1301
    ▷ :HEATEX E1302A
    ◢ EQUI HandWheel
      ◢ SUBE HandWheel-Centre
          BOX HandWheel-Centre-Box
          CYLI 1
          BOX 2
          DISH 1
      ◢ SUBE HandWheel-Arm-1
          CYLI 1
          CTOR 1
      ◢ SUBE HandWheel-Arm-2
          CYLI 1
          CTOR 1
      ◢ SUBE HandWheel-Arm-3
          CYLI 1
          CTOR 1
      ◢ SUBE HandWheel-Arm-4
          CYLI 1
          CTOR 1
```

- If any errors occur in your macro, consider error handling or changing the macro to avoid errors.
- Save the file as C:\Users\Public\Documents\AVEVA\Training3.0\USERDATA\PMLEX\ex3.mac and run it on the command line by entering **$m/%PMLLIB\ex3.mac** or by dragging and dropping the file into the command window.

An example of the completed macro can be found in the Training Assistant

Exercise 4 Changing Attribute

Perform the following tasks:

- Create a macro file to change the insulation specification to /60mm_FibreGlass for all elbow with bore diameter greater than or equal to 100mm within the SITE/SITE-PIPING-AREA01.
- Change the value of the attribute "Ispec" which defines the insulation specification assigned to the CE.
- Mark all the collected elbows with the name of the pipe they are part of. This can be done by using command **MARK WITH (NAME OF PIPE OF AAA)** where **AAA** is the name of the current element.
- To remove all marks a command **UNMARK ALL** can be used.
- Save the file as C:\Users\Public\Documents\AVEVA\Training3.0\USERDATA\PMLEX\ex4.mac and run it on the command line by entering **$m/%PMLLIB\ex4.mac** or by dragging and dropping the file into the command window.

An example of the completed macro can be found in the Training Assistant

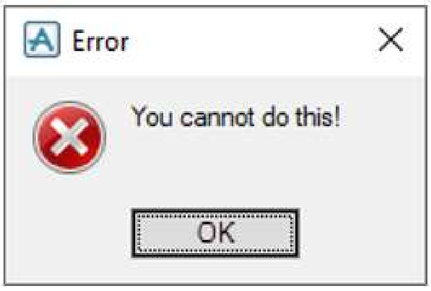
3.14 Alert Objects

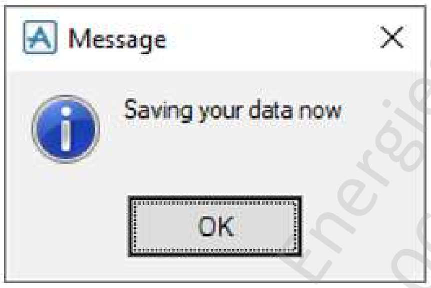
Alert objects allow user-controlled pop-up forms to be used, providing feedback to the user. The choice on which form type is used and whether one is needed at all is down the PML designer.

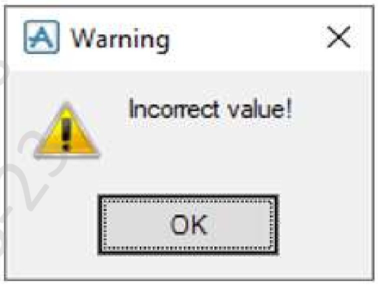
3.14.1 Alert Objects with no Return Value

There are three types of alert with no return value.

```
!!alert.error ( |You cannot do this!| )
!!alert.message ( |Saving your data now| )
!!alert.warning ( |Incorrect value!| )
```







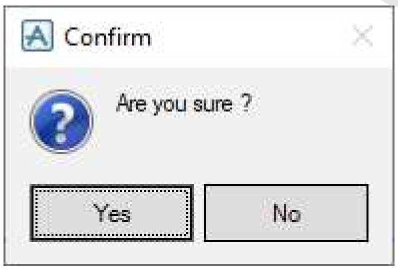
By default, all alert forms appear with the relevant button as near to the cursor as possible. To position an alert specifically, X and Y values can be specified as a proportion of the screen size.

```
!!alert.error( |You cannot do this!| , 0.25, 0.1 )
```

3.14.2 Alert Objects that return value

There are three types of alert which return a value. This value can be subsequently used to as an indication of the decision the user has made (i.e. as part of an IF statement).

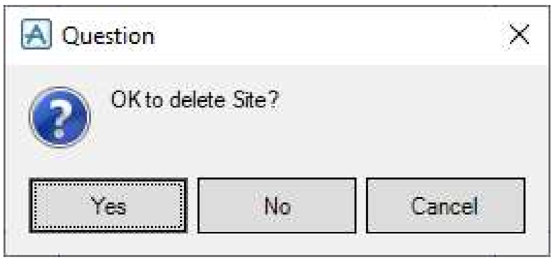
3.14.2.1 Confirm Alerts



A Confirm alert returns a string of 'YES' or 'NO

```
!answer = !!alert.confirm(|Are you sure ?|)
q var !answer
```

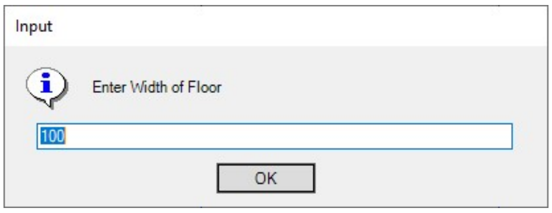
3.14.2.2 Questions Alert



A Question alert returns a string of 'YES' or 'NO' or 'CANCEL'

```
!answer = !!alert.question(|OK to delete
Site?|)
q var !answer
```

3.14.2.3 Input Alerts



An Input alert returns the value entered by the user. If the user does not enter a value, then the default value (set at definition) is returned.

```
!answer = !!alert.input( |Enter Width of
Floor|, |100| )
q var !answer
```

3.15 Undo and Redo

If you build an application that creates or deletes items from the **AVEVA™ E3D Design** database it is good practice to handle the ability to undo the modifications. Undo is a useful feature that users expect to be able to use.

3.15.1 Marking the Database

The undo button goes back to the last database mark or savework so your application should mark the database before modification.

MarkDB |Comment|

3.15.2 Undo and Redo Database Commands

UndoDB - Undo the database back to the last database mark

RedoDB - Redo the last Undo

These actions can be performed by including the above lines within your code. There is also an **UNDOABLE** object with built-in methods which can extend this functionality further.

Refer to the Software Customisation Reference Manual for information about the UNDOABLE object

4 PML Functions

Functions are lines of PML grouped together in a file designed to do something. When called, the lines of the function are run, and the intended action completed.

4.1 Functions Instead of Macros

Functions are designed to replace macros and can contain the same syntax that would have been placed inside a PML 1 style macro. It is intended that macros should no longer be used and any future PML should be written as functions. The following are the reasons why:

- Functions are preloaded through the PMLLIB search path.
- Functions can accept arguments of specific object-types.
- Functions can return values of specific object-types.

4.2 Creating a PML Function

A function is saved as a **.pmlfnc** file under the PMLLIB search path. The filename should be the same name as the function, for example, if the function is called **!!exampleFunction** then the file should be called **exampleFunction.pmlfnc**.

A function will typically return a value, although arguments are optional. The object-type of the returned object needs to be defined and it is done at the end of the define function line.

The following is an example of a simple function designed to return the full name of the current element. It is an example of a **RETURN** function with **NO ARGUMENTS**

```
define function !!nameCE() is STRING
    !ce = !!ce.flnn
    return !ce
endfunction
```

After it has been saved as a .pmlfnc file and a **PML REHASH ALL** on the command line, the function can be called by entering:

```
!name = !!nameCE()
```

After running this line, variable !name will be a string holding the full name of the current element.

If a function is defined with arguments, then these arguments will become local variables available within the function. These can then be used in expression, to control the function etc. The following is an example of a **RETURN** function with an **ARGUMENT**:

```
define function !!area( !radius is REAL ) is REAL
    !circleArea = !radius.power(2) * 3.142
    return !circleArea
endfunction
```

After definition, the function can be used as part of an expression as it returns a real object. This means that common expressions/calculations can be written as a function and used as required.

```
!height = 64
!cylinderVolume = !!area(2.3) * !height
q var !cylinderVolume
```

4.3 PML Procedures

A function that does not return a value is typically described as a PML procedure. This means that as there is no return and the return object-type does not need to be defined. The following example of a **NON-RETURN** function with **NO ARGUMENTS**:

```
define function !!lockCE()
    !!ce.lock = !!ce.lock.not()
Endfunction
```

The following is an example of a **NON-RETURN** function with **ARGUMENTS**:

```
define function !!setBoxPrimitiveSize ( !x is REAL, !y is REAL, !z is REAL )
    if !!ce.type.eq(|BOX|) then
        !!ce.xlen = !x
        !!ce.ylen = !y
        !!ce.zlen = !z
    endif
endfunction
```

Notice how both the examples use the !!CE object (a global object which represents the current element). PML procedures are typically used to interact with defined global variables

4.4 Recursive PML2 Functions

PML functions may be called recursively. This can be used to produce iterative solvers.

```
define function !!recursive( !arg1 is REAL, !arg2 is REAL)
    !var = !arg1 + !arg2
    ...
    Some PML code
    ...

    if ( condition not met ) then
        !!recursive(!var, !arg1)
    endif
endfunction
```

Recursive functions can be dangerous as they require a well-defined condition for exiting the recursion.

4.5 Making Use of Methods on PML Objects

After a variable has been defined as a specific object-type, the methods of the object will be available on the variable. Making use of these methods can help manipulate the information within a function to ensure the correct outcome.

From the example on the previous page, **!radius** is defined as a REAL object. Therefore, it has access to all the methods of a real object. The example makes use of the .power() method, and method which takes a real argument and raises the !variable to the power of the argument. The method returns the answer as a real object.

*When working with built-in objects, refer to **AVEVA™ E3D Design** Software Customisation Reference Manual for the available methods and information about them*

If you are working with different object-types, it is possible to switch been types. For example, there may be a need for a STRING to be seen as a REAL for use in an expression. The following would give an error:

```
!value = |56|
!result = !value * 2
```

As a STRING object cannot be multiplied by a REAL, an error will be returned. To avoid this error, the **.real()** method can be used to return a STRING from a REAL. Note, the original variable remains a STRING, but it is seen as a REAL for the expression:

```
!value = |56|
!result = !value.real() * 2
```

*There are equivalent methods available for the other standard objects, refer to **AVEVA™ E3D Design** Software Customisation Reference Manual.*

4.5.1 Method Information

For each object-type in the Software Customisation Reference Manual, there is a table which lists the available methods and supporting information. For each method there will be:

NAME	The name of the method or member. For example, a REAL object has a method named Cosine. If there are any arguments, they are indicated in the brackets () after the name. For example, the REAL object has a method named BETWEEN which takes two REAL arguments
RESULT	The type of value returned by the method. For example, the result of the method Cosine is a REAL value. Some methods do not return a value: these are shown as NO RESULT.
PURPOSE	This column tells you what the member or method does along with other information about the method or members.

*Note that for the system-defined **AVEVA™ E3D Design** object types, the members can be listed by using the **.attributes()** method and the methods can be listed using the **.methods()** method.*

The following are some examples of ARRAY object methods to explain the different styles:

`!numberOfNames = !nameStrings.size()`	This method returns the number of elements currently in the array. This is an example of a RESULT method with NO-AFFECT on the original object.
`!nameStrings.clear()`	This method deletes the contents of the array, but not the array. This is an example of a NO RESULT method which does AFFECT the original object.
`!newNameArray = !nameStrings.removeFrom(5,10)`	This method result removes 10 elements (starting at element 5) from the array. These elements are then returned by the method. This is an example of a RESULT method which does AFFECT the original object. If you need to remove part of the array and do not need it, then it does not need assigning to a variable

4.5.2 Method Concatenation

One of the advantages of object-orientated programming is that methods can be built-up within a single line of code. This means that complex manipulations can be made in fewer lines of PML. This process will only work if the previous method returns a suitable object for the following method. For example:

```
!line = 'hello world how are you'
!newline = !line.upcase().split().sort()
q var !line !newline


<STRING> 'hello world how are you'
<ARRAY>
    [1] <STRING> 'ARE'
    [2] <STRING> 'HELLO'
    [3] <STRING> 'HOW'
    [4] <STRING> 'WORLD'
    [5] <STRING> 'YOU'
```

In this example, the first two methods are valid for STRING objects. The **.split()** method returns an ARRAY object, so the following method has to be a valid method for an ARRAY object.

4.6 Using the !!CE Object

!!CE is a special GLOBAL PML variable that always points to the current **AVEVA™ E3D Design** element. It is a DBREF object, and its members are the attributes of the current element. Enter **q var !!CE** onto the command line and compare it against the elements attributes (Query>Attributes...). You will notice the returned attribute information is the same of the members list of the !!CE object. This means that the !!CE object can be used to assign the values of attributes to !variables. For example:

```
!branchHeadBore = !!CE.hbore
```

This assigns the HBORE attribute (taken from the current BRAN element) to the variable !BranchHeadBore making it real.

It will be necessary to check that HBORE is a valid attribute of the current element before running this line. It may cause an error.

If the !!CE object member is an object itself, that object could also have members so further information be obtained. For example, obtain the east coordinate of a head of a BRAN element, either of the two following can be used:

```
!headPosition = !!CE.hpos
!headEasting  = !!CE.hpos.east
```

or:

```
!headEasting = !headPosition.east
```

If the !!CE object member is an object with built-in methods, then these methods can also be called:

```
!BranPosWRTZone = !!CE.hpos.wrt(ZONE)
```

returns a POSITION object w.r.t the owning ZONE.

This process can also be reversed, and values can be applied to the attributes of the !!CE. This means that it is possible to record the current value of an attribute, modify and reassign back to the CE. For example, enter out the following onto the command line:

```
Q POS
!position = !!CE.pos
!position.up = !position.up + 2000mm
!!CE.pos = !position
Q POS
```

These lines will have moved the CE up by 2000mm. This same logic can be applied to other attributes, providing the object-type is considered.

Exercise 5 Convert a Macro into a PML Procedure

Perform the following tasks:

- In C:\Users\Public\Documents\AVEVA\Training3.0\USERDATA\PMLEX directory find the

  file **ex5.mac**, a PML 1 macro file which builds a Hose Reel as a piece of equipment.
- Debug the existing macro and convert it into a new PML procedure with NO ARGUMENTS.
- Save the new file as ex6.pmlfnc under the PMLEX search path.
- Update the PMLEX search path by entering **PML REHASH ALL**
- Test the function in the command window by entering **!!ex5()**
- Update the procedure and give it TWO ARGUMENTS so it becomes a function. The two arguments should be:
  - The name of the EQUIPMENT
  - The diameter of the Hose Reel.
- Check the updated function to make sure the Hose Reel is created as expected. Consider error handling while you check the function.
- Save the modified procedure as another PML function file with a suitable name.

An example of the completed procedure can be found in Training Assistant

5 Collections

A very powerful feature of the **AVEVA™ E3D Design** database is the ability to collect and evaluate data according to rules. There are two available methods for collection that are valid in PML. The first is a command syntax PML 1 style and the other is with a PML COLLECTION object. Both are still valid, but when developing new PML, a COLLECTION object should be used in preference.

5.1 COLLECT Command Syntax (PML 1 Style)

The **COLLECT** syntax is based around three specific pieces of information:

- What element type is required?
- If specific elements are required, what is different about them?
- Which part of the hierarchy to look in?

If you wish to collect all the EQUI elements for the current ZONE, enter the following:

```
var !equipment collect all EQUI for ZONE
!equipment = !!CollectAllFor('EQUI',",ZONE)
q var !equipment
```

If you wish to collect all the piping components owned by a specific BRAN, enter the following:

```
var !pipeComponents collect all with owner eq /200-B-4/B1 for SITE
!pipeComponents = !!CollectALLFor(",'owner eq /200-B-4/B1',SITE)
q var !pipeComponents
```

If you wish to collect all the BOX primitives below the current element, enter the following:

```
var !boxes collect all BOX for CE
!boxes = !!CollectAllFor('BOX',",CE)
q var !boxes
```

You do not need to specify level of the hierarchy to search within i.e. the FOR.

5.2 EVALUATE command syntax (PML 1 style)

After elements have been collected through the **COLLECT** syntax, they are stored as an ARRAY. This array (like any array) can be processed using the **EVALUATE** syntax to provide further information.

To get the names of all the elements held in !equipment, enter the following:

```
var !equipmentNames evaluate NAME for ALL from !equipment
q var !equipmentNames
```

To get the full names of the elbows held within !pipeComponents, enter the following:

```
var !elbows evaluate FLNN for all ELBO from !pipeComponents
q var !elbows
```

To get the names (without the leading slash) of the pumps in !equipment, enter the following:

```
var !pumps evaluate NAMN for ALL with MATCHWILD( NAMN, |P*|) from !equipment
q var !pumps
```

To get the volume of all the boxes in !boxes, enter the following:

```
var !volume eval ( xlen * ylen * zlen ) for ALL from !boxes
q var !volume
```

To get a list of RTEXT for all pipe components within zone /ZONE-PIPING-AREA01 with Nominal size (PARAM[1] OF CATREF) equal to 25, enter the following:

```
 var !flRdet eval (DTXR) for all with (ATTRIB PARA[1] of CATREF eq 25) for /ZONE-PIPING-
AREA01
q var !flRdet
```

5.3 COLLECTION Object (PML 2 Style)

A PML2 object has replaced an old PML1 COLLECT ALL command syntax. An advantage of using a COLLECTION object is that an ARRAY OF DBREF objects is returned. It is recommended to use PML2 objects instead of PML1 commands when required.

A COLLECTION object is assigned to a variable in the same way as other objects:

```
!collection = object COLLECTION()
q var !collection
q var !collection.methods()
```

Once assigned, the methods on the object are used to set up the parameters of the collection. To set the element type for the collection to EQUI elements only, enter the following:

```
!collection.type(|EQUI|)
```

If more than one element type is required, the following could be entered.

```
!elementTypes = |EQUI BRAN SCTN|
!collection.types(!elementTypes.split())
```

The scope of the collection must be a DBREF object and should be passed as an argument to the **.scope()** method. For example, enter the following to set the scope as the current element:

```
!collection.scope(!!ce)
```

To filter the results, the **.filter()** method must be passed an EXPRESSION object. This means that the process is in two steps:

```
!expression = object EXPRESSION(|name of owner eq '/200-B-4/B1'|)
!collection.filter(!expression)
```

Once the collection object has been set up, the .results() is used to return the collected elements as an ARRAY of DBREF objects.

```
!results = !collection.results()
q var !results
```

5.4 Evaluating the Results from a COLLECTION Object

After an ARRAY of DBREF objects has been generated from a COLLECTION object, the entire array can be evaluated using the **.evaluate()** method on an array object. The argument for the **.evaluate()** method must be a **BLOCK** object defined with the expression that needs evaluating.

To get an ARRAY of STRINGs holding the full names of the collected elements, enter the following:

```
!block = object BLOCK(|!results[!evalIndex].flnn|)
!resultNames = !results.evaluate(!block)
q var !resultNames
```

Notice how the BLOCK object uses the local variable !evalIndex. This variable effectively allows the .evaluate() method to loop through the ARRAY. To get the positions of the collected elements, enter the following:

```
!resultPos = !results.evaluate(object BLOCK(|!results[!evalIndex].pos|))
q var !resultPos
```

Instead of defining the block as a separate variable, this second example shows that the object can be defined within the argument to another object.

Notice how the evaluated ARRAY contains the correct object types generated from the evaluation

Exercise 6 Change of Pipe Name

Perform the following tasks:

- Find the Macro file in C:\Users\Public\Documents\AVEVA\Training3.0\USERDATA\PMLEX\ex6.mac which can change the name of all Pipes available in /SITE-PIPING-AREA01.
- The name of the pipe should be change to :-

  <Pipe Diameter> - <Pipe Spec> - 'ZONE_AREA01' - <Pipe Temperature>

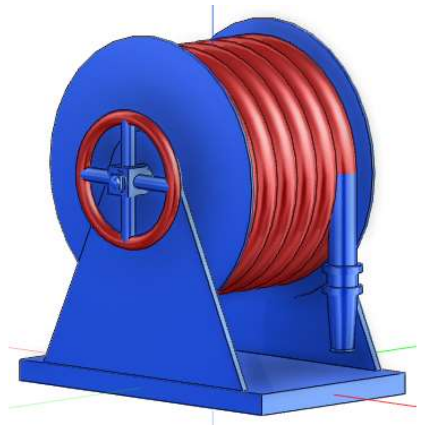
  For Example :- **100-C-12 -> 100mm-/F1C-Zone_AREA01-50**
- Collect all the details of Pipe available in /ZONE-PIPING-AREA01.
- Apply the Do Loop condition which will allow to repeat the process for all pipes.
- Start to get the details about the Pipe Diameter, Pipe Temperature and convert the values to the String.
- Also get the details about the Pipe Specification (Pspec) and Owner details.

Exercise 7 Highlight Element using Collection

Perform the following tasks:

- Create a PML function that will apply colour to an element type below the current element.
- Write the function to accept two arguments:
    - The element type to be highlighted – STRING.
    - The highlight colour to use – REAL.
- Try to use the COLLECTION object with user preference.
- Use the ENHANCE syntax to highlight the elements.

The PML2 way of highlighting elements is discussed in TG1031-CC0-C01



- Save the new file as ex7.pmlfnc under the PMLEX path.
- Update the PMLLIB search path by entering **PML REHASH ALL**
- Test the function in the command window. For example, the following function call **!!ex7(|CTOR|, 2)** would highlight the hose wheel as shown.

- Update the function so that it returns an array of the collected elements.
- Test the function by querying the size of the returned array on the command line.
- For example: **q var !!ex7(|CTOR|, 2).size()** <REAL> 16

An example of the completed procedure can be found in Training Assistant

This page is intentionally left blank.

CHAPTER 6

6 Miscellaneous

6.1 Error Tracing

It is possible to list all the commands that **AVEVA™ E3D Design** runs when a PML operation is carried out. This list can either be printed to the command window or written to an external file.

Enter **$r109 /c:\temp\trace.log** onto the command line

Where **c:\temp\trace.log** is the output file name (it shall be created if it does not exist)

Now every PML action will be traced to the external file. The file will list every line of code and action carried out. Lines read will be indicated by a line number in square brackets. Lines not read will be indicated by a line number between round brackets. Entry and exit points between methods, functions and objects are indicated as well as any errors.

Enter **$r110** onto the command line to print the same lines to the command window.

Printing to the screen should only be done when a small number of lines are expected. The command window may run out of lines to display the information

Once you are finished with error tracing, enter **$r** to the command line. If the file is not closed, information will continue to be added to it.

This will need to be before the output file can be opened

To find out more information about the $r command, enter $HR into the command window.

6.2 PML Encryption

To encrypt the pml files before sharing them can be done using PML Publisher. Once encrypted, the files can still be used with any compatible AVEVA program, but it would be impossible to read and modify them through a text editor. Encrypted files may be used without additional license, but the encryption utility described below is separately distributed and licensed.

Once encrypted, PML cannot be decrypted. A reference copy of all PML should be kept safe.

The default encryption is implemented using the Microsoft Base Cryptographic Provider, a general-purpose cryptographic service provider (CSP) which is included in, among other operating systems. There is a limit to the strength of this encryption and is not secure against all possible attacks. AVEVA may release improved algorithms with future releases of the product. If this happens, encrypted files will require re-encrypting.

The encrypted example of the Pml form **!!nameCE** would be created as file shown below:

```
--<004>-- Published PML 1.0 >--
return error 99 'Unable to decrypt file in this software version'
$** abdfe19b3008494b6399edda08b66004
$** MR+zhtg-egE2Ig9IiHSVmdPo08ChKexa7wbfcyODTbfjTFWU02pK3v4sXq5i
$** TKW3dEFRJCd60uSzaLXdc5fvLeOKqXO71uFlZ1vEsIOOHvq8viAwiys4rGXg
$** XLgFFVG7mpsmnFtrQDN3o51aiAgicFS6u08C7r8IaxUTUQAOdXeBmlp4TLXc
$** 9KR5LtAIugLrC9a7NxbF+0Hn-c5tOhUAEBG
```

Reading an encrypted file is slower than reading a decrypted on. Making use of the Buffer argument can help. Refer to the PML Publisher User Guide for more information.

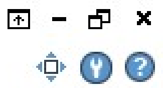
6.3 General Notes on PML

- PML functions and methods run through their definitions in line order (control logic can be used to alter this order)
- Functions should be written in preference to macros.
- Variables are instances of object definitions.
- The **RETURN** command can be used to exit a function/method as well as return a value.
- PML files are ASCII and can be created/edited in any basic text editor.
- PML 1 files are saved under **PMLUI** folder and PML 2 under the **PMLLIB** folder.
- PML 2 objects have specific file extensions (**.pmlfnc**, **.pmlobj** and **.pmlfrm**).
- When new file created content of PML library (PMLLIB) must be reindexed to make new file been available. This can be done by entering **PML REHASH ALL c**ommand.
- Once loaded, objects can be reloaded by entering **PML RELOAD OBJECT**.
- Once loaded, forms can be reloaded by entering **PML RELOAD FORM**.
- When declaring a string, text delimiters must be used. Either single quotes **'…'** or vertical bars **|…|**.
- File paths of files can be obtained by using **!!pml.getPathName(|form.pmlfrm|)**.
- The **$** is used as an escape character and can be used to expand a variable before it is read as command syntax. If a $ symbol is actually required, then two must be entered.
- To provide comments in statements with PML, one of the following can be used.
  - For a comment line, start the line with two hyphens **–**
  - For a comment at the end of a line, start the comment with **$***
  - For a block comment, start with a **$(** and end with a **$)**
- Variable names are not case sensitive

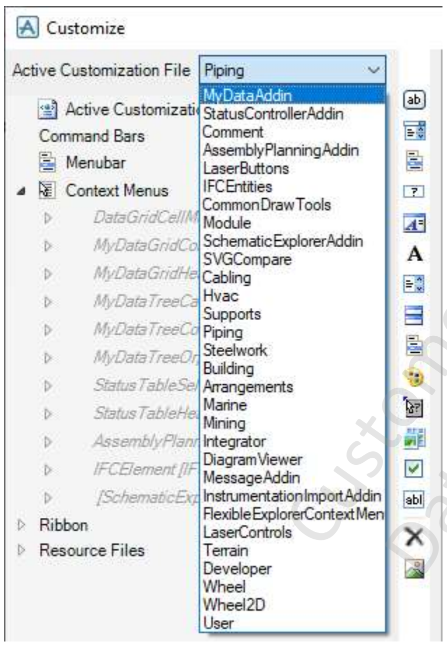
- String comparisons are case sensitive
- Variable names can be 16 characters long and should not start with a number or contain a dot
- It is good practice to name variables in lower case, using upper case to separate words e.g. !stringLength
- It is possible to abbreviate some command syntax. The required part is shown in upper case with the syntax graphs e.g. **WIDTH** means that **WID** is acceptable when declaring width.

6.4 Menu Additions

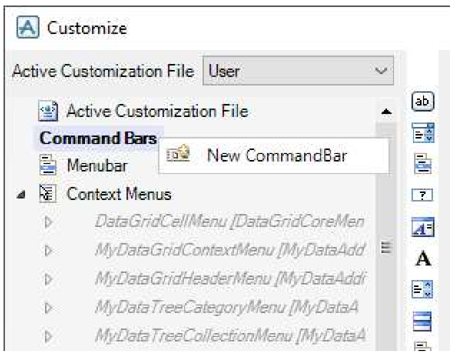
It is possible to define the user defined command bars to run the custom macros:



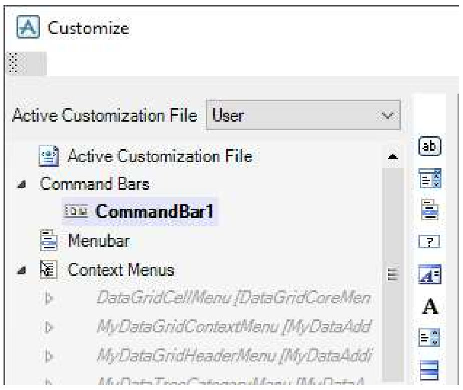
- On the Tab area, click on the **Customize** button to display the Customize form.



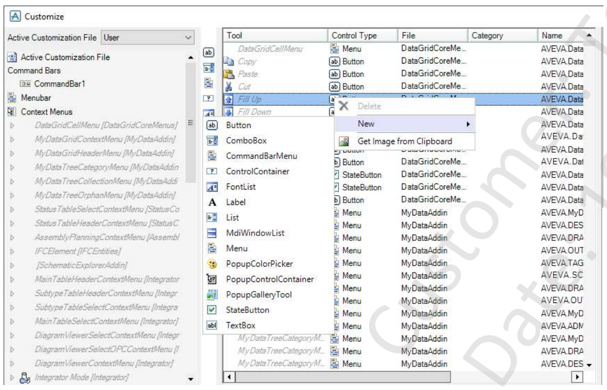
- From the Customize form **Active Customization File** menu, select **User**
- This will save a changes to a specific user file (file path given on the right hand side of the form)


Active Customization File: User
Customize
Active Customization File
Command Bars
New CommandBar
Menubar
Context Menus
DataGridCellMenu [DataGridCoreMen
MyDataGridContextMenu [MyDataAdd
MyDataGridHeaderMenu [MyDataAddi
MyDataTreeCategoryMenu [MyDataA
MyDataTreeCollectionMenu [MyDataA

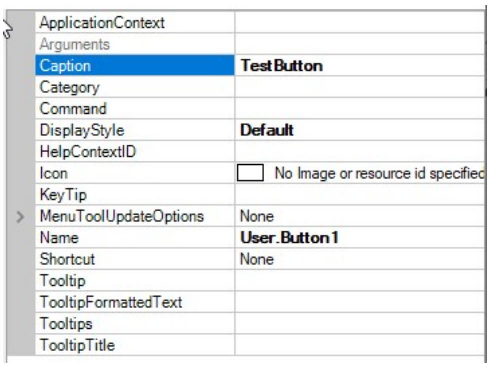
- Right click on the Command Bars heading in bottom window and click on **New Command Bar**.


Active Customization File: User
Customize
Active Customization File
Command Bars
CommandBar1
Menubar
Context Menus
DataGridCellMenu [DataGridCoreMen
MyDataGridContextMenu [MyDataAdd
MyDataGridHeaderMenu [MyDataAddi

- You will now see a preview of the command bar in the top left of the form (when the new Command Bar is selected)

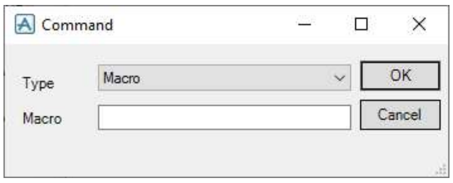


- Right click in the middle window in the form
- Create a **New > Button** from the menu



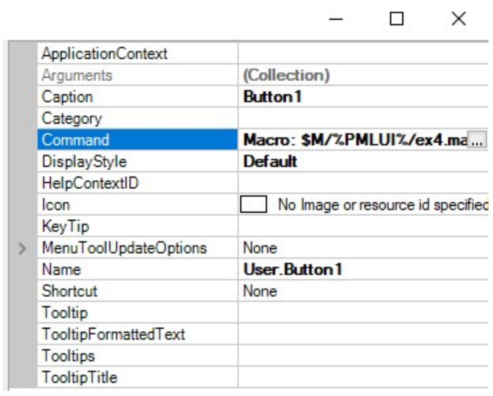
ApplicationContext	
Arguments	
Caption	**Test Button**
Category	
Command	
DisplayStyle	**Default**
HelpContextID	
Icon	No Image or resource id specified
KeyTip	
MenuToolUpdateOptions	None
Name	**User.Button1**
Shortcut	None
Tooltip	
TooltipFormattedText	
Tooltips	
TooltipTitle	

- With the new button selected in the middle window, look to the right of the form
- Select the line titled Command and click the small "…" button which appears on the right. This will let us set the command.



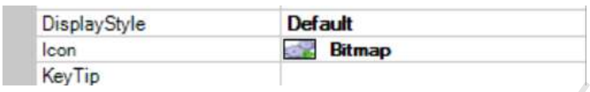
Type	Macro
Macro	

- A command can be a core command, command class or a PML command.
- Choose **Macro**, and type for example **$M/%PMLUI%/ex4.mac**
- Close this form by pressing OK



ApplicationContext	
Arguments	(Collection)
Caption	Button1
Category	
Command	Macro: $M/%PMLUI%/ex4.ma...
DisplayStyle	Default
HelpContextID	
Icon	No Image or resource id specifiec
KeyTip	
MenuToolUpdateOptions	None
Name	User.Button1
Shortcut	None
Tooltip	
TooltipFormattedText	
Tooltips	
TooltipTitle	

- The associated command is now set
- Select the line titled Icon. Click the **"…"** button that appears to the right
- Browse for the file *.png
- Click **Open**

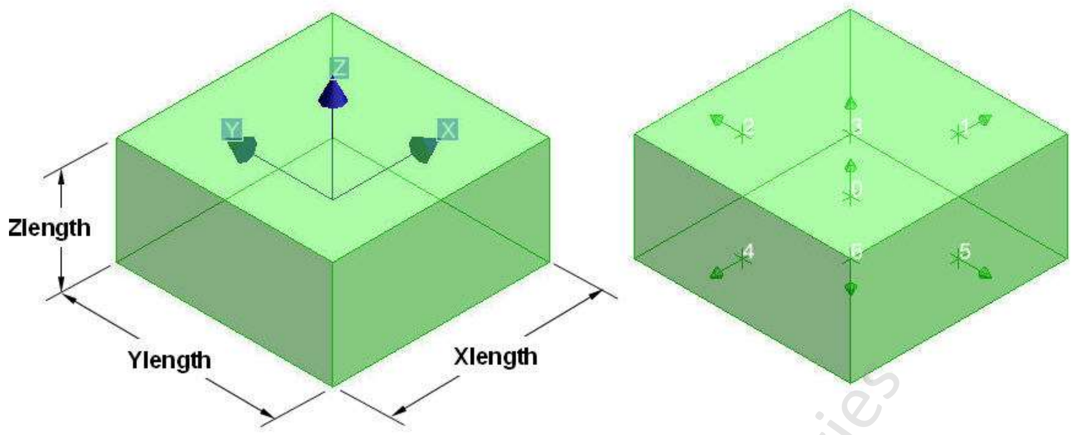


DisplayStyle	Default
Icon	Bitmap
KeyTip	

- The icon is now displayed in the right hand window.
- Set the tooltip to 'Show Nozzle Checker'
- Drag&Drop TestButton in to CommandBar1
- Select the new button in the middle window.
- Drag and Drop the new button beneath the new CommandBar object to make the association.
- Select the CommandBar object and notice the button is displayed in the preview.
- Click **Apply** and **OK** to close the form
- A new Command Bar will be displayed in the main application.
- Test the button

Appendix A – AVEVA™ E3D Design Primitives

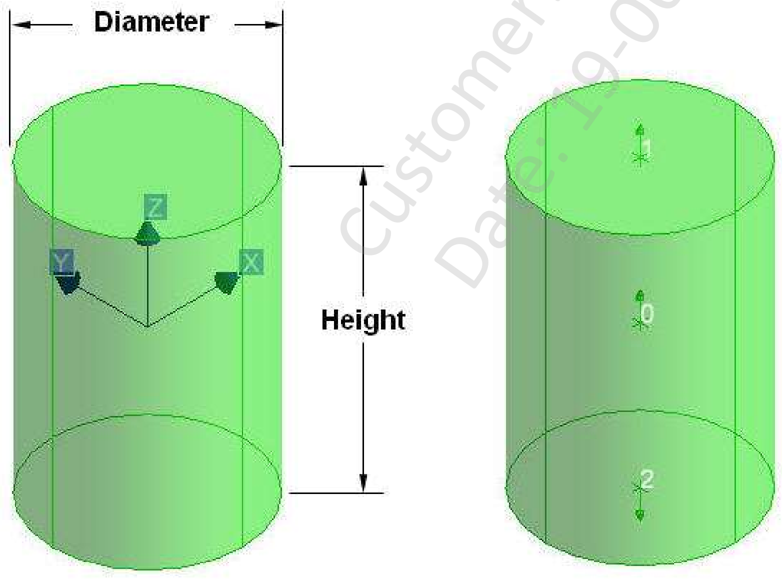
- **Box (BOX)**



Specific geometric attributes:

Xlength	Length parallel to X axis
Ylength	Length parallel to Y axis
Zlength	Length parallel to Z axis

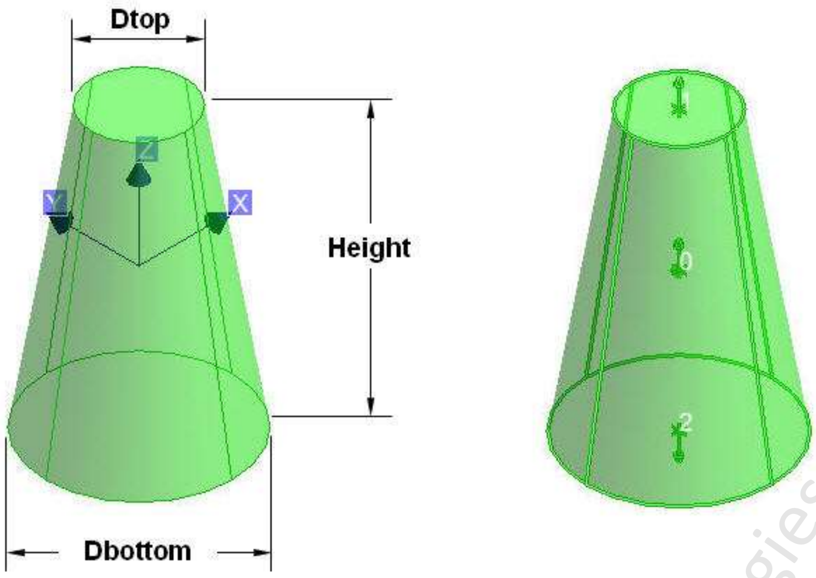
- **Cylinder (CYLI)**



Specific geometric attributes:

Diameter	Diameter of cylinder
Height	Length parallel to Z axis

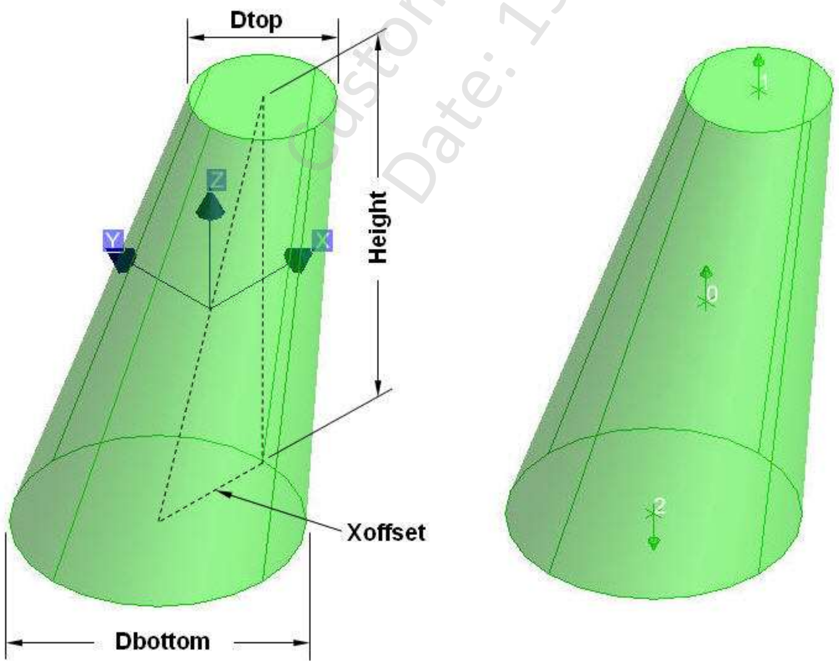
- **Cone (CONE)**



Specific geometric attributes:

Dtop	Diameter at top of cone
Dbottom	Diameter at bottom of cone
Height	Length parallel to Z axis

- **Snout (SNOU)**

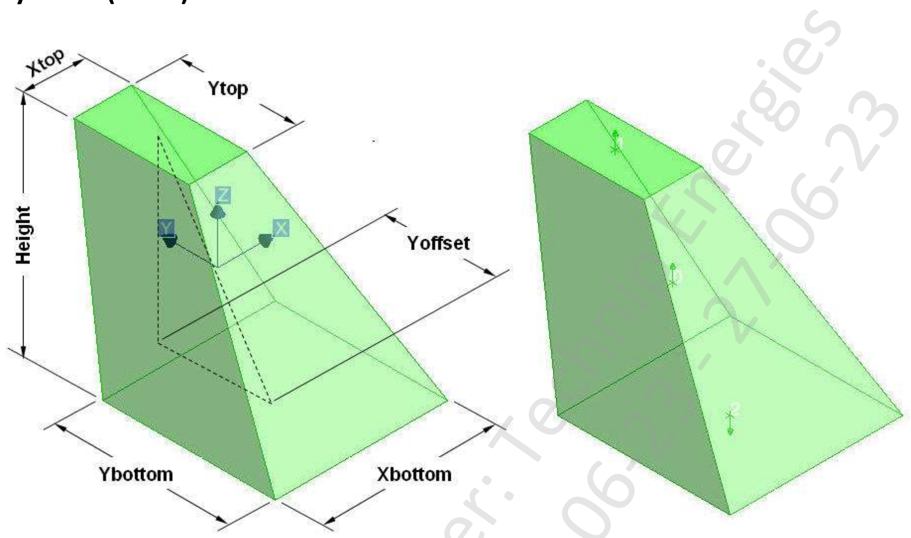


Specific geometric attributes:

Dtop	Diameter at top of snout
Dbottom	Diameter at bottom of snout
Xoffset	Offset of centre of top from centre of bottom on X axis
Yoffest	Offset of centre of top from centre of bottom on Y axis
Height	Length parallel to Z axis

Only an Xoffset is show in this example, however, both Yoffset and Xoffset may be set

- **Pyramid (PYRA)**

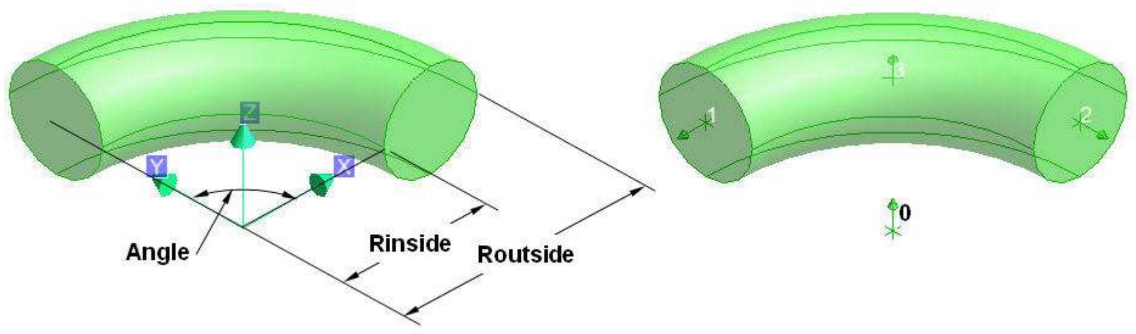


Specific geometric attributes:

Xbottom	Length of bottom of pyramid parallel to X axis
Ybottom	Length of bottom of pyramid parallel to Y axis
Xtop	Length of top of pyramid parallel to X axis
Ytop	Length of top of pyramid parallel to Y axis
Height	Length parallel to Z axis
Xoffset	Offset of centre of top from centre of bottom on X axis
Yoffset	Offset of centre of top from centre of bottom on Y axis

Only a Yoffset is show in this example, however, both Yoffset and Xoffset may be set.

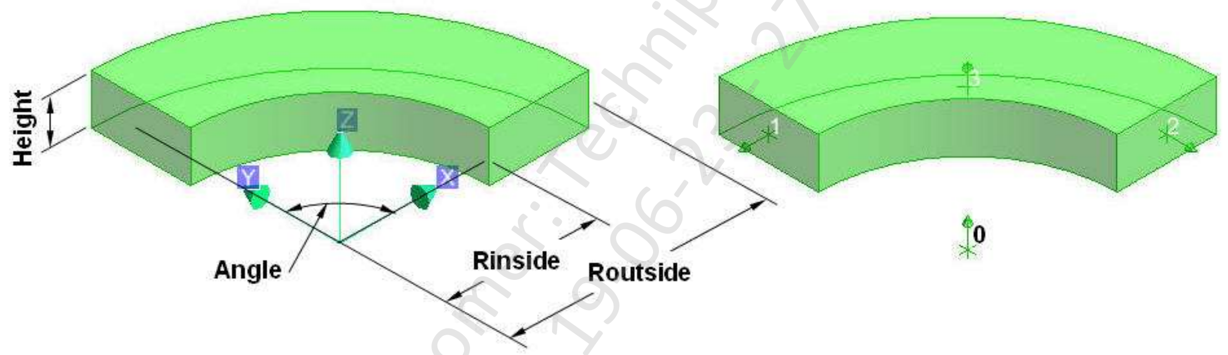
- **Circular Torus (CTOR)**



Specific geometric attributes:

Rinside	Inside radius in XY plane
Routside	Outside radius in XY plane
Angle	Subtended angle (maximum 180°)

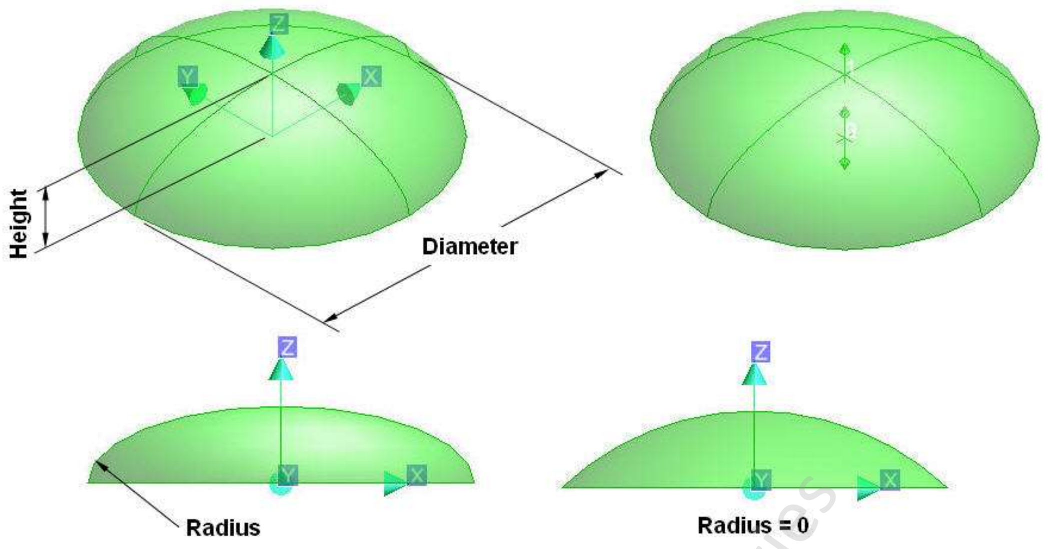
- **Rectangular Torus (RTOR)**



Specific geometric attributes:

Rinside	Inside radius in XY plane
Routside	Outside radius in XY plane
Height	Length parallel to Z axis
Angle	Subtended angle (maximum 180°)

- **Dish (DISH)**

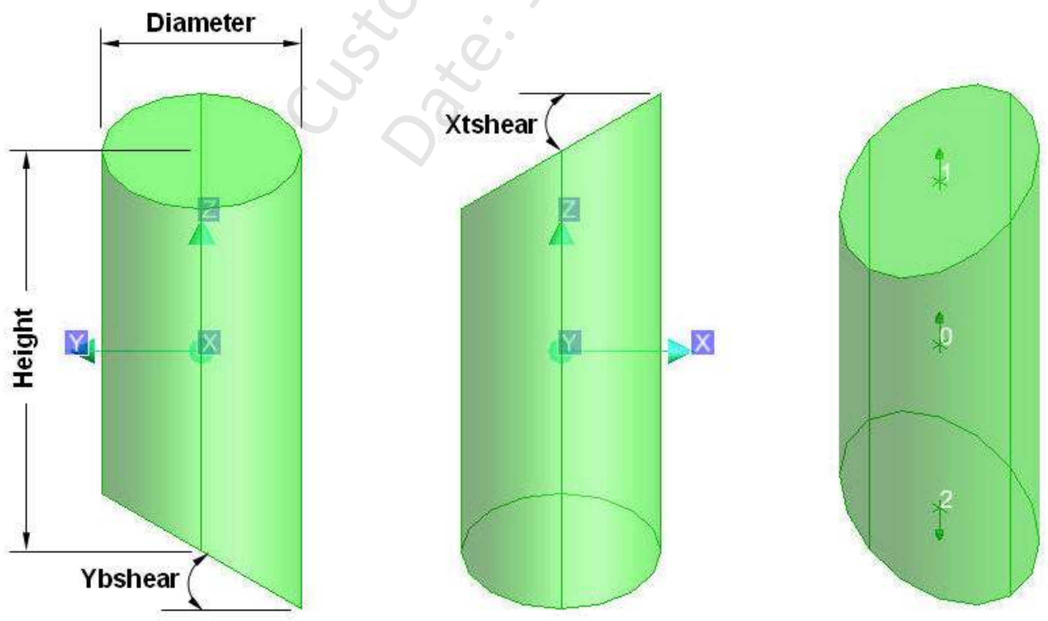


Specific geometric attributes:

Diameter	Diameter of dish in XY plane
Height	Height of dish parallel to Z axis
Radius	Knuckle radius

If the knuckle radius is 0 then the dish is represented as a segment of a sphere. If the knuckle radius is greater than 0 then the dish is represented as a partial ellipsoid, generally used to represent a torispherical end to a vessel.

- **Sloped Cylinder (SCYL)**

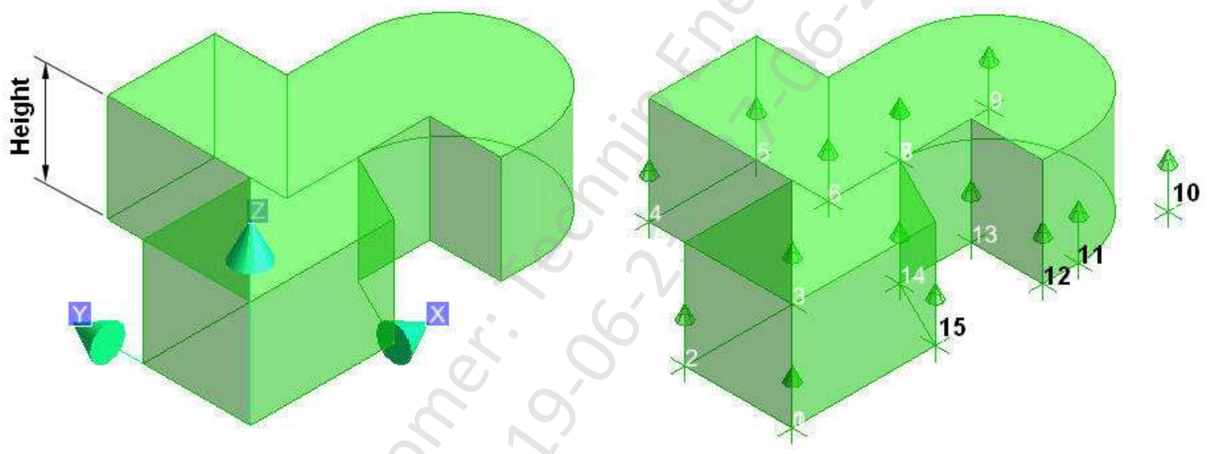


Specific geometric attributes:

Diameter	Diameter of sloped cylinder
Height	Length in Z axis from bottom centre to top centre
Xtshear	Inclination of top of cylinder in the XZ axis (in degrees)
Ytshear	Inclination of top of cylinder in the YZ axis (in degrees)
Xbshear	Inclination of bottom of cylinder in the XZ axis (in degrees)
Ybshear	Inclination of top of cylinder in the YZ axis (in degrees)

Only an Xtshear and Ybshear are shown in this example, however, Xtshear, Ytshear, Xbshear and Ybshear may be set in any combination to obtain the required results. The values for these attributes may be +ve or –ve.

- **Extrusion (EXTR)**

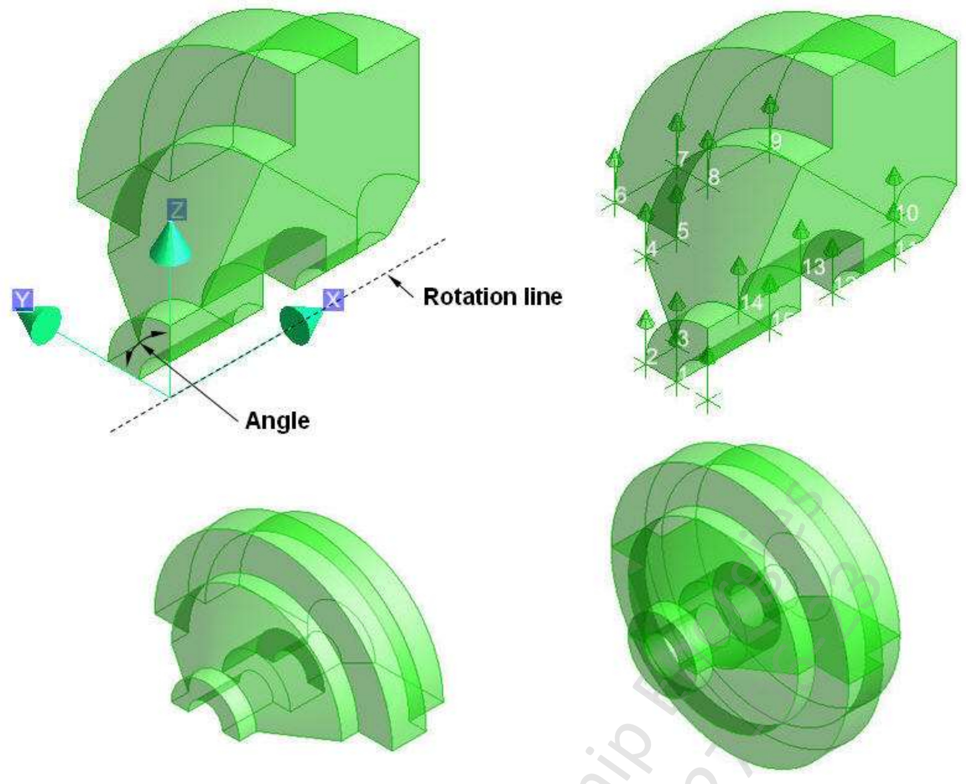


Specific geometric attributes:

Height - Height of extrusion in Z axis.

An extrusion is a 2D shape, defined by a series of vertices at each change in direction, extruded through a height. The primitive consists of three element types, i.e. EXTR, LOOP and VERTs.

- **Solid of Revolution (REVO)**



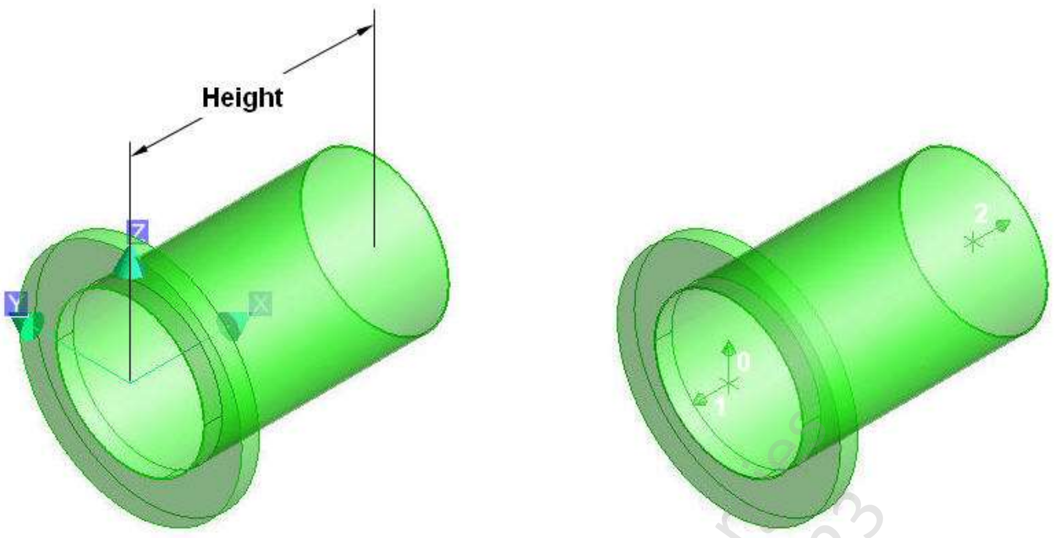
180° Rotation   360° Rotation

Specific geometric attributes:

Angle	Rotation angle around X axis (selected rotation line)

A solid of revolution is a 2D shape, defined by a series of vertices at each change in direction, rotated through a specified angle around a specified rotation axis. The primitive consists of three element types, i.e. REVO, LOOP and VERTs.

- **Nozzle (NOZZ)**



Although a nozzle is classed as a primitive, it is unlike the other primitives in that its geometry is determined in Paragon as part of a catalogue component. Nozzles of different types and geometry may be constructed in Paragon to suit the requirements of the Piping Specification.

The specific nozzle type is referenced from Paragon using the Spref (Specification Reference) attribute.

Specific geometric attributes:

Height - Height between nozzle face and end, i.e. from P1 to P2.