

# Current version TC\_14.1\_50.0

## Parametric Part Manager (Parametric Symbol Editor)

1.	Introduction .....	1
2.	Script syntax .....	2
3.	Script semantics .....	3
4.	Basic functions .....	3
4.1.	Description of parameters.....	4
4.2.	Functions for creation of 2D entities .....	4
4.3.	Functions for creation of 3D entities on the basis of 2D objects.....	6
4.4.	Functions for loading external symbols as elements .....	6
4.5.	Functions for transformations of geometric objects .....	7
4.6.	Functions for Boolean operations.....	7
4.7.	Auxiliary functions .....	8
4.8.	Setting and changing object properties.....	8
5.	Creating custom functions .....	9

### 1. Introduction

Parametric symbols (PSM) are defined using a text description (script).  
Text file with a description of a parametric symbol must have \*.psm extension.  
The name of the text file determines the name of the symbol.

A simple example of defining a parametric symbol is a rectangle, width, height and rotation angle of which are defined through parameters. Description of such symbol will look as follows:

File **rect.psm**:

```
// Here is a description of simple rectangle.  
H = Parameter("Height", 5, LINEAR, Interval(0, 100));  
L = Parameter("Length", 10, LINEAR, Interval(0, 200));  
Angle = Parameter("Angle", 0, ANGULAR, Interval(0, 360));  
Rect1 = Rectangle(H, L);  
Rect = RotateZ(Rect1, Angle);  
Output(Rect);
```

Let's examine each line of this example.

First line, which starts with "//", contains only a text comment for the example, and it does not affect behavior of the symbol. "//" characters mean that all text, starting with this position and up to the end of the line, should be considered a comment.

Second line defines characteristics of H parameter. To clarify, let's spell this line out as an equivalent sequence of lines with comments:

```
H =           // H – identifier (name) of the parameter in the symbol description,  
Parameter(   // Parameter – function that defines parameter characteristics  
    "Height", // User will see this name in the Properties dialog  
    5,        // 5 - Default parameter value  
    LINEAR,   // indicates that H parameter defines a linear value  
    Interval(0, 100) // Defines possible variation interval for the H parameter  
);           // End of Parameter function and entire description for H.
```

The next two lines in the example are similar to the previous one. They define the characteristics of L and Angle parameters.

The line `«Rect1 = Rectangle(H, L);»` defines a rectangle with H height and L length, the center of which lies at the origin of coordinates.

The line `«Rect = RotateZ(Rect1, Angle);»` defines Rect rectangle, which is rotated relative Rect1 rectangle by Angle angle.

The last line in the example shows that Rect object is the main (last) in the symbol description.

## 2. Script syntax

Description of a parametric symbol consists of the entire contents of a text file, except comments, tabs, and other control characters, which are ignored.

Comments are specified either using “//” characters that mean that all subsequent characters up to the end of the line are comments, or using the pair “/\*” and “\*/” that denote beginning and end of the comment, respectively.

Text description is a set of two types of operators:

`<Identifier> = <Expression>;`

and

`<Expression>;`

*Identifier* defines the symbolic name of an object. It is a set of Roman letters and Arabic numerals, which starts with a letter.

For example: PART2a.

Object identifiers should not be the same as names of functions or such names as PI, LINEAR, ANGULAR, that are used to designate the constants of the scripting language. The list of all reserved names is provided in Appendix 1.

Expression, in the first case, defines the dependence of the defined object on other objects. Expression syntax matches the expression syntax in the majority of programming languages. Expressions can contain function calls

`<Function name>( <list of parameters> ),`

arithmetic operations “+”, “-”, “\*”, “/” and parenthesis “(“ and “)”, that determine the sequence of performing arithmetic operations. Object identifiers and numbers serve as expression operands.

Examples of correct expression syntax:

```
(D -1/4) * k  
Polyline(Point(0, 0.25 - 1/8), Point(0, D), Arc1(L-C, - m, m), Point(0,0));
```

### 3. Script semantics

A script contains full description of a parametric symbol. Collection of script operators determines which actions need to be performed to create the resultant object(s).

Correct understanding of a script, which is requisite for creation of new scripts, requires having a clear understanding of how its operators are interpreted.

The list of resultant objects is defined in *Output(..)* operator. This operator must be present in the script.

Each identifier in the list of arguments for *Output* function should be defined with an operator of the type

$\langle \text{Identifier} \rangle = \langle \text{Expression} \rangle;$

which determines the method that will be used to create an object with this name.

In their turn, identifiers that are used in the *Expression* should also be defined with operators of this type.

A correct script describing a parametric symbol should conform to the following rules:

1. Script must contain exactly one *Output(..)* operator.
2. For each object identifier used in the *Output(..)* operator, and/or used in the right part of  $\langle \text{Identifier} \rangle = \langle \text{Expression} \rangle$  operators there should be exactly one “own” operator of  $\langle \text{Identifier} \rangle = \langle \text{Expression} \rangle$  type where this identifier is in the left part. That way, each identifier should be used at least once in the right part and once – in the left.
3. Script operators of  $\langle \text{Identifier} \rangle = \langle \text{Expression} \rangle$  type determine the dependency of identifiers (object) on each other. The identifier, which is in the left part, depends on the identifiers used in the right part of the operator. The script must not contain interdependent identifiers, i.e. there should not be situations where because of one or more operators I1 identifier depends on I2, but because of other operators I2 depends on I1.

The following situation

```
A = B + 0.5;  
B = sin(A);
```

is not allowed because in this case A and B identifiers are interdependent.

The sequence of script operators is not important (except certain special cases that will be described later), because operators are sorted before the script is run.

### 4. Basic functions

Probably the most significant advantage of this method of creating parametric symbols is the compact size and clarity of text description of parametric symbols in script form. The set of basic functions used in such description, determines the level of clarity and simplicity of scripts for a particular class of parametric symbols.

It is intended to expand the set of basic functions from version to version.

## 4.1. Description of parameters

Description of external parameters for symbols is done using Parameter function.

`<id> = Parameter(<name>, <default value>, <type>[, <condition1>][, <condition2>]..);`

where

`<name>` - the line, which contains parameter name displayed in the user interface;  
`<default value>` - default value of the parameter;  
`<type>` - defines the parameter type. The following values are possible:  
LINEAR means that the parameter is a linear value,  
ANGULAR means that the parameter is an angular value.

Other `<condition>` parameters are optional. They define possible restrictions imposed on parameters. Restrictions can be listed in arbitrary order and may take on the following forms:

`Set(<value>,..)` – a list of permissible values of the parameter

`Interval(<minvalue>, <maxvalue>)` – sets the minimum and maximum values of the parameter;

`LessThan(<value>)` – indicates that parameter value should be less than the specified value;

`LessOrEqual(<value>)`– indicates that parameter value should not be greater than the specified value;

`GreaterThan(<value>)` – indicates that parameter value should be greater than the specified value;

`GreaterOrEqual(<value>)`– indicates that parameter value should not be smaller than the specified value.

Restrictions should not contradict each other.

When specifying parameter restrictions, it is not allowed to use identifiers or expressions that directly or indirectly depend on other parameters, as arguments of the above-mentioned functions. Only constants or constant expressions can be used, for example: `LessOrEqual(PI/2)`.

Example of parameter description:

```
Alpha = Parameter("Rotation Angle", 45, ANGULAR, Interval(-90, 90));
```

## 4.2. Functions for creation of 2D entities

`circle` function is used to create circles.

`Circle(<radius>[, <cx>, <cy>])`

`<radius>` - circle radius.

`<cx>, <cy>` - optional arguments that set (x, y) coordinates of the circle center. By default, `cx = 0, cy = 0`;

Example:

```
C = Circle(D/2, 0, y0);
```

**Rectangle** function is used to create rectangles.

```
Rectangle(<width>, <height>[, <cx>, <cy>])
```

<width> - a number, rectangle width

<height> - a number, rectangle height

<cx>, <cy> - option arguments that set (x, y) coordinates of the rectangle center. By default, cx = 0, cy = 0;

Example:

```
rect = Rectangle(W, H, W/2, H/2); // Left bottom corner is in (0,0) point
```

**Polyline** function is used to create polylines consisting of straight line segments and arc segments.

Polyline(<list of arguments>)

Here, <list of arguments> is the list of arguments, delimited with commas. Arguments define individual segments of a polyline.

For polylines that contain only straight line segments, the <list of arguments> consists of only 2D points, defined using *Point(x,y)* function. For example, a rectangle can be defined in the following way:

```
rect = Polyline(  
Point(0,0),  
Point(W, 0),  
Point(W,H),  
Point(0, H),  
Point(0,0) );
```

It should be noted that a polyline, first and last points of which are coincident, is called a closed polyline. Such polyline bounds a certain area.

Polylines with arc segments are defined by adding auxiliary functions Arc0 and Arc1 to the list of arguments. Arc0 builds the circular arc clockwise, while Arc1 – counterclockwise. Functions have the following format:

```
Arc0 (<cx>, <cy>),  
Arc1 (<cx>, <cy>),
```

where <cx>, <cy> set (x,y) coordinates of the arc center.

Start and end point of an arc are defined by the preceding and the following arguments.

Arc0 and Arc1 cannot be the first or last argument in the list of arguments.

Another method of creating an arc is to use auxiliary function Fillet, which “smoothes” two linear segments that start and end in the preceding point, by adding an arc with the specified radius into the corner. This ensures smoothness at the junction points. Format of the function:

```
Fillet (<radius>).
```

Example:

```

Poly1 = Polyline( // Rectangle with rounded corners
Point(0,0),
Pont(W - r, 0), Arc1(W - r, r), Pont(W, r),
Point(W, H - r), Arc1(W - r, H - r), Point(W - r ,H),
Point(0, H), Fillet(r),
Point(0,0), Fillet(r) );

```

### 4.3. Functions for creation of 3D entities on the basis of 2D objects

**Thickness** function creates a 3D object based on the 2D one by adding thickness. It also allows changing thickness property of the 3D object.

Format: Thickness(<Object>, <value>),  
where <Object> - initial graphic object  
<value> - new value of Thickness

**Sweep** function creates a 3D object by “dragging” a specified profile along a path, defined by a 2D polyline. The profile is specified by a section, which is defined by a closed 2D polyline.

The function has the following format:

```
Sweep(<profile>, <path>[,<rotation angle>])
```

where

<profile> - profile defined by a 2D polyline;

<path> - path, long which the profile is “dragged”; the path is defined by a 2D polyline;

<rotation angle> - optional argument, which defines the rotation angle of the profile relative the Z axis; by default, the argument is equal to zero.

### 4.4. Functions for loading external symbols as elements

**StaticSymbol** function is used to load non-parametric symbols from external files, which the CAD system can “understand”.

Function format:

```
StaticSymbol(<FileName>[,BlockName])
```

<FileName> - line, which contains file name with extension. If the extension is not specified, native file format will be used;

<BlockName>- optional argument, which indicates that only the block with the given name should be used as the symbol for loading, and the rest of the contents should be ignored; if the argument is not defined, the active drawing will be loaded as a symbol.

Creating a list of files in a folder. Typical use of Set(FolderList(...))

```
<id> = FolderList(<path>, <mask> = "*.psm")
```

<path> - line, path to the folder to create a list of files.

<mask> - line, file mask.

Loading of a parametric symbol (a file with \*.psm extension) is done by calling a function, where function name is the name of the file, and arguments are parameters of the symbol to be loaded, in the order in which they are described in the file.

## 4.5. Functions for transformation of geometric objects

This class of functions is used for moving, rotating, scaling and otherwise transforming geometric objects, which transformations are related to transformation of the coordinate system. Functions create transformed objects, while original objects do not change.

**Move** function is used to move (shift) graphic objects.

Format:

```
Move(<Object>, <dx>, <dy> , <dz>[, <count>]),
```

where

<Object> - original graphic object;  
<dx>, <dy>, <dz> - value of movement along x, y and z axes, respectively;  
<count> - number of created objects, where each subsequent object is created by moving the preceding object; this argument is optional, with the default value of 1.

**RotateX**, **RotateY** и **RotateZ** functions are used to rotate graphic objects relative the axes, parallel to X, Y and Z axes, respectively.

Format of functions:

```
RotateX(<Object>, <rotation angle>[, <cy>, <cz>[, <count>]])  
RotateY(<Object>, <rotation angle>[, <cx>, <cz>[, <count>]])  
RotateZ(<Object>, <rotation angle>[, <cx>, <cy>[, <count>]])
```

Here:

<Object> - original graphic object;  
<Rotation angle> - angle of rotation;  
<cx>, <cy>, <cz> - determine the value of rotation axes' offset relative X, Y and Z axes (in accordance with function names). These arguments are optional; however, only all three arguments can be omitted at once. Default value for each of <cx>, <cy>, <cz> is zero.  
<count> - number of created objects, where each subsequent object is created by transforming the preceding object; this argument is optional, with the default value of 1.

## 4.6. Functions for Boolean operations

Functions of this class are used to perform Boolean operations on geometric objects. **BooleanUnion** function performs union operation, **BooleanSubtract** function – subtraction operation, and **BooleanIntersect** function – intersection of geometric objects.

Format of functions:

```
BooleanUnion(<Object>, <Object>, ...)  
BooleanSubtract(<Object>, <Object>, ...)  
BooleanIntersect(<Object>, <Object>, ...)
```

Here,

<Object> - initial graphic objects to be used in the Boolean operation.

## 4.7. Auxiliary functions

Functions **ExtentsX1**, **ExtentsX2**, **ExtentsY1**, **ExtentsY2**, **ExtentsZ1** и **ExtentsZ2** are used to calculate the extents of graphic objects.

Format of functions:

```
ExtentsX1 (<Object>)  
ExtentsX2 (<Object>)  
ExtentsY1 (<Object>)  
ExtentsY2 (<Object>)  
ExtentsZ1 (<Object>)  
ExtentsZ2 (<Object>)
```

Here,

<Object> - initial graphic object.

Presence of X, Y or Z characters in the function name determines along which axis the extents will be calculated. Figure 1 tells that the minimum value is required; figure 2 – maximum value.

## 5. Special functions and operators

**IF** function allows performing various actions depending on whether the specified condition is fulfilled or not fulfilled. It plays the role of a conditional operator, and can be used to create branches in the logic of building a parametric symbol.

Format of the function:

```
IF (<Condition>, <ExprOnTRUE>, <ExprOnFALSE>),
```

where

<Condition> - expression, which defines the condition under test using the following comparison operations: “==”(equal), “<”(less than), “>”(greater than), “<=” (not greater than), “>=”(not less than);

<ExprOnTRUE> - expression, which defines the value of IF function when the value of <Condition> is TRUE;

<ExprOnFALSE> - expression, which defines the value of IF function when the value of <Condition> is FALSE;

Example:

```
A = IF(L >= H, Rectangle(L, H), Rectangle(H, L));
```

Regardless of the specified size of L and H, the created rectangle A will be positioned horizontally (the longer side will be along the X axis).

### 5.1. Setting and changing object properties

SetProperties function is used to set (or reset) properties of geometric objects.

The function has the following format:

```
SetProperties (<Object>, <PropertyName> = PropertyValue, <PropertyName> =  
PropertyValue, ...);  
    <Object> - initial graphic object  
    <PropertyName> - property name, without quotation marks  
    <PropertyValue> - value
```

Example: `RedRect = SetProperty(BlueRect, PenColor = RED, PenWidth = 0.2);`

## 6. Creating custom functions

When scripts of the same type are created, which describe a particular class of parametric symbols, it can be convenient to have the sequence of repeated actions as a separate specialized functions. To achieve this, the repeated actions are put into a separate `<name>.psm` file. In this case, all input variables should be listed in the **Input** operator:

```
Input(<list of variable identifiers, separated with commas>);
```

**Output** operator should also be defined, but in the current version only the first object in the list of output objects will be used as the result. Other objects in the list will be ignored.

The file created in this manner needs to be placed in **Macro** folder. When the function is called, its format will be as follows:

```
<file name>(<list of input parameters>).
```

Example of custom function:

Below are the contents of the file `Box.psm`:

```
// Function to define Box function:
//   B = Box(Xmin, Ymin, Zmin, Xmax, Ymax, Zmax);
//   The function creates 3D box with given min/max values
Input(x0,y0,z0,x1,y1,z1);
R = Rectangle(x1-x0, y1-y0 // Rectangle with Xmin = x0, Xmax= x1
              (x0+x1)/2, (y0+y1)/2); // Ymin = y0, Ymax = y1
T = Thickness(R, z1-z0);
Output(Move(T,0, 0, z0)); // and Box's extents Zmin = z0, Zmax = z1 now
```

### Special functions without parameters:

PI - calculates the value of Pi = 3.14159...

## Parametric Symbols Appendix A

### *Reserved words*

### *Revision History*

## Appendix A - Reserved Words

№	Reserved word	Description
	PI	
	LINEAR	
	TEXT	
	ANGULAR	
	MATERIAL	
	FONT	
	COLOR	
	CHECKBOX	
	ITALIC	
	BOLD	
	UNDERLINE	
	BOX	
	ALLCAPS	
	STRICKETHROUGH	
	TOP	
	MIDDLE	
	BOTTOM	
	BASELINE	
	LEFT	
	CENTER	
	RIGHT	
	Call	
	Array	
	+	
	-	
	*	
	Div	
	Mod	
	/	
	-	
	sin	
	cos	
	tan	
	atan	
	min	
	max	
	**	
	=	
	==	
	!=	
	<	
	>	

	<=	
	>=	
	&	
	Solid	
	Extrude	
	UNIQUE	
	GraphicId	
	VertexId	
	Vertex	
	Face	
	Edge	
	Source	
	Bound	
	Intersect	
	OperationList	
	BlendArg	
	BlendParam	
	BlendType	
	BlendRadiusMode	
	BlendSetback	
	BlendRadiusBlendSmooth	
	BlendRadiusParam	
	BlendOffsetParam	
	BlendFaceEntity	
	BlendFaceEdge	
	BlendFaceVertex	
	BlendEdgeEdge	
	BlendEdgeVertex	
	BlendEdgeVertexMain	
	BlendEdgeVertexAux	
	ShellArg	
	ShellThickness	
	ShellFace	
	ShellEdge	
	FaceEditArg	
	Transform	
	ScaleX	
	ScaleY	
	ScaleZ	

	ShearXY	
	ShearXZ	
	ShearYZ	
	RotateX	
	RotateY	
	RotateZ	
	TranslateX	
	TranslateY	
	TranslateZ	
	Path	
	Profile	
	LateralFace	
	LateralEdge	
	CapFace	
	CapEdge	
	JointEdge	
	Profiles	
	HighLight	
	FaceMaterialArg	
	FaceMaterial	
	FaceOffsetArg	
	FaceHoleArg	
	FaceHole	
	BendId	
	BendRadius	
	BendAngle	
	BendNeutral	
	BendFlag	
	BendPosition	
	BendFlangeHeight	
	BendAxialDistance	
	BendAzimuthAngle	
	BendEdgeStartPosition	
	BendEdgeEndPosition	
	Face2FaceLoftArg	
	Face2FaceLoft	
	AssemblyAxis	
	Input	
	Output	
	Include	
	Units	

	StaticSymbol	
	FolderList	
	Macro	
	Parameters	
	Parameter	
	ParameterPoint	
	PointX	
	PointY	
	PointZ	
	Set	
	Interval	
	LessThan	
	GreatherThan	
	LessOrEquail	
	GreatherOrEquail	
	Circle	
	Rectangle	
	Polyline	
	Point	
	Arc0	
	Arc1	
	Fillet	
	IF	
	Move	
	Thickness	
	Sweep	
	Cone	
	BooleanUnion	
	BooleanSubtract	
	BooleanIntersec	
	G3Fillet	
	G3Chamfer	
	G3Shell	
	G3Offset	
	G3Slice	
	G3Bend	
	ExtentsX1	
	ExtentsX2	
	ExtentsY1	
	ExtentsY2	

	ExtentsZ1	
	ExtentsZ2	
	Text	
	TextFont	
	TextStyle	
	Group	
	SetProperties	
	PatternCopy	