

Quick Reference

Links

[Using the Path Library](#)

[Function Summary](#)

[Function Definitions](#)

[Glossary](#)

Path Library Version: [1-0](#)

ETAC Programming Language Version: [1-1](#)

ETAC Interpreter Version: [2-0-6-ena](#)

Path Library

Inclusion File: [etacPathFunctions.etac](#)

Data Object Name: [etacPathFunctions](#)

Required: [etacSplineFunctions.etac](#) (only for `pfBezPathHit()`)

Number of Functions: [15](#)

Release Date: [1 December 2025](#)

Path ETAC Script Library

1 December 2025

Copyright © **Victor Vella** (2025)
All rights reserved.



Other Related ETAC Documents

ETAC_Preliminaries.pdf	Preliminaries before using ETAC
ETACOverview.pdf	An Overview of ETAC
ETACProgLang(Official).pdf	The Official ETAC Programming Language
ETACVisualInteractionSystem.pdf	ETAC: Visual Interaction System
RunETAC.chm	Run ETAC Scripts Help
ETACWithCPP.pdf	ETAC: Interacting with C++
ETACCompiler.pdf	The ETAC Compiler
ETACCompiler.chm	ETAC Compiler Help
ETACErrorCodes.pdf	ETAC Compilation and Run-time Error Codes
FunctionsETACScriptLib.pdf	Functions ETAC Script Library
SplineScriptFnts.pdf	Spline ETAC Script Library
VISShapePathFnts.pdf	VIS Shape Path Functions
VISPointSetFnts.pdf	VIS Point Set Functions

Legal Information

ETAC and  (the ETAC logo) are unregistered trademarks (™) of Victor Vella for *computer software incorporating an implementation of a computer programming language*. There may be other owners of the “ETAC” trademark used for other purposes.

This document is copyright © by Victor Vella (2025). All rights reserved. Permission is hereby granted to make any number of exact electronic copies of this document without any remuneration whatsoever. Permission is also granted to make annotated electronic copies of this document for personal use only. Except for the permissions granted, and apart from any fair dealing as permitted under the relevant Copyright Act, no part of this document may be reproduced or transmitted in any form or by any means without the express permission of the author. The copyright of this document shall remain entirely with the original copyright holder.

The author of this document shall not be liable for any direct or indirect consequences arising with respect to the use of all or any part of the information in this document, even if such information is inaccurate or in error. The information in this document is subject to change without notice.

Contents

Quick Reference.....	i
Contents.....	iv
Document Conventions.....	v
1. Introduction.....	1
2. Features of an ETAC Script Library.....	1
3. Using an ETAC Script Library.....	1
4. Path ETAC Script Library Reference.....	3
4.1 Function Summary.....	4
4.2 Function Definitions.....	4
Bibliography.....	12
Glossary.....	13

Document Conventions

The following symbolic conventions are used in this document.

Symbol	Meaning
$\langle x \rangle$	separates x as a unit of information from the surrounding text.
$x \cdots$	middle ellipsis means zero, one, or more of the same kind as x .
$[x]$	means that x optional.
$\cdots$	ellipsis represents omitted text (as usual).
<i>text</i>	maroon coloured italic text is a link to the text's definition.
text	bold green text is a link into the document.
♦	indicates the end of a block of text.

Path ETAC Script Library

This document is for version **1-0-2-ena** of the Path ETAC Script Library for version **1-1** of the **ETAC™ Programming Language** implemented in RunETAC.exe and AppETAC.dll version **2-0-6-ena**. The source file of the library is in etacPathFunctions.etac.

(Australian English)

1. Introduction

ETAC™ (pronounced: E-tack) is a syntactically simple but extremely versatile dictionary and stack based interpreted script programming language. The ETAC programming language is not based on any other programming language. Familiarity with the ETAC programming language is required to fully understand this document. See the document ETACOverview.pdf for an overview of the language, and also the document ETACProgLang(Official).pdf for the official definition of the language.

The Path ETAC Script Library (referred to as the “Path Library”) is an *ETAC script library* containing various functions involving the relationships among paths as a sequence of points, *rectangles*, and *points*.

2. Features of an ETAC Script Library

An *ETAC script library* (ESL) is an *ETAC text script* file containing *ETAC function* definitions. The *ETAC functions* are constructed in such a way that the same function is allocated only once within a *main ETAC session*, even when the same ESL file is included more than once.

An ETAC programmer can arrange for an ESL so that only the desired *ETAC functions*, and their dependencies, are allocated within a *main ETAC session*. This feature is useful when only a few functions of an ESL containing a large number of *ETAC function* definitions are required in an ETAC program. By default, all of the *ETAC functions* in an ESL are included in an ETAC program.

The included functions of an ESL are typically allocated within a named *data object* having a name based on the name of the ESL.

3. Using an ETAC Script Library

An *ETAC script library* (ESL) exists in an *ETAC text script* file, and can be included in any *ETAC text script* as an inclusion file (using the `<:include>` pre-processor inclusion command). An ESL is typically included at the top of the *ETAC text script*. The processing of the ESL inclusion file causes a new named *data object* to be created containing the desired *ETAC function* allocations; the *data object* name is based on the name of the ESL. For example, the *data object* name into which the desired functions of the Path Library are included is called `etacPathFunctions`.

Each *ETAC function* within an ESL is identified by a pre-processor definition name beginning with the initials of the ESL name followed by the name of the particular function. For example, if an ESL function name is `xxxBezBoundingBox`, then the corresponding pre-processor definition name would be `@XXX_BEZ_BOUND_BOX`, where `xxx` and `XXX` are the initials of the ESL name (for the Path Library, `XXX` is `PF`, so the function name identifiers are `pfBezBoundingBox` and `@PF_BEZ_BOUND_BOX`, respectively).

The pre-processor definition name of each *ETAC function* within an ESL is called the *ESL function identifier* (for example, @PF_BEZ_BOUND_BOX is an *ESL function identifier*). To explicitly include a particular ESL function within some *ETAC text script*, the corresponding ESL function identifier must be defined before the ESL inclusion file is included. For example, to include the pfBezBoundingBox function of the Path Library into *ETAC text script*, <::define @PF_BEZ_BOUND_BOX> must be present before <::include "etacPathFunctions.etac">. If the included function calls other ESL functions, then those other ESL functions are automatically included as well.

Including an ESL inclusion file without specifying the definition of any *ESL function identifier* of that ESL results in all of the functions in that ESL being included. For example, specifying only <::include "etacPathFunctions.etac"> without defining any of the *ESL function identifiers* of the Path Library beforehand (in the same *ETAC text script*) results in all of the *ETAC functions* within the Path Library to be included into the *data object* etacPathFunctions.

If an *ESL function identifier* is defined more than once for the same ESL inclusion file, then the corresponding *ETAC function* is allocated only once within the *main ETAC session*, even when the same ESL inclusion file is included more than once.

In summary, to use an *ETAC script library*, the following are required:

- The file name of the ESL, which must be included before any of the ESL functions are used.
- A list of *ESL function identifiers* (corresponding to the desired ESL functions), which must be defined before the inclusion of the ESL file. The identifiers are obtained from the ESL source file or documentation.
- The name of the ESL *data object* (obtained from the ESL source file or documentation), which is automatically created by the inclusion of the ESL.
- A variable used to access the ESL *data object*. The desired ESL functions are accessed via that variable, which is assigned from the return value of the global @NewData *ETAC function*.

The following example is an illustration of how to use the Path Library in a main ETAC program.

Using the Path Library

```
[*::10ETAC-SOURCE-V1::*]
[* ETAC main program file illustrating how to use the Path Library. *]
::include "TACGlobal.PTAC" [* Required by the Path Library. *]

...

::define @PF_BEZ_BOUND_BOX [* Allows allocation of the pfBezBoundingBox() function. *]
::define @PF_POLY_PATH_HIT [* Allows allocation of the pfPolyPathHit() function. *]
::define @PF_EXPAND_RECT [* Allows allocation of the pfExpandRect() function. *]
::include "etacPathFunctions.etac" [* Path Library inclusion file. *]
[* etacPathFunctions.etac creates the etacPathFunctions data object containing the
    three functions. *]

...

start_local; [* Create a local dictionary for the main program. *]
[* Other program variables allocated here. *]

    [* Allocate 'PFnts' variable for accessing ESL functions. *]
    PFnts :- @NewData("etacPathFunctions");

    [* PROGRAM *]
    ...

    Seq := PFnts.pfBezBoundingBox([...]); [* Call to ESL function. *]

    ...

end_local;
```

In the example above, three *ETAC functions*, `pfBezBoundingBox()`, `pfPolyPathHit()`, and `pfExpandRect()`, defined in the Path Library are to be used in the program (the three functions are for illustrative purposes only). The corresponding *ESL function identifiers* of those functions are defined (via `::define`) so that the three functions are automatically allocated in the `etacPathFunctions` *data object* when the Path Library inclusion file, `etacPathFunctions.etac`, is included in the program. The three *ESL function identifiers* must be defined before the inclusion of the Path Library inclusion file. If no *ESL function identifiers* were defined, then all the *ETAC functions* of the Path Library would have been allocated in the `etacPathFunctions` *data object*, even though only some of those functions are used in the program. Those three functions will not be allocated again within the `etacPathFunctions` *data object*, even if they were to be specified for inclusion in another *ETAC session*.

Somewhere in the program, after the inclusion of the Path Library, a replicate of the `etacPathFunctions` *data object* is created by the global `@NewData` function, and allocated to a variable, `PFnts`, to access the *ETAC functions* within the *data object*. The `etacPathFunctions` *data object* is replicated (via `@NewData`) rather than used directly (via `@Data`) so as to keep the original *ETAC functions* within the *data object* unchanged; *ETAC functions* within an ESL may contain initialised persistent local data which needs to remain as originally initialised. Using an ESL *data object* directly may cause undesirable changes in any persistent local data within that *data object*.

4. Path ETAC Script Library Reference

This section contains information about the Path Library. The Path Library exists in the `etacPathFunctions.etac` inclusion file. When that file is included (via `::include`) in *ETAC text script*, a *data object* named `etacPathFunctions` is automatically created containing the desired functions. Those functions can then be assessed via a replicate of that *data object*.

4.1 Function Summary

The table below contains an alphabetical list of the Path Library functions.

Function Summary for the Path Library

Function	Description
pfBezBoundingBox()	Calculates the bounding box that tightly enclosing the shape defined by a <i>poly-curve</i> .
pfBezPathHit()	Determines whether a <i>point</i> is near a <i>poly-curve</i> .
pfExpandRect()	Expands a <i>rectangle</i> by a given amount.
pfGetBound()	Returns the smallest bounding box containing four <i>points</i> .
pfGetPathBound()	Returns the smallest bounding box containing the points of any path.
pfInnerRect()	Creates a new <i>rectangle</i> which is the intersection of two given <i>rectangles</i> .
pfNormRect()	Normalises a <i>rectangle</i> .
pfOuterRect()	Creates a new <i>rectangle</i> which is the union of two given <i>rectangles</i> .
pfPolyPathHit()	Determines whether a <i>point</i> is near a poly-line path.
pfPtInRect()	Determines whether a <i>point</i> is in a <i>rectangle</i> .
pfRectDims()	Returns the width and height of a <i>rectangle</i> .
pfRectIsNorm()	Determines whether a <i>rectangle</i> is normalised.
pfRectIsNull()	Determines whether a <i>rectangle</i> is null.
pfRectToPath()	Converts a <i>rectangle</i> to an equivalent poly-line path.
pfShiftRect()	Shifts a <i>rectangle</i> by a specified amount.

4.2 Function Definitions

The following boxes contain a description of all the Path Library functions. The functions need to be called in the context of the `etacPathFunctions` *data object*. The corresponding *ESL function identifier* is shown at the top right of each box.

pfBezBoundingBox()

pfBezBoundingBox	(pPath)	→	<i>bound-box</i>	@PF_BEZ_BOUND_BOX
<i>pCPts</i>	A decimal sequence stack object.			
<i>bound-box</i>	A decimal sequence stack object.			

Details

Calculates the bounding box (*box-path*) that tightly enclosing the shape defined by a *poly-curve* (*pPath*).

pPath is the *poly-curve* path defining a shape.

bound-box is a *rectangle* specifying the smallest bounding box containing the shape defined by *pCPts*.

The returned bounding box tightly encloses the shape itself defined by *pPath* not the points within it. As a consequence, this function is not as efficient as the **pfGetPathBound()** function which simply calculates the smallest bounding box containing the points of a path.

Other Information

pfGetPathBound() ♦

pfBezPathHit()

pfBezPathHit (<i>pPath</i> <i>pPoint</i> <i>pTol</i>) → <i>bool</i>		@PF_BEZ_PATH_HIT
<i>pPath</i>	A decimal sequence stack object.	
<i>pPoint</i>	A decimal sequence stack object.	
<i>pTol</i>	A decimal stack object.	
<i>bool</i>	An integer stack object containing a logical boolean value.	

Details

Determines whether a *point* (*pPoint*) is near (*pTol*) a path (*pPath*).

pPath is a *poly-curve* path defining a shape.

pPoint is the *point* to test.

pTol (tolerance) is the approximate perpendicular distance of *pPoint* from the shape of the path defined by *pPath* for *bool* to be true. *pTol* is typically a small value like 0.2.

bool is true if *pPoint* is within *pTol* distance from any part of the shape defined by the specified path (*pPath*); otherwise it is false if *pPoint* is not near the shape.

For example, if *pPath* defines a circle, then this function returns true if the *point* specified by *pPoint* is near the circumference of the circle.

Other Information

pfPolyPathHit() ♦

pfExpandRect()

pfExpandRect (<i>pRect</i> <i>pHor</i> <i>pVer</i>) → <i>rect</i>		@PF_EXPAND_RECT
<i>pRect</i>	A decimal sequence stack object.	
<i>pHor</i>	An integer or decimal stack object.	
<i>pVer</i>	An integer or decimal stack object.	
<i>rect</i>	A decimal sequence stack object.	

Details

Expands a *rectangle* (*pRect*) by a given amount (*pHor*, *pVer*).

pRect is the *rectangle* to expand.

pHor is the amount to expand the *rectangle* horizontally. This value can be negative. The value is subtracted from *x-min*, and added to *x-max*.

pVer is the amount to expand the *rectangle* vertically. This value can be negative. The value is subtracted from *y-min*, and added to *y-max*.

rect is the same *rectangle* as *pRect* but expanded or contracted by the specified amount.

Note that there is no checking of the values *pHor* and *pVer*. Therefore, *pRect* can become non-normalised.

Other Information

pfNormRect() ♦

pfGetBound()

pfGetBound(<i>pPt0 pPt1 pPt2 pPt3</i>) → <i>bound-box</i>		@PF_GET_BOUND
<i>pPt0</i>	A decimal sequence stack object.	
<i>pPt1</i>	A decimal sequence stack object.	
<i>pPt2</i>	A decimal sequence stack object.	
<i>pPt3</i>	A decimal sequence stack object.	
<i>bound-box</i>	A decimal sequence stack object.	

Details

Returns the smallest bounding box (*bound-box*) containing four *points* (*pPt0*, *pPt1*, *pPt2*, *pPt3*).

pPt0 to *pPt3* are four *points*.

bound-box is a *rectangle* specifying the smallest bounding box containing the four *points*.

Other Information

pfGetPathBound() ♦

pfGetPathBound()

pfGetPathBound(<i>pPath</i>) → <i>bound-box</i>		@PF_GET_PATH_BOUND
<i>pPath</i>	A decimal sequence stack object.	
<i>bound-box</i>	A decimal sequence stack object.	

Details

Returns the smallest bounding box (*bound-box*) containing the points of a path (*pPath*).

pPath is a TAC sequence containing points of the form: [*x*₁, *y*₁ [, *x*₂, *y*₂, ...]], or an empty sequence.

bound-box is a *rectangle* specifying the smallest bounding box containing the points of *pPath*.

If *pPath* is an empty sequence, then *bound-box* will contain a null *rectangle* [0.0, 0.0, 0.0, 0.0].

Other Information

pfGetBound() ♦

pfInnerRect()

pfInnerRect (<i>pRect1</i> <i>pRect2</i>) → <i>rect</i>		@PF_INNER_RECT
<i>pRect1</i>	A decimal sequence stack object.	
<i>pRect2</i>	A decimal sequence stack object.	
<i>rect</i>	A decimal sequence stack object.	

Details

Creates a new *rectangle* (*rect*) which is the intersection of two specified *rectangles* (*pRect1*, *pRect2*).

pRect1 and *pRect2* are *rectangles*.

rect is the *rectangle* defining the intersection of the two specified *rectangles*, or is [0.0, 0.0, 0.0, 0.0] if the *rectangles* do not intersect.

pRect1 and *pRect2* are assumed to be normalised (neither the width nor height are negative). Either of the two *rectangles* can have zero width and/or height.

The intersection of two *rectangles* is the smallest *rectangle* that contains the common overlapping area of the two *rectangles*. A *rectangle* with zero width and/or height is considered to have an area of infinitesimal dimension rather than having no area in the relevant dimension. However, a null *rectangle* (all border positions are zero) is considered to have no area. If either *pRect1* or *pRect2* are null *rectangles*, then *rect* is a null *rectangle*.

Other Information

pfOuterRect() ▪ **pfNormRect()** ♦

pfNormRect()

pfNormRect (<i>pRect</i>) → <i>rect</i>		@PF_NORM_RECT
<i>pRect</i>	A decimal sequence stack object.	
<i>rect</i>	A decimal sequence stack object.	

Details

Normalises a *rectangle* (*pRect*).

pRect is a *rectangle*.

rect is the same *rectangle* as *pRect* but normalised. A normalised *rectangle* does not have negative width or height.

If the width or height of *pRect* is negative, then this function makes it positive by swapping the appropriate border positions in *pRect*. A negative dimension occurs when the maximum coordinate is less than the minimum coordinate of the dimension.

Other Information

pfRectIsNorm() ♦

pfOuterRect()

pfOuterRect(<i>pRect1</i> <i>pRect2</i>) → <i>rect</i>		@PF_OUTER_RECT
<i>pRect1</i>	A decimal sequence stack object.	
<i>pRect2</i>	A decimal sequence stack object.	
<i>rect</i>	A decimal sequence stack object.	

Details

Creates a new *rectangle* (*rect*) which is the union of two specified *rectangles* (*pRect1*, *pRect2*).

pRect1 and *pRect2* are *rectangles*.

rect is the *rectangle* defining the union of the two specified *rectangles*.

pRect1 and *pRect2* are assumed to be normalised (neither the width nor height are negative). Either of the two *rectangles* can have zero width and/or height.

The union of two *rectangles* is the smallest *rectangle* that contains both *rectangles*. A *rectangle* with zero width and/or height is considered to have an area of infinitesimal dimension rather than having no area in the relevant dimension. However, a null *rectangle* (all border positions are zero) is considered to have no area. If either *pRect1* or *pRect2* are null *rectangles*, then *rect* is the other *rectangle*.

Other Information

pfInnerRect() ▪ pfNormRect() ♦

pfPolyPathHit()

pfPolyPathHit(<i>pPath</i> <i>pPoint</i> <i>pTol</i>) → <i>bool</i>		@PF_POLY_PATH_HIT
<i>pPath</i>	A decimal sequence stack object.	
<i>pPoint</i>	A decimal sequence stack object.	
<i>pTol</i>	A decimal stack object.	
<i>bool</i>	An integer stack object containing a logical boolean value.	

Details

Determines whether a *point* (*pPoint*) is near (*pTol*) a poly-line path (*pPath*).

pPath is a poly-line path defining a shape.

pPoint is the *point* to test.

pTol (tolerance) is the approximate perpendicular distance of *pPoint* from a line segment of the path defined by *pPath* for *bool* to be true. *pTol* is typically a small value like 0.2.

bool is true if *pPoint* is within *pTol* distance from any line segment defined by the specified path (*pPath*); otherwise it is false if *pPoint* is not near a line segment.

A poly-line is a TAC sequence of points, that are the vertices of line segments, in the form:

[*x*₁, *y*₁[, *x*₂, *y*₂, ...]] .

Other Information

pfBezPathHit() ♦

pfPtInRect()

pfPtInRect (<i>pRect</i> <i>pPoint</i>) → <i>bool</i>		@PF_PT_IN_RECT
<i>pRect</i>	A decimal sequence stack object.	
<i>pPoint</i>	A decimal sequence stack object.	
<i>bool</i>	An integer stack object containing a logical boolean value.	

Details

Determines whether a *point* (*pPoint*) is in a *rectangle* (*pRect*).

pRect is a *rectangle*.

pPoint is the *point* to test.

bool is true if the *point* specified by *pPoint* is within or on the border of the *rectangle* defined by *pRect*. ♦

pfRectDims()

pfRectDims (<i>pRect</i>) → <i>width</i> <i>height</i>		@PF_RECT_DIMS
<i>pRect</i>	A decimal sequence stack object.	
<i>width</i>	A decimal stack object.	
<i>height</i>	A decimal stack object.	

Details

Returns the width (*width*) and height (*height*) of a *rectangle* (*pRect*).

pRect is a *rectangle*.

width is the width of the *rectangle* defined by *pRect*. This value may be negative if the *rectangle* is not normalised.

height is the height of the *rectangle* defined by *pRect*. This value may be negative if the *rectangle* is not normalised.

Other Information

pfRectIsNorm() ♦

pfRectIsNorm()

pfRectIsNorm (<i>pRect</i>) → <i>bool</i>		@PF_RECT_IS_NORM
<i>pRect</i>	A decimal sequence stack object.	
<i>bool</i>	An integer stack object containing a logical boolean value.	

Details

Determines whether a *rectangle* (*pRect*) is normalised.

pRect is a *rectangle*.

bool is true if the *rectangle* is normalised; otherwise it is false.

A normalised *rectangle* has non-negative width and height — the right edge is not less than the left edge and the top edge is not less than the bottom edge.

Other Information

pfNormRect() ♦

pfRectIsNull()

pfRectIsNull(<i>pRect</i>) → <i>bool</i>		@PF_RECT_IS_NULL
<i>pRect</i>	A decimal sequence stack object.	
<i>bool</i>	An integer stack object containing a logical boolean value.	

Details

Determines whether a *rectangle* (*pRect*) is null.

pRect is a *rectangle*.

bool is true if the *rectangle* is null; otherwise it is false.

A null *rectangle* has all dimensions being zero (ie: [0.0, 0.0, 0.0, 0.0]). Note that a *rectangle* with zero width and height is not necessarily a null *rectangle*. ♦

pfRectToPath()

pfRectToPath(<i>pRect</i>) → <i>path</i>		@PF_RECT_TO_PATH
<i>pRect</i>	A decimal sequence stack object.	
<i>path</i>	A decimal sequence stack object.	

Details

Converts a *rectangle* (*pRect*) to an equivalent poly-line path (*path*).

pRect is a *rectangle*.

path is a poly-line path of the form: [$x_1, y_1, x_2, y_2, \dots, x_5, y_5$], where $x_5 = x_1$ and $y_5 = y_1$.

If the *rectangle* in *pRect* is [x_1, y_1, x_2, y_2], then *path* is [$x_1, y_1, x_2, y_1, x_2, y_2, x_1, y_2, x_1, y_1$]. ♦

pfShiftRect()

pfShiftRect(<i>pRect</i> <i>pXAmount</i> <i>pYAmount</i>) → <i>rect</i>		@PF_SHIFT_RECT
<i>pRect</i>	A decimal sequence stack object.	
<i>pXAmount</i>	An integer or decimal stack object.	
<i>pYAmount</i>	An integer or decimal stack object.	
<i>rect</i>	A decimal sequence stack object.	

Details

Shifts a *rectangle* (*pRect*) by a specified amount (*pXAmount*, *pYAmount*).

pRect is the *rectangle* to shift.

pXAmount is the amount to shift the *rectangle* horizontally.

pYAmount is the amount to shift the *rectangle* vertically.

rect is the same *rectangle* as *pRect* but shifted by the specified amount.

Other Information

pfExpandRect () ♦

Bibliography

The Official ETAC Programming Language copyright © Victor Vella (2025).

ETAC: Visual Interaction System copyright © Victor Vella (2025).

Glossary

D

data object

The container of an ETAC dictionary used as a programmer-defined data structure consisting of stack objects identified by name. The dictionary itself is identified by the name defined by the private pre-processor definition `<_DATA_DICT>`.

E

ESL function identifier

The pre-processor definition name identifying an *ETAC function* within an *ETAC script library*. An *ESL function identifier* is specified by the programmer to include the corresponding *ETAC function* definition into *ETAC text script*.

ETAC function

The container of a special ETAC created procedure that creates a local dictionary then assigns the object stack arguments to that dictionary before calling the programmer-defined procedure.

ETAC script library

An *ETAC text script* file containing *ETAC function* definitions designed in such a way that the same required function is allocated only once within a *main ETAC session*.

ETAC session

The period devoted to the processing of ETAC code by the TAC processor after having been processed by the interpretation part of the ETAC interpreter. New *ETAC sessions* can exist within a given *ETAC session* for different ETAC code. Therefore, a given *ETAC session* can produce a new *ETAC session* (relating to different ETAC code from the given *ETAC session*) so that when the new *ETAC session* ends, the given *ETAC session* resumes.

ETAC text script

ETAC program code that is in human readable and writable text form. A file containing only *ETAC text script* typically has an extension of ‘etac’. Note that the term “ETAC text script” is used in the same sense as the word “code”, as in “ETAC text script code”.

M

main ETAC session

An *ETAC session* and all other new *ETAC sessions* produced directly or indirectly from that *ETAC session*, but not itself produced from any other *ETAC session*. A *main ETAC session* is typically begun via the RunETAC.exe and the AppETAC.dll computer programs.

P

point

A TAC sequence of the form $[x, y]$.

poly-curve

A TAC sequence of X and Y coordinates in the form $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$, where each $\langle x_i, y_i \rangle$ pair ($i = 1$ to n) represents a point of the *poly-curve*. A *poly-curve* consists of joined Bezier curve segments. Other public documentation needs to be consulted for the definition of a Bezier curve segment. A *poly-curve* must have at least four points (unless it is empty).

R

rectangle

A TAC sequence specifying a rectangle of the following form.

$[x-min, y-min, x-max, y-max]$,

where

- 1 $x-min$ is the left border position of the *rectangle*,
- 2 $y-min$ is the bottom border position of the *rectangle*,
- 3 $x-max$ is the right border position of the *rectangle*,
- 4 $y-max$ is the top border position of the *rectangle*.