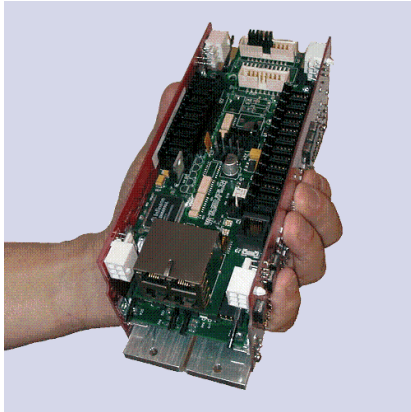




The Guidance Programming Language



GPL Dictionary Pages

Version 2.0.1, March 19, 2008
P/N: GPL0-DI-00110

Document Content

The information contained herein is the property of Precise Automation Inc., and may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without the prior written approval of Precise Automation Inc. The information herein is subject to change without notice and should not be construed as a commitment by Precise Automation Inc. This information is periodically reviewed and revised. Precise Automation Inc., assumes no responsibility for any errors or omissions in this document.

Copyright © 2004-2008 by Precise Automation Inc. All rights reserved.

The Precise Logo is a registered trademark of Precise Automation Inc.

Trademarks

Guidance 3400, Guidance 3300, Guidance 3200, Guidance 2400, Guidance 1400, Guidance 1300, Guidance 1200, Guidance Controller, Guidance Development Environment, GDE, Guidance Development Suite, GDS, Guidance Dispense, Guidance Programming Language, GPL, Guidance System, PrecisePlace 1300, PrecisePlace 1400, PrecisePlace 2300, PrecisePlace 2400, PreciseFlex, PrecisePower 500, PrecisePower 2000, PreciseVision, RIO are either registered or trademarks of Precise Automation Inc., and may be registered in the United States or in other jurisdictions including internationally. Other product names, logos, designs, titles, words or phrases mentioned within this publication may be trademarks, service marks, or trade names of Precise Automation Inc. or other entities and may be registered in certain jurisdictions including internationally.

Any trademarks from other companies used in this publication are the property of those respective companies. In particular, Visual Basic, Visual Basic 6 and Visual Basic.NET are trademarks of Microsoft Inc.

Disclaimer

PRECISE AUTOMATION INC., MAKES NO WARRANTIES, EITHER EXPRESSLY OR IMPLIED, REGARDING THE DESCRIBED PRODUCTS, THEIR MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. THIS EXCLUSION OF IMPLIED WARRANTIES MAY NOT APPLY TO YOU. PLEASE SEE YOUR SALES AGREEMENT FOR YOUR SPECIFIC WARRANTY TERMS.

Precise Automation Inc.
727 Filip Road
Los Altos, California 94024
U.S.A.
www.preciseautomation.com

Warning Labels

The following warning and caution labels are utilized throughout this manual to convey critical information required for the safe and proper operation of the hardware and software. It is extremely important that all such labels are carefully read and complied with in full to prevent personal injury and damage to the equipment.

There are four levels of special alert notation used in this manual. In descending order of importance, they are:



DANGER: This indicates an imminently hazardous situation, which, if not avoided, will result in death or serious injury.



WARNING: This indicates a potentially hazardous situation, which, if not avoided, could result in serious injury or major damage to the equipment.



CAUTION: This indicates a situation, which, if not avoided, could result in minor injury or damage to the equipment.

NOTE: This provides supplementary information, emphasizes a point or procedure, or gives a tip for easier operation

Table Of Contents

GPL Dictionary Pages Summary	1
Array Class	3
Array Class Summary	3
array.GetUpperBound Property	4
array.Length Property	5
array.Rank Property	6
Console Class	7
Console Class Summary	7
Console.Write Method	8
Console.WriteLine Method	9
Controller Class	10
Controller Class Summary	10
Controller.ErrorLog Property	12
Controller.Load Method	14
Controller.PDb Property	15
Controller.PDbNum Property	17
Controller.PowerEnabled Property	19
Controller.PowerState Property	21
Controller.RecordButton Property	23
Controller.ShowDialog Method	24
Controller.ShowDialogMCP Method	27
Controller.SleepTick Method	30
Controller.SoftEStop Property	31
Controller.SystemMessage Method	32
Controller.Tick Property	33
Controller.Timer Property	34
Controller.Unload Method	35
Exception Handling.....	36
Exception Handling Summary	36
Catch Statement	38
End Try Statement	39

Table Of Contents

Exit Try Statement	40
Finally Statement	41
Throw Statement	42
Try..Catch..Finally..End Try Statements	44
exception_object.Axis Property	47
exception_object.Clone Method	48
exception_object.ErrorCode Property	49
exception_object.Message Method	50
exception_object.Qualifier Property	51
exception_object.RobotError Property	52
exception_object.RobotNum Property	53
File and Serial I/O Classes	54
File and Serial I/O Classes Summary	54
File.CreateDirectory Method	56
File.DeleteDirectory Method	57
File.DeleteFile Method	58
File.GetDirectories Method	59
File.GetFiles Method	60
New StreamReader Constructor	61
streamreader_object.Close Method	62
streamreader_object.Peek Method	63
streamreader_object.Read Method	64
streamreader_object.ReadLine Method	65
New StreamWriter Constructor	66
streamwriter_object.AutoFlush Property	67
streamwriter_object.Close Method	68
streamwriter_object.Flush Method	69
streamwriter_object.NewLine Property	70
streamwriter_object.Write Method	71
streamwriter_object.WriteLine Method	72
Functions	73
Function Summary	73
CBool Function	74
CByte Function	76
CDBl Function	78

GPL Dictionary Pages

CInt	Function	80
CShort	Function	82
CSng	Function	84
CStr	Function	86
Fix	Function	88
Hex	Function	90
Int	Function	92
Rnd	Function	94

Location Class..... 96

Location Class Summary	96
location_object.Angle Property	98
location_object.Angles Method	99
location_object.Clone Method	100
location_object.Config Property	101
Location.Distance Method	103
location_object.Here Method	104
location_object.Here3 Method	106
location_object.Inverse Method	108
location_object.KineSol Method	110
location_object.Mul Method	112
location_object.Normalize Method	114
location_object.Pitch Property	115
location_object.Pos Property	117
location_object.PosWrtRef Property	119
location_object.RefFrame Property	121
location_object.Roll Property	122
location_object.Type Property	124
location_object.X Property	125
location_object.XYZ Method	127
location_object.XYZInc Method	129
Location.XYZValue Method	130
location_object.Y Property	132
location_object.Yaw Property	134
location_object.Z Property	136
location_object.ZClearance Property	138
location_object.ZWorld Property	140

Math Class 142

Math Class Summary	142
Math.Abs Method	144
Math.Acos Method	145
Math.Asin Method	146
Math.Atan Method	147
Math.Atan2 Method	148
Math.Ceiling Method	149
Math.Cos Method	150
Math.Cosh Method	151
Math.E Method	152
Math.Exp Method	153
Math.Floor Method	154
Math.Log Method	155
Math.Log10 Method	156
Math.Max Method	157
Math.Min Method	158
Math.PI Method	159
Math.Pow Method	160
Math.Sign Method	161
Math.Sin Method	162
Math.Sinh Method	163
Math.Sqrt Method	164
Math.Tan Method	165
Math.Tanh Method	166

Modbus Class 167

Modbus Class Summary	167
modbus_object.Close Method	168
modbus_object.ReadCoils Method	169
modbus_object.ReadDeviceID Method	170
modbus_object.ReadDiscreteInputs Method	172
modbus_object.ReadHoldingRegisters Method	173
modbus_object.ReadInputRegisters Method	175
modbus_object.Timeout Property	177
modbus_object.WriteMultipleCoils Method	178
modbus_object.WriteMultipleRegisters Method	179

GPL Dictionary Pages

modbus_object.WriteSingleCoil Method	180
modbus_object.WriteSingleRegister Method	181
Move Class	182
Move Class Summary	182
Move.Approach Method	184
Move.Arc Method	186
Move.Circle Method	189
Move.Delay Method	192
Move.Extra Method	193
Move.ForceOverlap Method	195
Move.Loc Method	198
Move.OneAxis Method	200
Move.Rel Method	202
Move.SetJogCommand Method	204
Move.SetSpeeds Method	206
Move.SetTorques Method	208
Move.StartJogMode Method	210
Move.StartTorqueCntrl Method	212
Move.StartVelocityCntrl Method	214
Move.StopSpecialModes Method	217
Move.Trigger Method	218
Move.WaitForEOM Method	220
Networking Classes	221
Networking Classes Summary	221
New IPEndPoint Constructor	223
ipendpoint_object.IPAddress Property	224
ipendpoint_object.Port Property	225
socket_object.Available Property	226
socket_object.Blocking Property	227
socket_object.Close Method	228
socket_object.Connect Method	229
socket_object.Receive Method	230
socket_object.ReceiveFrom Method	231
socket_object.ReceiveTimeout Property	233
socket_object.Send Method	234

Table Of Contents

socket_object.SendTimeout Property	235
socket_object.SendTo Method	236
New TcpClient Constructor	238
tcpclient_object.Client Method	239
tcpclient_object.Close Method	240
New TcpListener Constructor	241
tcplistener_object.AcceptSocket Method	242
tcplistener_object.Close Method	243
tcplistener_object.Pending Property	244
tcplistener_object.Start Method	245
tcplistener_object.Stop Method	246
New UdpClient Constructor	247
udpclient_object.Client Method	248
udpclient_object.Close Method	249
Profile Class	250
Profile Class Summary	250
profile_object.Accel Property	251
profile_object.AccelRamp Property	253
profile_object.Clone Method	255
profile_object.Decel Property	256
profile_object.DecelRamp Property	258
profile_object.InRange Property	260
profile_object.Speed Property	262
profile_object.Speed2 Property	264
profile_object.Straight Property	266
Reference Frame Class	268
RefFrame Class Summary	268
refframe_object.Loc Property	270
refframe_object.PalletIndex Property	272
refframe_object.PalletMaxIndex Property	274
refframe_object.PalletNextPos Method	276
refframe_object.PalletOrder Property	277
refframe_object.PalletPitch Property	279
refframe_object.PalletRowColLay Method	280
refframe_object.Pos Method	282

GPL Dictionary Pages

reframe_object.PosWrtRef Method	283
reframe_object.Type Property	284
Robot Class	286
Robot Class Summary	286
Robot.Attached Property	288
Robot.Base Property	289
Robot.Custom Property	291
Robot.DefLinComp Method	293
Robot.Dest Property	295
Robot.DestAngles Property	297
Robot.Home Method	299
Robot.HomeAll Method	300
Robot.LastProfile Property	301
Robot.RapidDecel Property	302
Robot.RestartBase Property	303
Robot.RestartTool Property	304
Robot.Selected Property	305
Robot.Source Property	306
Robot.SourceAngles Property	308
Robot.Tool Property	310
Robot.TrajState Property	312
Robot.Where Property	314
Robot.WhereAngles Property	316
Signal Class	318
Signal Class Summary	318
Signal.AIO Property	319
Signal.DIO Property	321
Statements	324
Statements Summary	324
Call Statement	325
Class Statement	327
Const Statement	328
Dim Statement	329
Do...Loop Statements	331

Table Of Contents

Else, ElseIf Statements	333
End Statements	334
Exit Statements	335
For...Next Statements	336
Function Statement	339
Get Statement	342
GoTo Statement	343
If..Then...Else...End If Statements	345
Loop Statements	347
Module Statement	348
Next Statements	349
Property Statement	350
ReDim Statement	353
Return Statement	354
Set Statement	355
Sub Statement	357
While...End While Statements	359
Strings	361
String Summary	361
String.Compare Method	363
string.IndexOf Method	365
string.Length Property	367
string.Split Method	368
string.Substring Method	369
string.ToLower Method	370
string.ToUpper Method	371
string.Trim Method	372
string.TrimEnd Method	373
string.TrimStart Method	374
Asc Function	375
Chr Function	376
Format Function	377
Instr Function	380
LCase Function	382
Len Function	383
Mid Function	384

UCase Function	385
Thread Class.....	386
Thread Class Summary	386
New Thread Constructor	387
thread_object.Abort Method	389
Thread.CurrentThread Shared Method	390
thread_object.Join Method	391
thread_object.Resume Method	392
thread_object.SendEvent Method	393
Thread.Sleep Shared Method	394
thread_object.Start Method	395
thread_object.Suspend Method	396
thread_object.ThreadState Property	397
Thread.WaitEvent Shared Method	398
Vision Classes.....	401
Vision Classes Summary	401
Vision.Disconnect Method	403
vision_object.ErrorCode Property	404
vision_object.Process Method	405
vision_object.Result Method	407
vision_object.ResultCount Method	409
vision_object.Status Property	411
Vision.ToolProperty Shared Property	412
visresult_object.ErrorCode Property	413
visresult_object.Info Property	414
visresult_object.InfoCount Property	415
visresult_object.InspectActual Property	416
visresult_object.InspectPassed Property	417
visresult_object.Loc Property	418
visresult_object.Type Property	420

GPL Dictionary Pages Summary

The Guidance Programming Language Dictionary Pages provide detailed information on each instruction, keyword, function, and class property and method that is available in GPL. For convenience, these descriptions are group either by their class or by their major function. Within each group they are sorted alphabetically.

In general, instruction names, keywords, function names, group names, and property and method names are indicated in **bold**. User specified variable names are indicated in *italics*. Sample GPL program snippets are presented in the Courier font.

The following table summarizes each of the major groups of descriptions.

Group	Description
<u>Array Class</u>	Provides the properties of any type of variable array.
<u>Console Class</u>	Provides methods for performing output to the serial console or to the GDE console window.
<u>Controller Class</u>	Provides access to general facilities provided by the motion control hardware such as power control, timers, etc.
<u>Exception Handling</u>	Includes statements for fielding execution exceptions and the Exception Class for storing exception information.
<u>File and Serial I/O Classes</u>	Provides File , StreamReader and StreamWriter classes that implement file and serial line input and output communications.
<u>Functions</u>	Includes standard functions, such as conversion routines, that do not fall into a specific class.
<u>Location Class</u>	Defines positions and orientations of the robot and objects.
<u>Math Class</u>	Provides the standard arithmetic and trigonometric functions.
<u>Modbus Class</u>	Permits programs to communicate with other intelligent devices using the MODBUS/TCP Ethernet communication protocol.
<u>Move Class</u>	Provides the basic methods for executing a motion between Locations using Profiles .
<u>Networking Classes</u>	Classes for Ethernet network communications. Includes IPEndPoint Class for specifying IP and port addresses; Socket Class that provides basis for networking I/O operations; TcpListener Class for TCP server applications; TcpClient Class for TCP client applications; and UdpClient Class for UDP server and client applications.
<u>Profile Class</u>	Defines sets of parameters that specify the trajectory to be followed when moving between Locations .
<u>RefFrame Class</u>	Defines robot and part reference frames. Cartesian Locations and RefFrames can be defined with respect to a RefFrame .
<u>Robot Class</u>	Provides access to the attributes and properties of each robot such as their current position and homing methods.
<u>Signal Class</u>	Reads and writes digital, analog and other simple means

	of input and output.
<u>Statements</u>	Includes control structures, user procedures and functions, and other common language elements.
<u>Strings</u>	Provides String manipulation methods in an Object oriented fashion.
<u>Thread Class</u>	Provides the means for starting, stopping, and monitoring the execution of independent threads.
<u>Vision Classes</u>	Provides the means for interfacing to PreciseVision and easily generating vision-guided motion applications.

Array Class

Array Class Summary

The following pages provide detailed information on the properties and methods of the **Array Class**.

Array variables of all types (e.g. **Strings**, **Locations**, **Integers**) are members of the built-in **Array Class**. You can use the properties of this class to determine the properties of an array.

The table below briefly summarizes the properties and methods for this class, which are described in greater detail in the sections that follow.

Member	Type	Description
<u>array.GetUpperBound</u>	Get Property	Returns the upper bound for a particular dimension of an array. The lower bound is always 0, so the total number of elements in this dimension is one greater than the upper bound.
<u>array.Length</u>	Get Property	Returns the total number of elements in the entire array, in all dimensions.
<u>array.Rank</u>	Get Property	Returns the array rank, which is the number of dimensions in an array.

array.GetUpperBound Property

Returns the maximum allowed array index for a particular dimension of an array.

```
...array.GetUpperBound( dimension )
```

Prerequisites

None

Parameters

dimension

A required numeric expression that specifies the index, from 0 to *rank*-1, of the dimension whose upper bound should be returned.

Remarks

In GPL, all array dimension indices start at 0 and end at the upper bound. This upper bound is the same value specified in a **Dim** or **ReDim** statement. The number of elements in an array dimension is 1 plus the upper bound value.

Examples

```
Dim array(3,4) As Integer
Dim d1, d2 As Integer
d1 = array.GetUpperBound(0)      ' Returns the value 3
d2 = array.GetUpperBound(1)      ' Returns the value 4
```

See Also

[Array Class](#) | [array.Length](#) | [Dim Statement](#) | [ReDim Statement](#)

array.Length Property

Returns the total number of elements in an entire array.

`...array.Length`

Prerequisites

None

Parameters

None

Remarks

In GPL, all array dimension indices start at 0 and end at the upper bound. The **Length** may be computed by multiplying (1+upper bound) of all array dimensions.

Do not be confused when using the **Length** property with **String** arrays. For example, if you declare: **Dim sarray(3) As String**.

`sarray.Length` is the number of elements in the array, in this case 4 (from 0 to 3).

`sarray(0).Length` is the length of the string contained in `sarray(0)`, initially 0.

Examples

```
Dim array(3,4) As Integer
Dim length As Integer
length = array.Length           ' Returns the value 20 = (1+3)*(1+4)
```

See Also

[Array Class](#) | [array.GetUpperBound](#) | [Dim Statement](#) | [ReDim Statement](#)

array.Rank Property

Returns the total number of dimensions (the rank) in the array.

```
...array.Rank
```

Prerequisites

None

Parameters

None

Remarks

The **Rank** of an array is the number of dimensions in that array.

Examples

```
Dim array(3,4) As Integer
Dim array2(5) As Integer
Dim r1, r2 As Integer
r1 = array.Rank           ' Returns 2
r2 = array2.Rank         ' Returns 1
```

See Also

[Array Class](#) | [Dim Statement](#) | [ReDim Statement](#)

Console Class

Console Class Summary

The following pages provide detailed information on the methods of the global **Console Class**. These methods support simple output to the GPL console.

The actual destination of console output depends on the presence of the *-event* switch on the **Start** console command. If *-event* is not present, console output is sent to the first serial port named `"/dev/com1"`. If *-event* is present, console output is sent to GDE where it is displayed in the GPL Output window.

The table below briefly summarizes the properties and methods for this class, which are described in greater detail in the sections that follow.

Member	Type	Description
Console.Write	Shared Method	Writes a number or a string to the console.
Console.WriteLine	Shared Method	Writes a number or a string to the console, followed by a line feed (LF) character.

Console.Write Method

Writes a numeric or string value to the GPL console with no line terminator.

```
Console.Write (number)  
-or-  
Console.Write (string)
```

Prerequisites

None

Parameters

number

A required numeric expression whose value is displayed.

string

A required string expression whose value is displayed.

Remarks

This method writes a single numeric or string value to the GPL console with no line terminator. Subsequent output continues on the same line. For output that combines both string and numeric values, use the **CStr** function.

The actual destination of console output depends on the presence of the *-event* switch on the **Start** console command. If *-event* is not present, console output is sent to the first serial port named `"/dev/com1"`. If *-event* is present, console output is sent to GDE where it is displayed in the GPL Output window.

Examples

```
Console.Write("Test ")      ' Produces the output: "Test 1"  
Console.Write(1)
```

See Also

[Console Class](#) | [Console.WriteLine](#) | [CStr Function](#) | [StreamWriter Class](#)

Console.WriteLine Method

Writes a numeric or string value to the GPL console followed by a line terminator.

Console.WriteLine (*number*)
 -or-
Console.WriteLine (*string*)

Prerequisites

None

Parameters

number

A required numeric expression whose value is displayed.

string

A required string expression whose value is displayed.

Remarks

This method writes a single numeric or string value to the GPL console followed by a line terminator. Subsequent output appears on the next line. For output that combines both string and numeric values, use the **CStr** function.

The actual destination of console output depends on the presence of the *-event* switch on the **Start** console command. If *-event* is not present, console output is sent to the first serial port named "/dev/com1". If *-event* is present, console output is sent to GDE where it is displayed in the GPL Output window.

Examples

```
Console.WriteLine("Test ") ' Produces the output: Test
Console.WriteLine(1)      ' 1

Dim ii As Integer
For ii = 1 To 10
    Console.WriteLine("The square of " & CStr(ii) _
        & " is " & CStr(ii*ii))
Next ii
```

See Also

[Console Class](#) | [Console.Write](#) | [CStr Function](#) | [StreamWriter Class](#)

Controller Class

Controller Class Summary

The following pages provide detailed information on the properties and methods of the global **Controller Class**. This class provides access to the general facilities provided by the Guidance Controller, e.g. high power control, E-Stop logic, configuration database values, etc. As such, this class and all of its members are uniquely defined for Precise controller products and do not conform to any other standards. In the case of certain methods, such as the **SleepTick**, very similar functionality is provided by other means within the Basic language. However, the members of this class were selected based upon ease-of-use considerations or because they provide some slightly different, but important, functionality.

As is standard in GPL, conversions between different arithmetic types, e.g. **Integer**, **Single**, **Double**, are automatically performed as required. So, for numeric properties and methods of the **Controller Class**, it is not necessary to have different variations of these members to deal with the different possible mixes of input parameter data types. Also, as appropriate, the properties and methods generally produce results that are formatted as **Double**'s. These results will automatically be converted to smaller data types as necessary, e.g. **Double** -> **Integer**, and will not generate an error so long as numeric overflow does not occur.

The table below briefly summarizes the properties and methods that are described in greater detail in the following sections.

Member	Type	Description
Controller.ErrorLog	Property	Returns an entry from the system Error Log as a String value or clears the Error Log.
Controller.Load	Method	Loads a GPL project into memory and compiles it in preparation for execution.
Controller.PDb	Property	Sets and gets any accessible value in the configuration parameter database.
Controller.PDbNum	Property	Optimized means to set and get a numeric value in the configuration parameter database.
Controller.PowerEnabled	Property	Sends a request to either turn on or off high (motor) power to the amplifier. Returns whether high power is on or off.
Controller.PowerState	Property	Gets the current state of the high power sequence.
Controller.RecordButton	Property	Sets and gets the latched Boolean value that indicates if the hardware MCP RECORD button has been pressed.
Controller.ShowDialog	Method	Displays a pop-up dialog box on the web Operator Control Panel.
Controller.ShowDialogMCP	Method	Displays a pop-up dialog box on the LCD display of the Precise Hardware Manual Control Pendant.
Controller.SleepTick	Method	Delays further execution of a thread for a specified number of Trajectory Generator

		periods.
<u>Controller.SoftEStop</u>	Property	Sets and gets the Boolean flag that triggers a Soft E-Stop.
<u>Controller.SystemMessage</u>	Method	Enters a message into the GPL system message log that is displayed on the web Operator Control Panel.
<u>Controller.Tick</u>	Property	Gets the execution repetition period for the Trajectory Generator.
<u>Controller.Timer</u>	Property	Gets the value of the controller's usec clock in units of seconds.
<u>Controller.Unload</u>	Method	Unloads an idle GPL project from memory.

Controller.ErrorLog Property

Returns an entry from the system Error Log as a **String** value or clears the Error Log.

```
Controller.ErrorLog = <value>
-or-
... Controller.ErrorLog( entry)
```

Prerequisites

None

Parameters

entry

A required numeric expression that specifies the **Integer** number of the Error Log entry to be returned. This value can range from 1 to n, where 1 indicates that the most recent entry should be returned.

Remarks

Whenever a runtime error occurs in the system, the error is time stamped and entered into the system Error Log. These errors can be generated by an executing thread or from the motion control system. In addition, GPL applications can enter items into the log using the **Controller.SystemMessage** method.

The entries in the Error Log are displayed on the web based Operator Control Panel and can be retrieved from the console interface.

This method permits GPL programs to retrieve entries from the Error Log one at a time. Each returned value contains the time stamp, marker indicating the thread that generated the error, the numeric error code and the text error message. A example of a typical returned value is as follows:

```
04-09-2007 12:27:14.223, Trj, -1611, "**Auto/Manual switch set to Manual**"
```

If you request an entry that does not exist, an empty string value is returned. Also, if a new entry is added to the log or the log is cleared as you are scanning through the log, you may get an inconsistent set of error entries.

If this property is assigned a non-zero value as indicated above, rather than retrieving an entry, all entries are deleted from the Error Log.

Examples

```
Dim err As String
Dim ii As Integer
```

```
For ii = 1 To 100
    err = Controller.ErrorLog(ii)      ' Retrieve all entries from log
    If (err <> "") Then
        Console.WriteLine(err)        ' Display all errors
    Else
        Exit For                      ' No more entries in the log
    End If
Next
Controller.ErrorLog = 1               ' Clear all entries in the log
```

See Also

[Controller Class](#) | [Controller.SystemMessage](#)

Controller.Load Method

Loads the files associated with a GPL project into memory and compiles them so that the project procedures are ready to be executed.

```
Controller.Load(project_folder_path)
```

Prerequisites

The project folder must contain a valid project file named *Project.gpr*. This project file describes all the remaining files within the project. The project must not be currently loaded.

Parameters

project_folder_path

A required string expression that specifies the name of the folder that contains the project to be loaded. Normally the folder is located on the "/flash" device.

Remarks

This method loads a project by first creating a folder in the controller's memory section that is allocated for GPL projects. Then, all of the files associated with the project are copied into the memory folder. Finally, the project is compiled so that the loaded procedures are ready to be executed.

No compilation errors are displayed on the console. Examine the file */GPL/project_name/Compile.log* for a listing of compiler messages.

This method will throw an exception if the project cannot be loaded, if it is already loaded, or if compilation errors occur.

Examples

```
Dim th As Thread
Controller.Load("/flash/projects/Test")
th = New Thread("Main", "Test", "Thread2")
th.Start()
```

See Also

[Controller Class](#) | [Controller.Unload](#) | [Thread.Start](#)

Controller.PDb Property

Sets and gets any accessible value in the configuration parameter database.

```
Controller.PDb(dataid, unit, unit2, array_index) = <new_string_value>
-or-
... Controller.PDb(dataid, unit, unit2, array_index)
```

Prerequisites

None

Parameters

dataid

A required numeric expression that specifies an **Integer** identification code for the parameter to be accessed. For example, the parameter for setting the system “test speed” is 601.

unit

An optional numeric expression that specifies an **Integer** unit number for the parameter to be accessed. For many parameters, e.g. the Controller, only a single unit exists. For parameters that refer to devices with multiple possible units, e.g. multiple robots driven by a single controller, this parameter ranges from 1 to n. If not specified, this value defaults to 1.

unit2

An optional numeric expression that specifies an **Integer** sub unit number for the parameter to be accessed. The use of the sub unit number is not very common and this parameter is normally just defaulted to 1.

array_index

An optional numeric expression that specifies an **Integer** array index for parameters that have multiple values. For example, for a robot with multiple axes, the “joint maximum soft stop limits” (*dataid* 16077) is an array with one value for each joint. If not specified, this value defaults to 0, which reads all possible array values.

Remarks

As described in the Controller Software Introduction, all of the key variables for configuring and monitoring the operation of the system are stored in a unified parameter

database. **Controller.PDb** can be used to read or write all accessible values in the parameter database.

Controller.PDb reads parameters and returns the results in a **String** or writes parameters by accepting a **String** expression. If the parameter contains numeric values, the values are represented as text numbers separated by commas (in the case of numeric arrays). If the parameter contains a single string value, the value is read into or read from a GPL **String** without delimiting quotation marks. If the parameter contains an array of strings, each string is delimited by double quotes and sequential values are separated by commas.

As a convenient for developing custom web pages, the parameter database contains a series of "GPL program strings" (DataID's 1800-1819) and "GPL program variable's" (DataID's 1850-1869). Custom web pages can read and write these values via ASP operations. Once the controller is restarted, the operating system does not alter any of these variable values.



WARNING: While database values can be freely read, care should be taken when writing to general database parameters. Unintentionally altering some values may cause the system to not operate properly.

Examples

```
Dim stg As String

Controller.PDb(541) = "" "Label1""",""Label2"""" ' Sets first two DOUT labels

stg = Controller.PDb(100) ' stg set to "Precise Automation
Inc"
```

See Also

[Controller Class](#) | [Controller.PDbNum](#)

Controller.PDbNum Property

Optimized means for setting and getting a numeric value in the configuration parameter database.

```
Controller.PDbNum(dataid, unit, unit2, array_index) = <new_value>
-or-
... Controller.PDbNum(dataid, unit, unit2, array_index)
```

Prerequisites

Can only access numeric parameter database values.

Parameters

dataid

A required numeric expression that specifies an **Integer** identification code for the parameter to be accessed. For example, the parameter for setting the system “test speed” is 601.

unit

An optional numeric expression that specifies an **Integer** unit number for the parameter to be accessed. For many parameters, e.g. the Controller, only a single unit exists. For parameters that refer to devices with multiple possible units, e.g. multiple robots driven by a single controller, this parameter ranges from 1 to n. If not specified, this value defaults to 1.

unit2

An optional numeric expression that specifies an **Integer** sub unit number for the parameter to be accessed. The use of the sub unit number is not very common and this parameter is normally just defaulted to 1.

array_index

An optional numeric expression that specifies an **Integer** array index for parameters that have multiple values. For example, for a robot with multiple axes, the “joint maximum soft stop limits” (*dataid* 16077) is an array with one value for each joint. If not specified, this value defaults to 1, the first array element.

Remarks

As described in the Controller Software Introduction, all of the key variables for configuring and monitoring the operation of the system are stored in a unified parameter

database. **Controller.PDbNum** is an variation of **Controller.PDb** that has been optimized to efficiently read and write numeric values stored in this database.

In addition to generally efficient operation, **Controller.PDbNum** operates especially quickly when reading and writing the "GPL program variable's" (DataID's 1850-1869). These database elements have been created to allow GPL projects to interface to custom web pages. Custom web pages can read and write these values via ASP operations. Once the controller is restarted, the operating system does not alter any of these variable values.



WARNING: While database values can be freely read, care should be taken when writing to general database parameters. Unintentionally altering some values may cause the system to not operate properly.

Examples

```
Dim limit As Single
limit = Controller.PDbNum(16077,,,2) ' Sets limit equal to the maximum
                                     ' allowable range of travel for jt 2
```

See Also

[Controller Class](#) | [Controller.PDb](#)

Controller.PowerEnabled Property

Sends a request to either turn on or off high (motor) power to the amplifier. Returns whether high power is on or off.

```
Controller.PowerEnabled = <boolean_value>
-or-
Controller.PowerEnabled(timeout) = <boolean_value>
-or-
... Controller.PowerEnabled
```

Prerequisites

Enabling power via this software command is not permitted on Category 3 safe systems. For Category 3 systems, a momentary contact, hardware “Enable Power” button must be manually pressed.

Parameters

timeout

An optional numeric value that specifies the maximum time, in seconds, to wait for power to come on. If less than or equal to zero or omitted, this property waits forever.

Remarks

Setting the **PowerEnabled** property **True** sends a request to the control system to enable high power to the amplifiers. For non-Category 3 safe systems, high power will be enabled only if a number of safety conditions are satisfied (e.g. no Hard E-Stop signal is asserted, no fatal system error exists, etc.). This property waits until the power actually comes on, with a time limit determined by the *timeout* parameter. If this parameter is positive and the power does not come on within the time limit, this property throws an exception that indicates why power did not come on.

Setting the **PowerEnabled** property **False** turns off high power to the amplifiers, but the property does not wait until power is actually off. Unlike the Hard E-Stop signal that delays for a fixed period of time before disabling power, turning off **PowerEnabled** forces all moving robots to completely decelerate to a stop and allows time for the brakes to be set before power to the amplifiers is disabled. Therefore, setting **PowerEnabled False** allows for a more orderly stopping of motion than does a Hard E-Stop but this operation is consequently somewhat slower.

The **PowerEnabled** property is automatically set to **False** by the system if High Power is disabled by any means and is automatically set to **True** if High Power is enabled.

Examples

GPL Dictionary Pages

```
Dim bState As Boolean
Controller.PowerEnabled = True
Controller.PowerEnabled(5) = True

bState = Controller.PowerEnabled
```

' Requests high power be enabled
' Requests high power be enabled
' and waits for up to 5 seconds
' bState will be set **True** if power is
' enabled, else will be set **False**.

See Also

[Controller Class](#) | [Controller.PowerState](#) | [Controller.SoftEstop](#) | [Robot.RapidDecel](#)

Controller.PowerState Property

Reads and returns an **Integer** value that indicates the current state of the amplifier high power sequencing.

... **Controller.PowerState**

Prerequisites

None

Parameters

None

Remarks

In order to enable high power to the amplifiers, the system must transition in an orderly fashion through several states to ensure that safety and hardware requirements are satisfied. The **PowerState** property indicates the current state of the power sequencing.

The possible values returned by this property and their interpretation are presented in the following table:

PowerState	Description
0	System initially starting up
1	Power off, fatal error has occurred
2	Power off, power sequence restarting
3	Power being turned off, no fault condition has occurred
4	Power being turned off, a fault condition has occurred
5	Power is off, a fault has occurred that must be cleared
6	Power is off, waiting for hardware enable power switch to be turned off
7	Power is off, waiting for enable power signal to be asserted
8	Power is coming up, enabling amplifiers
9	Power is on, performing motor commutation
10	Power is coming up, enabling servos and releasing brakes
11	Power is on, waiting to execute thread or Auto Execution task
12	Power is on, executing Auto Execution task

Examples

```
Dim state As Integer
state = Controller.PowerState ' Sets state to one of the values listed above
```

See Also

[Controller Class](#) | [Controller.PowerEnabled](#) | [Controller.SoftEstop](#) | [Robot.RapidDecel](#)

Controller.RecordButton Property

Reads and writes the latched **Boolean** value that indicates if the hardware MCP RECORD button has been pressed.

```
Controller.RecordButton = <boolean_value>
-or-
... Controller.RecordButton
```

Prerequisites

None

Parameters

None

Remarks

Whenever the RECORD key on the Precise Hardware Manual Control Pendant (MCP) is pressed, the value of this property is automatically set to **True**. This property value remains **True** until it is manually set to **False**.

The RECORD key on the MCP and this property provide a convenient means for GPL projects to receive a command from the operator to record key data, typically taught robot locations.

The value of this property can also be accessed via the Parameter Database as the "MCP Record button pressed" (DataID 632) value.

Examples

```
Dim taught_loc As New Location
If (Controller.RecordButton) Then
    taught_loc.Here ' Save current robot location
    Controller.RecordButton = False
End if
```

See Also

[Controller Class](#)

Controller.ShowDialog Method

Displays a pop-up dialog box on the web interface Operator Control Panel.

```
Controller.ShowDialog( button_labels, message, button_index)
-or-
Controller.ShowDialog( button_labels, message, button_index, text_field )
-or-
Controller.ShowDialog( mode, button_labels, message, button_index, field_labels,
field_values )
```

Prerequisites

None

Parameters

mode

(3rd form of this method) A required numeric expression that specifies the display mode. Currently only the value 1 is supported, which indicates a simple list.

button_labels

A required **String** expression containing the button labels to be displayed. Up to 4 buttons can be specified, separated by commas. If the button labels contain blanks or commas, they should be enclosed in quotes. The string must not contain the vertical bar "|" character.

message

A required **String** expression containing the message to be displayed in the dialog box. The string must not contain the vertical bar "|" character.

button_index

A required **ByRef Integer** variable that receives the index of the button pressed in the dialog box. Set to 1 for the first button, 2 for the second, etc.

text_field

(2nd form of this method) An optional **ByRef String** variable that receives the value of any text entered into the dialog box text field. Its initial value is shown as the default value of the text field. The string must not contain the vertical bar "|" character.

field_labels

(*3rd form of this method*) A required 1-dimensional **String** array that contains labels to be displayed preceding each data field in the dialog box. Each **String** array element contains a label for a separate field. Up to to 12 fields are permitted. The number of elements in this array determines the number of fields. The **Strings** must not contain the vertical bar "|" character.

field_values

(*3rd form of this method*) A required 1-dimensional **String** array that receives the value of any text entered into the dialog box text fields. The initial values of this array are displayed as the default values of the text fields. The **Strings** must not contain the vertical bar "|" character.

Remarks

This method provides a simple way for a GPL procedure to communicate with the operator without creating a custom web page. When **ShowDialog** is called, its operation is as follows:

- 1.
2. Waits if another thread is already displaying a dialog box.
3. Posts the dialog box for display and waits for the user to open the Operator Control Panel on the web interface and click on a button.
4. Un-displays the dialog box.
5. Returns the button index and optional text field information to the user.

Since this method generates a dialog box within a browser, any special text formatting must be defined as standard HTML specifications. In particular, to add a carriage return and line feed, include "
" within the text. To have a section of text left justified, precede it with "<p align=left>" and terminate it with "</p>".

This method is overloaded to support several dialog box styles.

In the simplest (1st) form, the pop-up displays only the *message* text and labeled buttons. When the user clicks on one of the buttons, the index of the button clicked is returned in the *button_index* variable.

In the *text_field* (2nd) form, the pop-up also displays a single text field that can be overwritten by the user. When the user clicks on one of the buttons, the current value of the text field is returned in the *text_field* variable, and the index of the button clicked is returned in the *button_index* variable.

In the more complex *field_values* and *field_labels* (3rd) form, the dialog box allows multiple fields to be entered and returned. The *mode* parameter selects the display mode and must currently be set to 1. When the user clicks on one of the buttons, the values of the fields are returned in the *field_values* array, and the index of the button clicked is returned in the *button_index* variable.

If the thread displaying the dialog box is paused or stopped, the dialog box is un-displayed immediately.

Examples

```
Dim bi As Integer
Controller.ShowDialog("Okay", "Ready to begin process", bi)

Public Sub Test1
    Dim bi As Integer
    Dim reply As String
    reply = "Part_1" ' Default is Part_1
    Controller.ShowDialog("Okay, Cancel", _
        "Enter part name", bi, reply)
    If bi = 1 Then
        ... ' Okay selected
    Else
        ... ' Cancel selected
    End If
    Console.WriteLine("You entered: " & reply)
End Sub

Public Sub Test2
    Dim Buttons As String = "Okay, Cancel"
    Dim Text As String = "Enter the field values"
    Dim Label(2) As String
    Dim Field(2) As String
    Dim Index As Integer

    Label(0) = "X value"
    Label(1) = "Y value"
    Label(2) = "Z value"

    Field(0) = "100.0"
    Field(1) = "100.0"
    Field(2) = "0.0"

    Controller.ShowDialog(1, Buttons, Text, Index, Label, Field)

    Console.WriteLine("Button: " & CStr(Index))
    Console.WriteLine("Field 0: " & Field(0))
    Console.WriteLine("Field 1: " & Field(1))
    Console.WriteLine("Field 2: " & Field(2))
End Sub
```

See Also

[Controller Class](#) | [Controller.ShowDialogMCP](#) | [Controller.SystemMessage](#)

Controller.ShowDialogMCP Method

Displays a pop-up dialog box on the LCD display of the Precise Hardware Manual Control Pendant.

```
Controller.ShowDialogMCP( button_mask, message, button_return)  
-or-  
Controller.ShowDialogMCP( button_mask, message, button_return, text_field )
```

Prerequisites

Precise Hardware Manual Control Pendant must be connected to the controller.

Parameters

button_mask

A required **Integer** expression whose bits specify the MCP key presses that will terminate the dialog box. A value of -1 indicates that the maximum number of keys are permitted to terminate the dialog process.

message

A required **String** expression containing the message to be displayed on the LCD display. If a *text_field* is specified, the *message* must include a substring ('##...##') that defines where the characters of the *text_field* are output in the MCP display. The number of pound signs (#) defines the width of the input field.

button_return

A required **ByRef Integer** variable that receives the bit flag that indicates the button that was pressed to terminate the dialog operation.

text_field

An optional **ByRef String** variable that receives the value of any text entered into the dialog box text field. The initial value of this variable is displayed as the default value of the text field. Given the key pad layout of the Precise MCP, the *text_field* can only contain a numeric value that consists of 0-9, ., + or - characters.

Remarks

This method provides a simple way for a GPL procedure to communicate with the operator via the Precise Hardware Manual Control Pendant. When **ShowDialogMCP** is called, its operation is as follows:

- 1.

2. Waits if another thread is already displaying a MCP dialog box.
3. Replaces the standard MCP display with the contents of the *message* and the optional embedded *text_field*, and lights the LED on the APP key.
4. If the optional *text_field* is defined, accepts presses of the 0-9, ., +, - or DEL keys and presents the results in the LCD display.
5. If the display and keypad are switched back to their standard mode due to a manual control operation or error message, blinks the APP key LED until the APP key is pressed to re-display the dialog.
6. When one of the specified termination keys is pressed, un-displays the dialog box.
7. Returns the termination key button bit flag and the optional text field value.

The MCP keypad buttons that can be specified to terminate the dialog mode are listed in the following table together with their associated *button_mask* and *button_return* values.

Key Label	<i>button_mask</i> & <i>button_return</i>
Enter	&H000001
Record	&H000002
Yes	&H000004
No	&H000008
Quit	&H000010
Prev	&H000020
Next	&H000040
F1	&H010000
F2	&H020000
F3	&H040000
F4	&H080000

By default, when a dialog is first displayed on the MCP, a beep is generated to alert the operator. The beeping operation can be suppressed by resetting the "Beep MCP when APP mode started" (DataID 636) system parameter.

If the thread displaying the dialog box is paused or stopped, the dialog box is un-displayed immediately.

Examples

```

Dim but As Integer
Dim ss, CRLF As String
CRLF = Chr(GPL_CR) & Chr(GPL_LF)
ss = " Ready to begin" & CRLF & CRLF _
    & " <Yes> or <No>"
Controller.ShowDialogMCP(&H4+&H8, ss, but)

Dim but As Integer
Dim reply, ss, CRLF As String
CRLF = Chr(GPL_CR) & Chr(GPL_LF)
ss = " Enter part number:" & CRLF _
    & " '#####'" & CRLF & CRLF _
    & " <Enter> or <Quit>"
reply = "12" ' Default reply value
Controller.ShowDialogMCP(&H1+&H10, ss, but, reply)
If but = &H10 Then

```

```
        Console.WriteLine("Request cancelled")
    Else
        Console.WriteLine("You entered: " & reply)
    End If
```

See Also

[Controller Class](#) | [Controller.ShowDialog](#) | [Controller.SystemMessage](#)

Controller.SleepTick Method

Delays further execution of a thread for a specified number of Trajectory Generator periods.

Controller.SleepTick(*ticks*)
-or-
Controller.SleepTick

Prerequisites

None

Parameters

ticks

An optional numeric expression that specifies an **Integer** number of Trajectory Generator periods that execution is to be delayed. If this parameter is not specified, the value is defaulted to 1.

Remarks

Often times, a program must poll input data values periodically. While it is possible to use a “busy loop” that counts for a fixed number of times, this technique unnecessary consumes CPU time that could be more productively spent by system drivers or other GPL threads. The **SleepTick** method allows a thread to relinquish control of the CPU for a specified period of time and then resume execution at the next sequential statement.

Since many operations are synchronized to the operation of the Trajectory Generator, the delay time for this method is specified in units of Trajectory Generator execution periods.

Please note that other programming languages like Basic typically have other means for putting a thread to sleep for a specified period of time.

Examples

Controller.SleepTick	' Delays thread execution until
	' after the start of the next
	' trajectory cycle
Controller.SleepTick (2/Controller.Tick)	' Delays thread execution for
	' approximately 2 seconds

See Also

[Controller Class](#) | [Controller.Tick](#) | [Controller.Timer](#)

Controller.SoftEStop Property

Reads and writes the **Boolean** value that triggers a Soft E-Stop condition when **True**.

```
Controller.SoftEStop = <boolean_value>
-or-
... Controller.SoftEStop
```

Prerequisites

None

Parameters

None

Remarks

A Soft E-Stop initiates a rapid deceleration of all robots currently in motion and generates an error condition for all GPL programs that are attached to a robot. This property can be used to quickly halt all robot motions in a controlled fashion when an error is detected.

This function is similar to a Hard E-Stop except that Soft E-Stop leaves High Power enabled to the amplifiers and is therefore used for less severe error conditions. Leaving power enabled is beneficial in that it prevents the robot axes from sagging and does not require high power to be manually re-enabled before program execution and robot motions are resumed. This function is also similar to a Rapid Deceleration feature except that a Rapid Deceleration only affects a single robot and no program error is generated.

If set, the **SoftEStop** property is automatically cleared by the system if High Power is disabled and re-enabled.

Examples

```
Dim bState As Boolean
Controller.SoftEStop = True      ' Triggers a Soft E-Stop condition
bState = Controller.SoftEStop    ' bState will be set True since a
                                ' Soft E-Stop has been asserted
```

See Also

[Controller Class](#) | [Controller.PowerEnabled](#) | [Controller.PowerState](#) | [Robot.RapidDecel](#)

Controller.SystemMessage Method

Enters a message into the GPL system message log that is displayed on the web Operator Control Panel.

Controller.SystemMessage(*message*)

Prerequisites

None

Parameters

message

A required **String** expression containing the message to be entered into the message log.

Remarks

This method enters a line into the system message log with other system messages and error message entries. The system message log is kept sorted in time order. This log is displayed by the Operator Control Panel in the System Messages box.

Examples

```
Controller.SystemMessage("Cycle time: " & CStr(now-saved))  
Controller.SystemMessage("Operation complete")
```

See Also

[Controller Class](#) | [Controller.ErrorLog](#) | [Controller.ShowDialog](#) | [Controller.ShowDialogMCP](#)

Controller.Tick Property

Double value that specifies the execution period for the Trajectory Generator in seconds.

...Controller.Tick

Prerequisites

None

Parameters

None

Remarks

The Trajectory Generator is the task that evaluates robot motion plans and generates the series of individual commands to move each joint of each robot along its designated path. To accomplish this task, the Trajectory Generator executes at a configurable repetition rate. The Tick property returns the period of the repetition rate in seconds. Typically this will be set to a value of 0.004 or 0.008 seconds.

Examples

```
Dim period As Double
period = Controller.Tick      ' Sets period equal to the Trajectory
                             ' Generator execution period, e.g. 0.004
                             ' seconds
```

See Also

[Controller Class](#) | [Controller.SleepTick](#) | [Controller.Timer](#)

Controller.Timer Property

Returns the current value of the controller's usec clock, in units of seconds, as a **Double**.

...Controller.Timer

Prerequisites

None

Parameters

None

Remarks

This method reads the current value of the controller's usec clock and returns the value in units of seconds. This clock value starts counting from January 1, 1988. Given the number of significant bits in a **Double**, the **Timer** value will not lose accuracy until approximately the year 2124.

Examples

```
Dim StartTime, ElapsedTime As Double
StartTime = Controller.Timer           ' Reads system clock
Controller.SleepTick(2/Controller.Tick) ' Sleep for about 2 seconds
ElapsedTime = Controller.Timer-StartTime ' Value will be approx 2
```

See Also

[Controller Class](#) | [Controller.SleepTick](#) | [Controller.Tick](#)

Controller.Unload Method

Unloads the files and data associated with a GPL project from memory.

Controller.Unload(*project_name*)

Prerequisites

No procedures in this project can be currently executing.

Parameters

project_name

A required string expression that contains the name of the project to be unloaded.

Remarks

This method unloads a project by removing all of its associated data from the controller's memory and removing all associated files from the GPL project memory area.

This method throws an exception if any procedure in this project is currently executing. No exceptions are thrown if the project is not currently loaded or does not exist.

Examples

```
Dim th As Thread
Controller.Load("/flash/projects/Test")
th = New Thread("Main", "Test", "Thread2")
th.Start()
th.Join(0)           ' Wait for thread to complete
Controller.Unload("Test")
```

See Also

[Controller Class](#) | [Controller.Load](#) | [Thread.Join](#)

Exception Handling

Exception Handling Summary

The following pages provide detail information on the exception handling instructions and the properties and methods of the **Exception Class**. The exception handling statements provide a structured means for a procedure to detect and respond to program execution exceptions that would otherwise cause the procedure to halt execution. When an exception occurs, information on the cause of the exception can be automatically saved in an **Exception Object** and execution can be branched to a block of code designed to service the exception.

Exception Objects have two basic forms: a general **Exception** and a robot **Exception**. Both forms store a numerical code that indicates the type of exception. In addition, the robot **Exception** includes the number of the robot and the axes that are associated with the exception. The general form of the **Exception** includes a **Qualifier** value that can provide additional information on the nature of the exception.

The table below briefly summarizes the exception handling statements that are described in greater detail in the following pages.

Statement	Description
Catch	Used within a Try...Catch...Finally...End Try series of statements to mark the start of the block of instructions executed when an exception occurs.
End Try	Marks the end of the exception handling structure.
Exit Try	Terminates the execution of a Try or Catch block of instructions.
Finally	Used within a Try...Catch...Finally...End Try series of statements to mark the start of the block of instructions that is always executed at the completion of the Try or Catch blocks.
Throw	Generates a program execution exception.
Try...Catch...Finally...	Exception handling structure that captures execution exceptions within a block of instructions and executes statements to field the exception if necessary.

The table below briefly summarizes the properties and methods of the **Exception Class** that are described in greater detail in the following pages.

Member	Type	Description
exception_obj.Axis	Property	Sets and gets a bit mask indicating the robot axes associated with a robot Exception .
exception_obj.Clone	Method	Method that returns a copy of the <i>exception_obj</i> .
exception_obj.ErrorCode	Property	Sets and gets the number of the error message.
exception_obj.Message	Method	Returns the full text string that is generated based upon the <i>exception_obj</i> properties.
exception_obj.Qualifier	Property	Sets and gets the error message qualifier for a general Exception .

<u>exception_obj.RobotError</u>	Property	Sets and gets the Boolean that indicates if an Exception is a robot or general type.
<u>exception_obj.RobotNum</u>	Property	Sets and gets the number of the robot associated with a robot Exception .

Catch Statement

Used within a **Try...Catch...Finally...End Try** series of statements to mark the start of the block of instructions executed when an exception occurs.

Catch *exception_object*

Prerequisites

Must always follow a **Try** statement block. Either a **Catch** or **Finally** statement or one of each must appear in a **Try** structure.

Parameters

exception_object

Required **Exception Object**. The *exception_object* must already have a data section allocated prior to the execution of this instruction, i.e. the **New** qualifier should have been previously used in a **Dim** statement to instantiate the **Object**.

Remarks

The **Catch** statement marks the start of the block of instructions that is executed if an exception occurs during the execution of the corresponding **Try** block of instructions. If the **Catch** block is triggered, the information on the execution exception is automatically stored into the *exception_object*.

If an exception occurs during the execution of the **Catch** block of statements, thread execution will be terminated unless the violating instructions are themselves contained within a **Try** structure or if a higher-level **Try** structure exists.

At the completion of the **Catch** block, the statements in the following **Finally** block are executed if they exist, otherwise execution continues at the first step following the associated **End Try**.

Please see the documentation on the **Try...Catch...Finally...End Try** Statements for further information on the use of this statement.

See Also

[Exception Handling](#) | [Try...Catch...Finally...End Try Statements](#)

End Try Statement

This statement marks the end of the exception handling structure.

End Try

Prerequisites

Must always follow a **Catch** or **Finally** statement block.

Remarks

Please see the documentation on the **Try...Catch...Finally...End Try** Statements for further information on the use of this statement.

See Also

[Exception Handling](#) | [Try...Catch...Finally...End Try Statements](#)

Exit Try Statement

This statement terminates the execution of either a **Try** or a **Catch** block of instructions.

Exit Try

Prerequisites

Can only be specified within a **Try** or **Catch** block of instructions. In particular, this instruction is illegal within a **Finally** block.

Remarks

If this statement is executed within a **Try** or a **Catch** block of instructions, statement execution immediately branches to the first statement in the **Finally** block or, if the **Finally** block is not defined, the first statement following the subsequent **End Try**.

Please see the documentation on the **Try...Catch...Finally...End Try** Statements for information on the general format of the exception handling structure.

See Also

[Exception Handling](#) | [Try...Catch...Finally...End Try Statements](#)

Finally Statement

Used within a **Try...Catch...Finally...End Try** series of statements to mark the start of the block of instructions that is always executed at the completion of the **Try** or **Catch** blocks.

Finally

Prerequisites

Must always follow a **Try** or **Catch** statement block. Either a **Catch** or **Finally** statement or one of each must appear in a **Try** structure.

Remarks

The **Finally** statement marks the start of the block of instructions that is always executed after the successful execution of a **Try** series of statements or at the completion of the **Catch** series of statements. This allows a program to specify a series of statements that are guaranteed to be executed before execution continues following the **End Try** statement.

Please see the documentation on the **Try...Catch...Finally...End Try** Statements for further information on the use of this statement.

See Also

[Exception Handling](#) | [Try...Catch...Finally...End Try Statements](#)

Throw Statement

Generates a program execution exception.

Throw *exception_object*

Prerequisites

None

Parameters

exception_object

Required **Exception Object**. The **Exception** can contain either a general or a robot formatted error.

Remarks

This statement can be included in any procedure and need not be contained within a **Try...Catch...Finally...End Try** structure. Whenever it is executed, a program exception is immediately signaled. If this statement is not executed within a **Try** block, execution of the thread is terminated and the error contained within the *exception_object* is reported to the operator.

The **Throw** statement is often used within a **Catch** block. If the **Exception** captured by the **Catch** is not to be processed by the **Catch** block, the **Exception** can be reissued by a **Throw** statement. This allows **Exceptions** that are not to be serviced by a **Catch** to be passed to a higher-level **Catch** or to halt thread execution.

To allow application programs to generate their own special **Exceptions**, two error codes exist that are never automatically generated by the controller:

(-786) *Project generated error*
 (-1038) *Project generated robot error*

These error codes can be emitted by the **Throw** instruction to alert the operator to special exception conditions not normally detected by GPL.

Examples

```
Dim excl As New Exception
Try
retry:
    Move.Loc(loc1, profile1)
    Move.WaitForEOM
Catch excl
    If (excl.ErrorCode = -153) Then ' Soft envelope error?
        profile1.Speed *= .9 ' Yes, reduce speed
        GoTo retry
    End If
```

```
        Throw excl          ' Emit unknown error  
    End Try
```

See Also

[Exception Handling](#)

Try..Catch..Finally..End Try Statements

Exception handling structure that captures execution exceptions within a block of instructions and, if necessary, executes statements to field the exception.

```
Try
  [try_statements]
[ Catch exception_object
  [catch_statements]]
[ Finally
  [finally_statements]]
End Try
```

Prerequisites

None

Parameters

try_statements

Optional statement or list of statements whose exceptions, if any, will be handled by another block of code rather than immediately resulting in the termination of thread execution.

exception_object

Exception Object, required if the **Catch** statement is defined. When an exception occurs during the execution of the *try_statements*, the exception description is automatically stored in the *exception_object* prior to the execution of the *catch_statements*. The *exception_object* must already have a data section allocated prior to the execution of the **Catch**, i.e. the **New** qualifier should have been previously used in a **Dim** statement to instantiate the **Object**.

catch_statements

Optional statement or list of statements that are executed if an exception occurs during the execution of the *try_statements*.

finally_statements

Optional statement or list of statements that are always executed at the successful completion of the *try_statements* or the completion of the *catch_statements*.

Remarks

If an exception of any type occurs when the *try_statements* are executed, rather than halting execution and reporting the error, the system automatically stores the exception information in the *exception_object* and branches execution to the start of the *catch_statements*. The *catch_statements* can test the *exception_object* to determine the nature of the exception and then perform whatever corrective action is necessary. If the *try_statements* complete execution without an error or when the *catch_statements* complete execution after an exception, the *finally_statements* are always executed to perform any required cleanup. At the completion of the *finally_statements*, regular instruction execution continues at the first statement following the **End Try**.

A **Try** structure must contain either a single **Catch** statement or a single **Finally** statement or one of each type of statement. If a **Catch** statement is specified, it must always include an *exception_object*.

Try structures can be nested within each other to an arbitrary depth. For example, a **Try** structure can be contained within the *catch_statements* of another, higher-level **Try** structure. Also, procedure calls can be contained within any of the statement blocks including the *try_statements*.

If an exception occurs within a procedure that is invoked within a **Try** structure with a **Catch**, the execution of the procedure is immediately terminated and execution will continue at the first instruction in the *catch_statements* in the calling procedure. This feature allows a single **Try Catch** to be placed at a very high-level and capture any exceptions in any lower level routines. This case is illustrated in Example #1 below.

Alternately, if the called procedure generates an exception within a **Try** structure with a **Catch**, the *catch_statements* within the called routine will service the exception. However, if an exception occurs in a called procedure within a **Try** without a **Catch** but with a **Finally**, the *finally_statements* in the called routine will be executed first, then execution of the called procedure will be terminated, after which execution will continue in the *catch_statements* of the calling procedure. This case is illustrated in Example #2 below.

There are special limitations on the use of **GoTo** instructions in connection with **Try** structures. A **GoTo** contained in the *catch_statements* can branch execution into the corresponding *try_statements*. Also, **GoTo**'s can be contained in the *try_statements*, *catch_statements*, and the *finally_statements* so long as the branch is to an instruction within the same block of statements. All other branching into and out of the **Try** statement blocks and the main code is not permitted, e.g. you cannot branch from outside of a **Try** structure into the *try_statements* or out of the *try_statements* into the *finally_statements*. These special limitations are illustrated in Example #3 below.

Lastly, an **Exit Try** statement is provided for prematurely terminating a series of *try_statements* or *catch_statements*. When this instruction is executed in either the *try_statements* or the *catch_statements*, execution branches and continues at the first statement in the *finally_statements*. **Exit Try** instructions are not permitted in the *finally_statements*.

Examples

Example #1

```
Public Sub MAIN
    Dim excl As New Exception
```

```
Try
    test()
    Console.WriteLine("Test completed") ' Never gets here
Catch ex1
    Console.WriteLine("Exception!") ' Is executed
End Try
End Sub

Public Sub test()
    Dim ii As Integer
    ii = 1 / 0 ' Generates exception
    Console.WriteLine("Inside Test") ' Never gets here
End Sub
```

Example #2

```
Public Sub MAIN
    Dim ex1 As New Exception
    Try
        test()
        Console.WriteLine("Test completed") ' Never gets here
    Catch ex1
        Console.WriteLine("Exception!") ' Is executed
    End Try
End Sub

Public Sub test()
    Dim ii As Integer
    Try
        ii = 1 / 0 ' Generates exception
        Console.WriteLine("Inside Test") ' Never gets here
    Finally
        Console.WriteLine("Finally in Test") ' Is executed
    End Try
    Console.WriteLine("Test done") ' Never gets here
End Sub
```

Example #3

```
Dim ex1 As New Exception
Dim index As Integer
Robot.Attached = 1
Try
retry:
    Move.Loc(loc1, profile1)
    Move.WaitForEOM
Catch ex1
    Controller.SystemMessage(ex1.Message)
    Controller.ShowDialog("Ok,Cancel","Retry?",index)
    If index = 1 Then
        If Robot.Attached = 0 Then
            Controller.PowerEnabled = True
            Robot.Attached = 1
        End If
        GoTo retry ' LEGAL BRANCH
    End If
    GoTo bad_jump ' ILLEGAL BRANCH!!!
End Try
bad_jump:
```

See Also

[Exception Handling](#) | [Exit Try Statement](#) | [Throw Statement](#)

exception_object.Axis Property

Sets and gets a bit mask indicating the robot axes associated with a robot **Exception**.

```
exception_object.Axis = <new_bitmask_value>
-or-
...exception_object.Axis
```

Prerequisites

Only valid for robot **Exceptions**.

Parameters

None

Remarks

For robot **Exceptions**, the **Axis** property specifies the robot axes or motors that are associated with the error condition. This value is a bit mask where the least significant bit (&H1) represents the first axis or motor. Up to 12 bits can be set and multiple bits can be set at the same time. For example, when the error code is -1012 (Joint out-of-range), the **Axis** property bits indicate the which axes have violated their software ranges of motion.

When a **New Exception** is created, it defaults to a general **Exception** not a robot. When an **Exception** is set to a robot type, the **Axis** bits are initially all set to 0.

Examples

```
Dim excl As New Exception      ' Create new general exception
excl.RobotError = True        ' Indicate its a robot error
excl.ErrorCode = -1012        ' *Joint out-of-range*
excl.Axis = &HA               ' Specify axes 2 and 4
Console.WriteLine(excl.Message) ' *Joint out-of-range* Robot 1: 2 4
```

See Also

[Exception Handling](#) | [exception_object.RobotError](#) | [exception_object.RobotNum](#)

exception_object.Clone Method

Method that returns a copy of the *exception_object*.

...exception_object.Clone

Prerequisites

None

Parameters

None

Remarks

For objects, if a program contains a simple assignment statement:

object_1 = *object_2*

the result is that *object_1* points to the same data as *object_2*. Any subsequent change of a property in either *object_1* or *object_2* affects the data associated with both objects.

If you wish to make an independent copy of an object, the **Clone** method is the standard means for performing this operation:

object_1 = *object_2.Clone*

Examples

```
Dim exc1 As New Exception ' Create new exception with data
Dim exc2 As Exception     ' Create new exception with no data
exc1.ErrorCode = -1002    ' *Invalid axis* error code
exc1.RobotError = True
exc2 = exc1.Clone         ' Makes a copy of exc1 data
exc2.Axis = &HC           ' Does not affect exc1 data
Console.WriteLine(exc1.Message) ' *Invalid axis* Robot 1
Console.WriteLine(exc2.Message) ' *Invalid axis* Robot 1: 3 4
```

See Also

[Exception Handling](#)

exception_object.ErrorCode Property

Sets and gets the number of the error message.

```
exception_object.ErrorCode = <new_value>
-or-
...exception_object.ErrorCode
```

Prerequisites

None

Parameters

None

Remarks

The **ErrorCode** property of an **Exception** is the primary value that indicates the type of exception that is represented by the *exception_object*. This value can range from 4095 to -4095 and each utilized value has a text string associated with it for display purposes. In most cases, the **ErrorCode** is further qualified by additional information such as a robot number, axis number or other information.

To facilitate the interpretation of the **ErrorCodes**, positive values indicate success or warning conditions and negative numbers indicate an error of some type. A value of 0 is the general success code.

For a full listing of the defined **ErrorCode** values, please see the "System Error Codes" section of the *Precise Documentation Library*.

When a **New Exception** is created, it defaults to a general **Exception** with an **ErrorCode** value of 0 (success).

Examples

```
Dim excl As New Exception      ' Create new general exception
excl.ErrorCode = -786          ' *Project generated error*
excl.Qualifier = 8             ' Specify the qualifier
Console.WriteLine(excl.Message) ' *Project generated error*: 8
```

See Also

[Exception Handling](#)

exception_object.Message Method

Returns the full text string that is generated based upon the *exception_obj* properties.

```
...exception_object.Message
```

Prerequisites

None

Parameters

None

Remarks

Given any *exception_object*, this method interprets the **ErrorCode** and any defined refinement information such as the **RobotNum**, **Axis**, or **Qualifier** properties as appropriate and returns the equivalent text string that is normally output to indicate this exception.

Examples

```
Dim excl As New Exception      ' Create new general exception
excl.RobotError = True         ' Indicate its a robot error
excl.ErrorCode = -1012         ' *Joint out-of-range*
excl.Axis = &HA                ' Specify axes 2 and 4
Console.WriteLine(excl.Message) ' *Joint out-of-range* Robot 1: 2 4
```

See Also

[Exception Handling](#)

exception_object.Qualifier Property

Sets and gets the error message qualifier for a general **Exception**.

```
exception_object.Qualifier = <new_value>
-or-
...exception_object.Qualifier
```

Prerequisites

Only valid for general **Exceptions**.

Parameters

None

Remarks

For general **Exceptions**, the **Qualifier** property specifies an additional number that can be used to further refine the meaning of an error condition. This value is stored as a 16-bit unsigned number and can therefore range from 0 to 65535. For example, when the error code is -786 (Project generated error), the **Qualifier** property can be used by the GPL Project to convey which of several different special error conditions was detected.

When a **New Exception** is created, it defaults to a general **Exception** with a **Qualifier** property of 0. When an **Exception** is changed from a robot to a general type, the **Qualifier** value is reset to 0.

Examples

```
Dim excl As New Exception      ' Create new general exception
excl.ErrorCode = -786          ' *Project generated error*
excl.Qualifier = 8             ' Specify the qualifier
Console.WriteLine(excl.Message) ' *Project generated error*: 8
```

See Also

[Exception Handling](#) | [exception_object.RobotError](#)

exception_object.RobotError Property

Sets and gets the **Boolean** that indicates if an **Exception** is a robot or general type.

```
exception_object.RobotError = <boolean_value>
-or-
...exception_object.RobotError
```

Prerequisites

None

Parameters

None

Remarks

Setting the **RobotError** property of an *exception_object* to **True** indicates that it is a robot **Exception** and therefore has a **RobotNum** and an **Axis** property. Otherwise, setting **RobotError** to **False** indicates that the *exception_object* is a general **Exception** and has a **Qualifier** property.

Both robot and general **Exceptions** have the same effect in terms of halting thread execution and disabling robot power. The only difference between the two types of **Exceptions** is which additional properties exist to further refine the interpretation of the error code.

When a **New Exception** is created, it defaults to a general **Exception**. To switch between robot and general **Exception** types, the **RobotError** property should be set as needed.

Examples

```
Dim excl As New Exception      ' Create new general exception
excl.RobotError = True         ' Indicate its a robot error
excl.ErrorCode = -1006         ' *Robot already attached*
excl.RobotNum = 3              ' Specify the robot
Console.WriteLine(excl.Message) ' *Robot already attached* Robot 3
```

See Also

[Exception Handling](#)

exception_object.RobotNum Property

Sets and gets the number of the robot associated with a robot **Exception**.

```
exception_object.RobotNum = <new_value>
-or-
...exception_object.RobotNum
```

Prerequisites

Only valid for robot **Exceptions**.

Parameters

None

Remarks

For robot **Exceptions**, the **RobotNum** property specifies the number of the robot associated with the error condition. This value can range from 0 to 16. A value of 0 indicates that it is a conveyor belt and values from 1 to 16 specify regular robot numbers. For example, when the error code is -1006 (Robot already attached), the **RobotNum** property indicates which robot was being accessed when this error was generated.

When a **New Exception** is created, it defaults to a general **Exception** not a robot. When an **Exception** is set to a robot type, the **RobotNum** value is initially set to 1.

Examples

```
Dim excl As New Exception      ' Create new general exception
excl.RobotError = True         ' Indicate its a robot error
excl.ErrorCode = -1006         ' *Robot already attached*
excl.RobotNum = 3              ' Specify the robot
Console.WriteLine(excl.Message) ' *Robot already attached* Robot 3
```

See Also

[Exception Handling](#) | [exception_object.RobotError](#) | [exception_object.Axis](#)

File and Serial I/O Classes

File and Serial I/O Classes Summary

The following pages provide detailed information on the properties and methods for the various classes that implement both file and serial port input and output communications.

The **File Class** is designed specifically for managing disk files and disk file directories. The **StreamReader** and **StreamWriter Classes** apply to both file and serial communications.

The tables below briefly summarize the properties and methods for each Class, which are described in greater detail in the following sections.

File Class Member	Type	Description
File.CreateDirectory	Shared Method	Creates a file directory and the path to the directory.
File.DeleteDirectory	Shared Method	Deletes a single, empty file directory.
File.DeleteFile	Shared Method	Deletes a single file.
File.GetDirectories	Shared Method	Returns an array of strings containing the names of directories in a directory.
File.GetFiles	Shared Method	Returns an array of strings containing the names of files in a directory.

StreamReader Member	Type	Description
New StreamReader	Constructor Method	Opens a file or serial port device for reading.
streamreader_obj.Close	Method	Closes the file or device associated with a StreamReader Object .
streamreader_obj.Peek	Method	Returns the next byte from an input stream without removing it from the stream.
streamreader_obj.Read	Method	Returns the next byte from an input stream and removes it from the stream.
streamreader_obj.ReadLine	Method	Reads a line from the input stream terminated by LF, CR, or CR-LF.

StreamWriter Member	Type	Description
New StreamWriter	Constructor Method	Opens a file or serial port device for writing.
streamwriter_obj.AutoFlush	Property	Sets or gets the property that controls whether or not output is buffered.
streamwriter_obj.Close	Method	Closes the file or device associated with a

		StreamWriter Object.
<u>streamwriter_obj.Flush</u>	Method	Immediately writes any buffered data for a StreamWriter Object .
<u>streamwriter_obj.NewLine</u>	Property	Sets or gets the property that controls how lines are terminated by the WriteLine method.
<u>streamwriter_obj.Write</u>	Method	Writes a number or a String to an output device or file.
<u>streamwriter_obj.WriteLine</u>	Method	Writes a number or a String to an output device or file, followed by the NewLine line terminator.

File.CreateDirectory Method

Creates a file directory and the path to the directory.

File.CreateDirectory (*path*)

Prerequisites

Directories can only be created on the devices "/ROMDISK" and "/flash".

Parameters

path

A **String** that contains the path for the directory to create, beginning with the device name and ending with the new directory name.

Remarks

This method creates a directory in the location specified by the *path* parameter. If any intermediate directories in the path are undefined, they are automatically created.

An error occurs if the final directory already exists.

If any error occurs, this method throws an **Exception**.

Examples

```
File.CreateDirectory( "/ROMDISK/temp/new_directory" ) ' Create "new_directory"
                                                    ' Also creates "temp" if
needed
```

See Also

[File and Serial I/O](#) | [File.DeleteDirectory](#)

File.DeleteDirectory Method

Deletes a single, empty file directory.

File.DeleteDirectory (*path*)

Prerequisites

The directory must be empty.

Parameters

path

A **String** that contains the path for the directory to delete, beginning with the device name and ending with the new directory name.

Remarks

This method deletes a single directory in the location specified by the *path* parameter, provided that the directory is empty. If any files or sub-directories exist within the directory, an error occurs.

An error also occurs if the final directory does not exist.

If any error occurs, this method throws an **Exception**.

Examples

```
File.DeleteDirectory( "/ROMDISK/temp/new_directory" ) ' Delete "new_directory"
' if empty
```

See Also

[File and Serial I/O](#) | [File.CreateDirectory](#) | [File.DeleteFile](#)

File.DeleteFile Method

Deletes a single file.

File.DeleteFile (*path*)

Prerequisites

The file cannot be open for read or write.

Parameters

path

A **String** that contains the path to the file to delete, beginning with the device name and ending with the file name.

Remarks

This method deletes a single file in the location specified by the *path* parameter.

An error occurs if the file does not exist.

If any error occurs, this method throws an **Exception**.

Examples

```
File.DeleteFile("/ROMDISK/myfile.txt") ' Delete "myfile.txt"
```

See Also

[File and Serial I/O](#) | [File.DeleteDirectory](#)

File.GetDirectories Method

Reads a directory, gets the names of all sub-directories, and returns them in an array of **Strings**.

```
<string_array> = File.GetDirectories ( path )
```

Prerequisites

Directories can only be read on the devices "/ROMDISK" and "/flash".

Parameters

path

A required **String** expression that contains the path to the directory that is to be read. The *path* may not specify wild-card file name matching.

Remarks

This method permits a GPL program to retrieve the names of sub-directories within a directory. If the specified directory path does not exist, this method throws an exception.

One sub-directory name is returned per array element. The length of the returned **String** array indicates how many sub-directories were discovered. The sub-directory names are relative to the specified *path*.

If sub-directories are being actively created or deleted when this method is invoked, some existing sub-directories may be missed or a blank **String** element may be returned.

Examples

```
Dim files() As String
Dim ii As Integer
files = File.GetDirectories(path)
Console.WriteLine(CStr(files.Length) & " directories seen")
For ii = 1 To files.Length
    Console.WriteLine("File " & CStr(ii) & ": " & files(ii-1))
Next ii
```

See Also

[File and Serial I/O](#) | [File.GetFiles](#)

File.GetFiles Method

Reads a directory, gets the names of all non-directory files, and returns them in an array of **Strings**.

```
<string_array> = File.GetFiles ( path )
```

Prerequisites

Directories can only be read on the devices "/ROMDISK" and "/flash".

Parameters

path

A required **String** expression that contains the path to the directory that is to be read. The *path* may not specify wild-card file name matching.

Remarks

This method permits a GPL program to retrieve the names of files within a directory. If the specified directory path does not exist, this method throws an exception.

One file name is returned per array element. The length of the returned **String** array indicates how many files were detected. The file names are relative to the specified *path*.

If files are being actively created or deleted when this method is invoked, some existing files may be missed or a blank **String** element may be returned.

Examples

```
Dim files() As String
Dim ii As Integer
files = File.GetFiles(path)
Console.WriteLine(CStr(files.Length) & " files seen")
For ii = 1 To files.Length
    Console.WriteLine("File " & CStr(ii) & ": " & files(ii-1))
Next ii
```

See Also

[File and Serial I/O](#) | [File.GetDirectories](#)

New StreamReader Constructor

Constructor for creating a **StreamReader Object**. Also opens a file or device for reading.

New StreamReader (*path*)

Prerequisites

None

Parameters

path

A **String** that contains the path for the file or device to open. Local serial ports are devices named "/dev/com1", "/dev/com2", etc. Remote serial ports are named "/dev/comrxy" where "x" is the number of the remote device and "y" is the number of the serial port on the remote device. Temporary files may be placed on device "/ROMDISK" and permanent files may be placed on "/flash".

Remarks

This method opens a file or device and associates it with a new **StreamReader Object**.

If any error occurs, this constructor throws an **Exception**.

Examples

```
Dim com1 As New StreamReader("/dev/com1")           ' Open serial port #1
Dim tfile As New StreamReader("/ROMDISK/test.tmp") ' Open temporary file
Dim pfile As New StreamReader("/flash/save.txt")    ' Open permanent file
```

See Also

[File and Serial I/O](#) | [New StreamWriter](#)

streamreader_object.Close Method

Closes the file or device associated with a **StreamReader Object**.

```
streamreader_object.Close
```

Prerequisites

None

Parameters

None

Remarks

This method closes the file or device that is associated with a **StreamReader Object**. If any I/O error occurs, it throws an **Exception**. No error occurs if the file or device is not currently open.

Examples

```
streamreader_object.Close()
```

See Also

[File and Serial I/O](#) | [New StreamReader](#)

streamreader_object.Peek Method

Returns the next byte from an input stream without removing it from the stream.

```
...streamreader_object.Peek()
```

Prerequisites

The input stream must have been opened using a **New** to create the *streamreader_object*.

Parameters

None

Remarks

This method returns the next byte from the input stream as an **Integer**, but it does not remove the byte from the stream. The next input method call will still return this byte.

If any I/O error occurs or an end-of-file is encountered, this method returns -1.

For serial devices, this method does not block, but immediately returns -1 if no bytes are available to read.

If no device or file is open, this method throws an **Exception**.

Examples

```
Dim com1 As New StreamReader( "/dev/com1" )
Dim c As Integer
c = com1.Peek()
```

See Also

[File and Serial I/O](#) | [streamreader_object.Read](#)

streamreader_object.Read Method

Returns the next byte from an input stream and removes it from the stream.

```
...streamreader_object.Read()
```

Prerequisites

The input stream must have been opened using a **New** to create the *streamreader_object*.

Parameters

None

Remarks

This method returns the next byte from the input stream as an integer. The byte is removed from the stream so that subsequent calls do not return it.

If any I/O error occurs or an end-of-file is encountered, this method returns -1.

For serial devices, this method blocks if no bytes are available to read.

Be careful when using this method to read data from a serial port since it blocks until a byte is available. If for some reason the byte is lost due to an error, this method will continue blocking and hang your procedure.

If no device or file is open, this method throws an **Exception**.

Examples

```
Dim com1 As New StreamReader("/dev/com1")
Dim c As Integer
c = com1.Read()
```

See Also

[File and Serial I/O](#) | [streamreader_object.Peek](#) | [streamreader_object.ReadLine](#)

streamreader_object.ReadLine Method

Reads a line from the input stream terminated by LF, CR, or CR-LF.

```
...streamreader_object.ReadLine()
```

Prerequisites

The input stream must have been opened using a **New** to create the *streamreader_object*.

Parameters

None

Remarks

This method returns a **String** containing the next bytes in the input stream up to the next LF character (decimal value 10, **GPL_LF**) or CR character (decimal 13, **GPL_CR**). It blocks until the data followed by these line terminators is received or the end-of-file is seen.

Any LF, CR, or CR-LF pair is removed from the end of the string.

Note that the **StreamWriter.NewLine** property does not have any effect on how **ReadLine** interprets the end of line.

Be careful when using this method to read data from a serial port since it blocks until a line terminator is seen. If for some reason the line terminator is lost or corrupted due to an error, this method will continue blocking and hang your procedure.

If some other I/O error occurs, this method throws an **Exception**.

Examples

```
Dim file As New StreamReader("/flash/data.txt")  
Dim line As String  
line = file.ReadLine()
```

See Also

[File and Serial I/O](#) | [streamreader_object.Read](#)

New StreamWriter Constructor

Constructor for creating a **StreamWriter Object**. Also opens a file or device for writing.

```
New StreamWriter ( path )  
-or-  
New StreamWriter ( path, append )
```

Prerequisites

None

Parameters

path

A **String** that contains the path for the file or device to open. Serial ports are devices named `"/dev/com1"`, `"/dev/com2"`, etc. Remote serial ports are named `"/dev/comrxy"` where "x" is the number of the remote device and "y" is the number of the serial port on the remote device. Temporary files may be placed on device `"/ROMDISK"` and permanent files may be placed on `"/flash"`.

append

A **Boolean** value that determines whether or not new data should be appended to the end of an existing file. If *append* is **False**, a new file is always created, overwriting any existing file with the same name.

Remarks

This method opens a file or device and associates it with a new **StreamWriter Object**.

By default, **AutoFlush** is enabled for serial ports but not for files.

If any error occurs, this method throws an **Exception**.

Examples

```
Dim com1 As New StreamWriter("/dev/com1")           ' Open serial port #1  
Dim tfile As New StreamWriter("/ROMDISK/test.tmp") ' Open temporary file  
Dim pfile As New StreamWriter("/flash/save.txt")    ' Open permanent file
```

See Also

[File and Serial I/O](#) | [New StreamReader](#) | [streamwriter_object.AutoFlush](#)

streamwriter_object.AutoFlush Property

Sets or gets the **AutoFlush** property that controls whether or not output is buffered.

```
streamwriter_object.AutoFlush = <boolean_value>
-or-
...streamwriter_object.AutoFlush
```

Prerequisites

None

Parameters

None

Remarks

Setting this property to **True** causes output requests to immediately write data to the file or device. Setting it to **False** buffers the output and lets the system decide when to write it. Buffered output is always immediately written when a **Flush** or **Close** method is executed.

Setting **AutoFlush** to **True** for files may significantly slow down any write operations.

By default, **AutoFlush** is set to **True** for serial ports and **False** for files.

Examples

```
Dim pfile As New StreamWriter("/flash/save.txt") ' Open permanent file
pfile.AutoFlush = True
```

See Also

[File and Serial I/O](#) | [streamwriter_object.Flush](#)

streamwriter_object.Close Method

Closes the file or device associated with a **StreamWriter Object**.

```
streamwriter_object.Close
```

Prerequisites

None

Parameters

None

Remarks

This method closes the file or device that is associated with a **StreamWriter Object**. Any pending buffered output is written before the close completes.

If buffered output is being written, this method blocks until the output is complete.

If any I/O error occurs, this method throws an **Exception**. No error occurs if the file or device is not currently open.

Examples

```
streamwriter_object.Close()
```

See Also

[File and Serial I/O](#) | [New StreamWriter](#)

streamwriter_object.Flush Method

Immediately writes any buffered data for a **StreamWriter Object**.

streamwriter_object.Flush

Prerequisites

The output stream must have been opened using a **New** to create the *streamwriter_object*.

Parameters

None

Remarks

This method immediately writes any buffered data to the output device or file. When output is performed, this method blocks until it is complete.

Calling the **Flush** method is redundant if the **AutoFlush** property is set to True.

Explicit flush operations are more efficient than setting **AutoFlush** to True if you are performing a number of small write requests. If **AutoFlush** is True, each small write request causes output to occur. If **AutoFlush** is **False**, the small write requests can be buffered and the entire buffer is written by a single **Flush**.

A **Flush** equivalent is always performed by the **Close** method.

If any I/O error occurs, this method throws an **Exception**.

Examples

```
Dim com As New StreamWriter("/dev/com1")
com.AutoFlush = False      ' Disable automatic flush
com.Write("Write")
com.Write(" a short ")
com.WriteLine("message")
com.Flush
```

See Also

[File and Serial I/O](#) | [streamwriter_object.AutoFlush](#)

streamwriter_object.NewLine Property

Sets or gets the **NewLine** property that controls how lines are terminated by the **WriteLine** method.

```
streamwriter_object.NewLine = <newline_string>
-or-
...streamwriter_object.NewLine
```

Prerequisites

None

Parameters

None

Remarks

This property is a string of 0, 1 or 2 bytes that is appended to the end of any output performed by the *streamwriter_object*.**WriteLine** method.

By default the **NewLine** value is a 2-byte string containing an ASCII CR character (decimal 13, **GPL_CR**) followed by an LF character (decimal value 10, **GPL_LF**).

Typical settings for this property are CR, LF, or CR-LF. If set to an empty string, no terminator is added to the end of lines.

Examples

```
Dim pfile As New StreamWriter("/dev/com1")    ' Open serial port 1
pfile.NewLine = Chr(GPL_LF)                  ' Set terminator to LF (10)

...

pfile.NewLine = Chr(GPL_CR)                   ' Set terminator to CR (13)
```

See Also

[File and Serial I/O](#) | [streamwriter_object.WriteLine](#)

streamwriter_object.Write Method

Writes a number or a **String** to an output device or file.

```
streamwriter_object.Write( number )  
-or-  
streamwriter_object.Write( string_value )
```

Prerequisites

The output stream must have been opened using a **New** to create the *streamwriter_object*.

Parameters

number

A numeric value that is converted to a **String** and written.

string_value

A **String** expression this is written. Each byte of the **String** may be an arbitrary 8-bit value.

Remarks

This method writes **String** data to an output device or file. If a number is passed as the argument, it is first converted to an ASCII **String** value and then output.

Buffering of data is determined by the setting of the **AutoFlush** property. When output is actually performed, this method blocks until it is complete.

If any I/O error occurs, this method throws an **Exception**.

Examples

```
Dim tfile As New StreamWriter( "/ROMDISK/test.tmp" )  
tfile.Write("Test ")      ' Writes "Test "  
tfile.Write(3.14)         ' Writes "3.14" on the same line as "Test "
```

See Also

[File and Serial I/O](#) | [streamwriter_object.WriteLine](#)

streamwriter_object.WriteLine Method

Writes a number or a **String** to an output device or file, followed by the **NewLine** line terminator.

```
streamwriter_object.WriteLine( number )
-or-
streamwriter_object.WriteLine( string_value )
```

Prerequisites

The output stream must have been opened using a **New** to create the *streamwriter_object*.

Parameters

number

A numeric value that is converted to a **String** and written.

string_value

A **String** expression this is written. Each byte of the **String** may be an arbitrary 8-bit value.

Remarks

This method is the same as the **Write** method with the addition that it appends the value of the **NewLine** property to any output requests.

This method writes **String** data to an output device or file. If a number is passed as the argument, it is first converted to an ASCII **String** value and then output.

Buffering of data is determined by the setting of the **AutoFlush** property. When output is actually performed, this method blocks until it is complete.

If any I/O error occurs, this method throws an **Exception**.

Examples

```
Dim tfile As New StreamWriter( "/ROMDISK/test.tmp" )
tfile.WriteLine( "Test" )           ' Writes "Test"
tfile.WriteLine( 3.14 )             ' Writes "3.14" on the line following "Test"
```

See Also

[File and Serial I/O](#) | [streamwriter_object.NewLine](#) | [streamwriter_object.Write](#)

Functions

Function Summary

The following sections present detailed information on the standard functions that are supported by GPL. These functions are not grouped into a specific Class and are provided in this manner to be compatible with other Basic Language systems.

As is standard in GPL, conversions between different arithmetic types, e.g. **Boolean**, **Integer**, **Single**, **Double**, are automatically performed as required. So, it is not necessary to have different variations on these functions to deal with the different possible mixes of input parameter data types. Also, these functions generally produce results that are formatted as **Double**'s. These results will automatically be converted to smaller data types as necessary, e.g. **Double** -> **Integer**, and will not generate an error so long as numeric overflow does not occur.

The table below briefly summarizes the system functions that are described in greater detail in the following sections.

Function	Description
CBool (<i>expression</i>)	Converts any numeric type or String to Boolean
CByte (<i>expression</i>)	Converts any numeric type or String to Byte .
CDBl (<i>expression</i>)	Converts any numeric type or String to Double .
CInt (<i>expression</i>)	Converts any numeric type or String to Integer .
CShort (<i>expression</i>)	Converts any numeric type or String to Short .
CSng (<i>expression</i>)	Converts any numeric type or String to Single .
CStr (<i>expression</i>)	Converts any numeric type to String .
Fix (<i>number</i>)	Truncates towards zero any numeric type returning only the integer portion of the number.
Hex (<i>expression</i>)	Converts an Integer value to String in Hexadecimal format.
Int (<i>number</i>)	Truncates towards negative infinity any numeric type returning only the integer portion of the number.
Rnd (<i>seed</i>)	Returns a pseudo random number.

CBool Function

Converts any numeric type or **String** to a **Boolean** value.

...CBool (*expression*)

Prerequisites

None

Parameters

expression

A required numeric or string expression. The numeric expression can yield any type of result, i.e. **Boolean**, **Byte**, **Double**, **Integer**, **Short** or **Single**.

Remarks

The conversion operators are a group of functions that convert an expression that evaluates to any numeric or string type into a specified data type. The conversion tests that the converted value falls within the proper range of values for the returned data type. If the converted value is out of range, an error is generated.

As opposed to the **Int** and **Fix** functions, the conversion functions convert real numbers to integers by rounding rather than truncation.

The following table summarizes all of the conversion functions.

Function	Returned Data Type	Range of Valid Expression Values
CBool	Boolean	Any 0 or non-zero value
CByte	Byte	0 to 255
CDBl	Double	-1.79769313486231E+308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486231E+308 for positive values.
CInt	Integer	-2,147,483,648 to 2,147,483,647
CShort	Short	-32768 to 32767
CSng	Single	-3.402823E+38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E+38 for positive values.
CStr	String	Any valid Double value
Hex	String	Any valid Integer value

Examples

```
Dim s_val As Single
s_val = CInt(3.14159) ' Sets s_val equal to 3
s_val = CByte(300)    ' WILL GENERATE AN ERROR
```

See Also

[Functions](#) | [Fix Function](#) | [Int Function](#)

CByte Function

Converts any numeric type or **String** to a **Byte** value.

...CByte (*expression*)

Prerequisites

None

Parameters

expression

A required numeric or string expression. The numeric expression can yield any type of result, i.e. **Boolean**, **Byte**, **Double**, **Integer**, **Short** or **Single**.

Remarks

The conversion operators are a group of functions that convert an expression that evaluates to any numeric or string type into a specified data type. The conversion tests that the converted value falls within the proper range of values for the returned data type. If the converted value is out of range, an error is generated.

As opposed to the **Int** and **Fix** functions, the conversion functions convert real numbers to integers by rounding rather than truncation.

The following table summarizes all of the conversion functions.

Function	Returned Data Type	Range of Valid Expression Values
<u>CBool</u>	Boolean	Any 0 or non-zero value
<u>CByte</u>	Byte	0 to 255
<u>CDBl</u>	Double	-1.79769313486231E+308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486231E+308 for positive values.
<u>CInt</u>	Integer	-2,147,483,648 to 2,147,483,647
<u>CShort</u>	Short	-32768 to 32767
<u>CSng</u>	Single	-3.402823E+38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E+38 for positive values.
<u>CStr</u>	String	Any valid Double value
<u>Hex</u>	String	Any valid Integer value

Examples

```
Dim s_val As Single
s_val = CInt(3.14159) ' Sets s_val equal to 3
s_val = CByte(300)    ' WILL GENERATE AN ERROR
```

See Also

[Functions](#) | [Fix Function](#) | [Int Function](#)

CDBl Function

Converts any numeric type or **String** to a **Double** value.

...CDBl (*expression*)

Prerequisites

None

Parameters

expression

A required numeric or string expression. The numeric expression can yield any type of result, i.e. **Boolean**, **Byte**, **Double**, **Integer**, **Short** or **Single**.

Remarks

The conversion operators are a group of functions that convert an expression that evaluates to any numeric or string type into a specified data type. The conversion tests that the converted value falls within the proper range of values for the returned data type. If the converted value is out of range, an error is generated.

As opposed to the **Int** and **Fix** functions, the conversion functions convert real numbers to integers by rounding rather than truncation.

The following table summarizes all of the conversion functions.

Function	Returned Data Type	Range of Valid Expression Values
<u>CBool</u>	Boolean	Any 0 or non-zero value
<u>CByte</u>	Byte	0 to 255
<u>CDBl</u>	Double	-1.79769313486231E+308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486231E+308 for positive values.
<u>CInt</u>	Integer	-2,147,483,648 to 2,147,483,647
<u>CShort</u>	Short	-32768 to 32767
<u>CSng</u>	Single	-3.402823E+38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E+38 for positive values.
<u>CStr</u>	String	Any valid Double value
<u>Hex</u>	String	Any valid Integer value

Examples

```
Dim s_val As Single
s_val = CInt(3.14159) ' Sets s_val equal to 3
s_val = CByte(300)    ' WILL GENERATE AN ERROR
```

See Also

[Functions](#) | [Fix Function](#) | [Int Function](#)

CInt Function

Converts any numeric type or **String** to an **Integer** value.

...CInt (*expression*)

Prerequisites

None

Parameters

expression

A required numeric or string expression. The numeric expression can yield any type of result, i.e. **Boolean**, **Byte**, **Double**, **Integer**, **Short** or **Single**.

Remarks

The conversion operators are a group of functions that convert an expression that evaluates to any numeric or string type into a specified data type. The conversion tests that the converted value falls within the proper range of values for the returned data type. If the converted value is out of range, an error is generated.

As opposed to the **Int** and **Fix** functions, the conversion functions convert real numbers to integers by rounding rather than truncation.

The following table summarizes all of the conversion functions.

Function	Returned Data Type	Range of Valid Expression Values
CBool	Boolean	Any 0 or non-zero value
CByte	Byte	0 to 255
CDBl	Double	-1.79769313486231E+308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486231E+308 for positive values.
CInt	Integer	-2,147,483,648 to 2,147,483,647
CShort	Short	-32768 to 32767
CSng	Single	-3.402823E+38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E+38 for positive values.
CStr	String	Any valid Double value
Hex	String	Any valid Integer value

Examples

```
Dim s_val As Single
s_val = CInt(3.14159) ' Sets s_val equal to 3
s_val = CByte(300)   ' WILL GENERATE AN ERROR
```

See Also

[Functions](#) | [Fix Function](#) | [Int Function](#)

CShort Function

Converts any numeric type or **String** to a **Short** value.

...CShort (*expression*)

Prerequisites

None

Parameters

expression

A required numeric or string expression. The numeric expression can yield any type of result, i.e. **Boolean**, **Byte**, **Double**, **Integer**, **Short** or **Single**.

Remarks

The conversion operators are a group of functions that convert an expression that evaluates to any numeric or string type into a specified data type. The conversion tests that the converted value falls within the proper range of values for the returned data type. If the converted value is out of range, an error is generated.

As opposed to the **Int** and **Fix** functions, the conversion functions convert real numbers to integers by rounding rather than truncation.

The following table summarizes all of the conversion functions.

Function	Returned Data Type	Range of Valid Expression Values
<u>CBool</u>	Boolean	Any 0 or non-zero value
<u>CByte</u>	Byte	0 to 255
<u>CDbl</u>	Double	-1.79769313486231E+308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486231E+308 for positive values.
<u>CInt</u>	Integer	-2,147,483,648 to 2,147,483,647
<u>CShort</u>	Short	-32768 to 32767
<u>CSng</u>	Single	-3.402823E+38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E+38 for positive values.
<u>CStr</u>	String	Any valid Double value
<u>Hex</u>	String	Any valid Integer value

Examples

```
Dim s_val As Single
s_val = CInt(3.14159) ' Sets s_val equal to 3
s_val = CByte(300)    ' WILL GENERATE AN ERROR
```

See Also

[Functions](#) | [Fix Function](#) | [Int Function](#)

CSng Function

Converts any numeric type or **String** to a **Single** value.

...CSng (*expression*)

Prerequisites

None

Parameters

expression

A required numeric or string expression. The numeric expression can yield any type of result, i.e. **Boolean**, **Byte**, **Double**, **Integer**, **Short** or **Single**.

Remarks

The conversion operators are a group of functions that convert an expression that evaluates to any numeric or string type into a specified data type. The conversion tests that the converted value falls within the proper range of values for the returned data type. If the converted value is out of range, an error is generated.

As opposed to the **Int** and **Fix** functions, the conversion functions convert real numbers to integers by rounding rather than truncation.

The following table summarizes all of the conversion functions.

Function	Returned Data Type	Range of Valid Expression Values
CBool	Boolean	Any 0 or non-zero value
CByte	Byte	0 to 255
CDBl	Double	-1.79769313486231E+308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486231E+308 for positive values.
CInt	Integer	-2,147,483,648 to 2,147,483,647
CShort	Short	-32768 to 32767
CSng	Single	-3.402823E+38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E+38 for positive values.
CStr	String	Any valid Double value
Hex	String	Any valid Integer value

Examples

```
Dim s_val As Single
s_val = CInt(3.14159) ' Sets s_val equal to 3
s_val = CByte(300)    ' WILL GENERATE AN ERROR
```

See Also

[Functions](#) | [Fix Function](#) | [Int Function](#)

CStr Function

Converts any numeric type to a **String** value.

...CStr (*expression*)

Prerequisites

None

Parameters

expression

A required numeric expression. The numeric expression can yield any type of result, i.e. **Boolean**, **Byte**, **Double**, **Integer**, **Short** or **Single**.

Remarks

The conversion operators are a group of functions that convert an expression that evaluates to any numeric or string type into a specified data type. The conversion tests that the converted value falls within the proper range of values for the returned data type. If the converted value is out of range, an error is generated.

As opposed to the **Int** and **Fix** functions, the conversion functions convert real numbers to integers by rounding rather than truncation.

The following table summarizes all of the conversion functions.

Function	Returned Data Type	Range of Valid Expression Values
<u>CBool</u>	Boolean	Any 0 or non-zero value
<u>CByte</u>	Byte	0 to 255
<u>CDBl</u>	Double	-1.79769313486231E+308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486231E+308 for positive values.
<u>Clnt</u>	Integer	-2,147,483,648 to 2,147,483,647
<u>CShort</u>	Short	-32768 to 32767
<u>CSng</u>	Single	-3.402823E+38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E+38 for positive values.
<u>CStr</u>	String	Any valid Double value
<u>Hex</u>	String	Any valid Integer value

Examples

```
Dim stg As String
stg = CStr(3.14159)      ' Sets stg equal to "3.14159"
```

See Also

[Functions](#) | [Fix Function](#) | [Format Function](#) | [Int Function](#)

Fix Function

Returns the integer portion of any number by truncating towards zero.

```
...Fix ( number )
```

Prerequisites

None

Parameters

number

A required numeric expression. The numeric expression can yield any type of result, i.e. **Boolean**, **Byte**, **Double**, **Integer**, **Short** or **Single**.

Remarks

The **Int** and **Fix** functions return the integer portion of any number by truncating the fraction part of the value. For positive numbers, these two functions are identical. However, for negative numbers, the **Int** function returns the first negative number less than or equal to the input expression value. Alternately, the **Fix** function returns the first negative number that is greater than or equal to the input expression value. For example:

```
Dim s_val As Single
s_val = Int(-1.2)      ' Sets s_val equal to -2
s_val = Fix(-1.2)     ' Sets s_val equal to -1
s_val = Int(-1.9)     ' Sets s_val equal to -2
s_val = Fix(-1.9)     ' Sets s_val equal to -1
```

Unlike the conversion routines (e.g. **CInt**, **CShort**), these functions truncate their values rather than round them. For example:

```
Dim s_val As Single
s_val = Int(1.2)      ' Sets s_val equal to 1
s_val = CInt(1.2)     ' Sets s_val equal to 1
s_val = Int(1.9)     ' Sets s_val equal to 1
s_val = CInt(1.9)     ' Sets s_val equal to 2
```

In addition, the conversion routines test the converted values to ensure that the returned value is within the range of a specific data type. The **Int** and **Fix** routines simply eliminate the fraction portion of any number and perform no range testing.

Examples

```
Dim s_val As Single
s_val = Int(3.14159)  ' Sets s_val equal to 3
s_val = Int(3.99999)  ' Sets s_val equal to 3
```

See Also

[Functions](#) | [Int Function](#)

Hex Function

Converts an **Integer** value to a **String** value in Hexadecimal format.

...Hex (*expression*)

Prerequisites

None

Parameters

expression

A required numeric expression. The numeric expression can yield any type of result, i.e. **Boolean**, **Byte**, **Double**, **Integer**, **Short** or **Single**, however, the value is converted to **Integer** prior to conversion to a String value.

Remarks

The conversion operators are a group of functions that convert an expression that evaluates to any numeric or string type into a specified data type. The conversion tests that the converted value falls within the proper range of values for the returned data type. If the converted value is out of range, and error is generated.

As opposed to the **Int** and **Fix** functions, the conversion functions convert real numbers to integers by rounding rather than truncation.

The following table summarizes all of the conversion functions.

Function	Returned Data Type	Range of Valid Expression Values
<u>CBool</u>	Boolean	Any 0 or non-zero value
<u>CByte</u>	Byte	0 to 255
<u>CDBl</u>	Double	-1.79769313486231E+308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486231E+308 for positive values.
<u>Clnt</u>	Integer	-2,147,483,648 to 2,147,483,647
<u>CShort</u>	Short	-32768 to 32767
<u>CSng</u>	Single	-3.402823E+38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E+38 for positive values.
<u>CStr</u>	String	Any valid Double value
<u>Hex</u>	String	Any valid Integer value

Examples

```
Dim stg As String
Dim ii As Integer
ii = CInt("&H1234") ' Sets ii equal to 4660
stg = Hex(ii)       ' Sets stg equal to "1234"
```

See Also

[Functions](#) | [Fix Function](#) | [Format Function](#) | [Int Function](#)

Int Function

Returns the integer portion of any number by truncating towards negative infinity.

...Int (*number*)

Prerequisites

None

Parameters

number

A required numeric expression. The numeric expression can yield any type of result, i.e. **Boolean**, **Byte**, **Double**, **Integer**, **Short** or **Single**.

Remarks

The **Int** and **Fix** functions return the integer portion of any number by truncating the fraction part of the value. For positive numbers, these two functions are identical. However, for negative numbers, the **Int** function returns the first negative number less than or equal to the input expression value. Alternately, the **Fix** function returns the first negative number that is greater than or equal to the input expression value. For example:

```
Dim s_val As Single
s_val = Int(-1.2)      ' Sets s_val equal to -2
s_val = Fix(-1.2)     ' Sets s_val equal to -1
s_val = Int(-1.9)     ' Sets s_val equal to -2
s_val = Fix(-1.9)     ' Sets s_val equal to -1
```

Unlike the conversion routines (e.g. **CInt**, **CShort**), these functions truncate their values rather than round them. For example:

```
Dim s_val As Single
s_val = Int(1.2)      ' Sets s_val equal to 1
s_val = CInt(1.2)     ' Sets s_val equal to 1
s_val = Int(1.9)     ' Sets s_val equal to 1
s_val = CInt(1.9)     ' Sets s_val equal to 2
```

In addition, the conversion routines test the converted values to ensure that the returned value is within the range of a specific data type. The **Int** and **Fix** routines simply eliminate the fraction portion of any number and perform no range testing.

Examples

```
Dim s_val As Single  
s_val = Int(3.14159) ' Sets s_val equal to 3  
s_val = Int(3.99999) ' Sets s_val equal to 3
```

See Also

[Functions](#) | [Fix Function](#)

Rnd Function

Returns a pseudo random number.

```
...Rnd ( seed )
```

Prerequisites

None

Parameters

seed

An optional expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns a pseudo random number whose value is greater than or equal to 0 and less than 1.0.

The returned value is only pseudo random because the returned numbers are part of an extremely long sequence of values that only repeat after 2^{32} numbers are generated. Each time that the controller is restarted, the starting point or *seed* in the sequence is determined by the system clock calendar. So, the sequence of values produced by this function appears quite random for normal testing purposes.

If it is desired to force the sequence of numbers to restart at a fixed value, thereby allowing a test to be exactly repeated, the optional *seed* parameter can be used as follows:

<i>seed</i> value	Effect on function
<0	The specified <i>seed</i> value is taken as the starting point for the pseudo random sequence and the sequence will be continued from this value. The number returned by this execution of the Rnd will always be the same.
=0	The last value returned by the Rnd function will be returned again.
>0	The next number in the pseudo random sequence will be returned.
Not specified	Same as specifying a <i>seed</i> value >0.

Examples

```
Dim r_val As Single
r_val = Rnd()           ' Sets r_val to some random value
r_val = Rnd(-1)         ' Forces seed to -1, will return same number
                        ' each time.
r_val = Rnd()           ' Returns next value after seed
r_val = Rnd(0)          ' Returns same value as last line above
```

See Also

[Functions](#)

Location Class

Location Class Summary

The following pages provide detailed information on the properties and methods of the **Location Class**. This class and its **Location Object** instances provide the fundamental means for representing robot and part positions and orientations within GPL. **Location Objects** and **Profile Objects** (which define motion performance parameters) are the standard arguments required by most **Move** methods for defining how to drive the robot along a path to a destination specified by a **Location**.

Each **Location Object** contains data that defines: a **Type** indicator; a position and orientation; clearance information that is used to safely approach the **Location**; and robot configuration specific information that pertains to the target robot.

There are two **Type**'s of **Location Objects**: Angles and Cartesian. The Angles **Locations** store robot positions as an array of axes positions. When we refer to the "position" or "total position" of an Angles **Location**, we are referring to the array of axes positions. The more general **Type** is called a Cartesian **Location**. Cartesian **Locations** contain a Cartesian position and orientation that is displayed as an **X, Y, Z** displacement and a set of three Euler Angles: **Yaw**, **Pitch**, and **Roll**. In addition to this position and orientation, each Cartesian **Location** contains an optional pointer to a reference frame object. The **X, Y, Z, Yaw, Pitch**, and **Roll** values define the **Location's** "position with respect to the reference frame" (**PosWrtRef**). When we refer to the "position" or "total position" of a Cartesian **Location**, we are discussing the combined effect of the "position with respect to the reference frame" and any specified reference frames.

Since flexible automation must alter a robot's actions in order to accommodate to variations in a material handling, assembly or other type of operation, extensive methods are provided for mathematically manipulating the position and orientation of **Locations**. The table below briefly summarized the properties and methods that are described in greater detail in the following sections.

Member	Type	Description
<u>location_obj.Angle</u>	Property	Sets and gets a single axis position for an Angles Location .
<u>location_obj.Angles</u>	Method	Changes all of the axes positions values in an Angles Location .
<u>location_obj.Clone</u>	Method	Returns a copy of the <i>location_obj</i> .
<u>location_obj.Config</u>	Property	Sets and gets the bit flags that specify special robot specific location attributes.
<u>Location.Distance</u>	Method	Returns the distance between the XYZ positions of two Cartesian Locations .
<u>location_obj.Here</u>	Method	Modifies the "total position" of the <i>location_obj</i> to be equal to the current location of a robot.
<u>location_obj.Here3</u>	Method	Defines the "total position" of <i>location_obj</i> based upon the XYZ coordinates of three specified locations.
<u>location_obj.Inverse</u>	Method	Returns the inverse of the "total position" of the Cartesian <i>location_obj</i> .

<u>location_obj.Kinesol</u>	Method	Returns a Cartesian Location equivalent to an Angles Location for a specific kinematic model or vise versa.
<u>location_obj.Mul</u>	Method	Returns the result of combining the “total position” of <i>location_obj</i> with the “total position” of another Cartesian Location .
<u>location_obj.Normalize</u>	Method	Corrects the value of the PosWrtRef of a Cartesian Location for any mathematical inconsistencies in the value.
<u>location_obj.Pitch</u>	Property	Sets and gets the Pitch angle of the PosWrtRef of a Cartesian Location .
<u>location_obj.Pos</u>	Property	Sets and gets the “total position” of the <i>location_obj</i> .
<u>location_obj.PosWrtRef</u>	Property	Sets and gets the PosWrtRef of a Cartesian Location .
<u>location_obj.RefFrame</u>	Property	Sets and gets a pointer to the reference frame object that the <i>location_object</i> is defined relative to.
<u>location_obj.Roll</u>	Property	Sets and gets the Roll angle of the PosWrtRef of a Cartesian Location .
<u>location_obj.Type</u>	Property	Sets and gets the Type specification.
<u>location_obj.X</u>	Property	Sets and gets the X position value of the PosWrtRef of a Cartesian Location .
<u>location_obj.XYZ</u>	Method	Changes the X, Y, Z, Yaw, Pitch, and Roll values of the PosWrtRef of a Cartesian Location .
<u>location_obj.XYZInc</u>	Method	Increments the X, Y, and Z values of the PosWrtRef of a Cartesian Location .
<u>Location.XYZValue</u>	Method	Returns a Cartesian Location with a "total position" equal to specified X, Y, Z, Yaw, Pitch, and Roll coordinates.
<u>location_obj.Y</u>	Property	Sets and gets the Y position value of the PosWrtRef of a Cartesian Location .
<u>location_obj.Yaw</u>	Property	Sets and gets the Yaw angle of the PosWrtRef of a Cartesian Location .
<u>location_obj.Z</u>	Property	Sets and gets the Z position value of the PosWrtRef of a Cartesian Location .
<u>location_obj.ZClearance</u>	Property	Sets and gets the distance along the Z-axis that defines the safe approach position to the Location .
<u>location_obj.ZWorld</u>	Property	Sets and gets the flag that indicates if the approach distance is measured along the Tool or World Z coordinate axis.

location_object.Angle Property

Sets and gets the position of a single robot axis, in units of millimeters or degrees, to and from an Angles **Location Object**.

```
location_object.Angle(axis) = <new_numeric_value>
-or-
...location_object.Angle(axis)
```

Prerequisites

The *location_object* must be an Angles **Location Object**.

Parameters

axis

A required numeric expression that specifies the number of the axis to be accessed. This value can range from 1 for the first axis up to a maximum value of 12.

Remarks

An Angles **Location Object** stores the position of the robot as a set of axes position values. For generality, a **Location Object** always contains 12 axes positions although the trajectory generation task will only make use of one value for each axis configured for the robot.

The **Angle** property allows a program to access and manipulate individual axis position values. To set all of the axes positions at one time, the **Angles** method should be utilized.

If the *location_object* is not of the **Angles** type, accessing the **Angle** property will generate an error.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
Dim ang As Double
loc1.Angles(-21.5, 23.2, 10) ' Set loc1 to Angles type and define position
ang = loc1.Angle(2)          ' ang will be set to 23.2
loc1.Angle(2) *= 2            ' Position of axis 2 will be 46.4
```

See Also

[Location Class](#) | [location_object.Angles](#)

location_object.Angles Method

Changes all of the axes positions values stored in an Angles **Location Object**.

```
location_object.Angles(axis_1, ..., axis_12)
```

Prerequisites

None

Parameters

axis_1, ..., axis_12

Up to 12 optional numeric expressions that specifies the new position value for each of the robot axes. If an expression is not specified, the corresponding axis position will default to a value of 0. Each value is in units of millimeters or degrees as appropriate for the axes.

Remarks

An Angles **Location Object** stores the position of the robot as a set of axes position values. For generality, a **Location Object** always contains 12 axes positions although the trajectory generation task will only make use of one value for each axis configured for the robot.

The **Angles** method sets the values of all of the axes positions in the *location_object*. Any unspecified positions are set to 0. To read or write individual axis positions, the **Angle** property should be utilized.

As a convenience, independent of the initial **Type** of the *location_object*, at the conclusion of this operation, the *location_objectType* will be set to indicate it is an Angles **Location Object**.

Examples

```
Dim loc1 As New Location      ' Create new Location with default values
Dim ang As Double
loc1.Angles(-21.5, 23.2, 10)  ' Set loc1 to Angles type and define
ang = loc1.Angle(2)          ' ang will be set to 23.2
loc1.Angle(2) *= 2           ' Position of axis 2 will be 46.4
```

See Also

[Location Class](#) | [location_object.Angle](#)

location_object.Clone Method

Method that returns a copy of the *location_object*.

```
...location_object.Clone
```

Prerequisites

None

Parameters

None

Remarks

For objects, if a program contains a simple assignment statement:

object_1 = *object_2*

the result is that *object_1* points to the same data as *object_2*. Any subsequent change of a property in either *object_1* or *object_2* affects the data associated with both objects.

If you wish to make an independent copy of an object, the **Clone** method is the standard means for performing this operation:

object_1 = *object_2*.**Clone**

Examples

```
Dim loc1 As New Location      ' Create new location set to default values
Dim loc2 As Location          ' Create new location with no data allocated
loc1.X = 10.2                  ' Set X position in loc1.
loc2 = loc1.Clone              ' Makes a copy of loc1 data
loc2.Y = -27.1                 ' Doesn't affect loc1 data
```

See Also

[Location Class](#)

location_object.Config Property

Sets and gets an **Integer** bit mask that specifies how the Cartesian position of a **Location Object** is to be converted to a set of axes position values.

```
location_object.Config = <new_Integer_value>
-or-
...location_object.Config
```

Prerequisites

None

Parameters

None

Remarks

For some robots, there are multiple sets of axes positions that will position the robot's tool or gripper at the same position and orientation. For simple robots, this can occur if a wrist axis can rotate more than 360 degrees. For more complex geometries, the alternate sets of axes positions might correspond to what is termed "right" and "left" shoulder configurations.

GPL's optional kinematic modules include methods for automatically selecting among different sets of positions in some instances. For example, if the final wrist axis of a robot can rotate a total of 720 degrees, GPL can automatically select which revolution of this axis should be selected as the destination for a motion to a Cartesian end point. Normally, GPL will rotate the wrist to the closest position that satisfies the Cartesian specification. However, if this would violate a wrist joint limit stop, GPL will rotate the wrist in the opposite direction.

In other cases, GPL cannot automatically select the best set of joint angles to be used. In these cases, GPL will generally try to maintain the robot in the same configuration unless instructed otherwise. For example, if a position can be reached in both a "right" and a "left" shouldered configurations, GPL will maintain the same shoulder configuration unless explicitly directed to change. This is done to prevent large, unexpected motions that can occur when switching the shoulder configuration.

To both indicate the current geometric configuration and to specify a change in configuration, the **Config** property provides a series of bit flags that instruct GPL how it is to convert Cartesian **Locations** into joint angles. When a Cartesian destination is specified with one or more of these bits set, the next motion to this **Location** will try to put the robot into the specified configuration. If bits are not set, GPL assumes that the robot should be instructed to stay in its current configuration.

While some configuration changes can be implemented during either a Cartesian or joint-interpolated motion, other changes can only be performed during joint-interpolated motions. For example, you cannot change from a right to a left shouldered configuration

and simultaneously move the tool tip along a Cartesian straight-line path. If a configuration bit is specified which is not compatible with the specified motion type, the configuration bit is ignored and no error is generated.

The bits currently defined for the **Config** property are described in the following table. As a programming convenience, these bits also have GPL constants defined.

Config Bit Mask	GPL Constant	Legal During Cartesian Motion	Description
&H01	GPL_Righty	No	Change robot to a right shouldered configuration.
&H02	GPL_Lefty	No	Change robot to a left shouldered configuration.
&H04	GPL_Above	No	Change robot to have the elbow above the wrist.
&H08	GPL_Below	No	Change robot to have the elbow below the wrist.
&H10	GPL_Flip	No	Change robot to have the wrist pitched up.
&H20	GPL_NoFlip	No	Change robot to have the wrist pitched down.
&H1000	GPL_Single	Yes	Restrict the wrist axis to be within +/- 180 degrees rather than use its full range of motion.

Since the robot configuration options are a function of the robot's geometry, please see the documentation in the Kinematics Library for which bits apply to your robot.

Examples

```
Dim loc1 As New Location ' Create new Cartesian Location
loc1.Config = GPL_Righty+GPL_Single
' Set mask word to force robot to right
' shouldered and limit wrist rotation
```

See Also

[Location Class](#) | [Robot.Dest](#) | [Robot.Where](#)

Location.Distance Method

Returns the distance between the XYZ positions of two Cartesian **Location Objects**.

```
...Location.Distance(location_object1, location_object2)
```

Prerequisites

location_object1 and *location_object2* must both be Cartesian **Location Objects**.

Parameters

location_object1

A required Cartesian **Location Object** or a method or property that returns a Cartesian **Location Object** value.

location_object2

A required Cartesian **Location Object** or a method or property that returns a Cartesian **Location Object** value.

Remarks

This method computes the distance between the positions of two Cartesian **Location Objects** and returns the result as a **Double**.

Examples

```
Dim a As New Location      ' Create Locations and allocate
Dim b As New Location
Dim dist As Double
a.XYZ(10,23,-17,0,0,90)    ' Define A, orientation doesn't matter
b.XYZ(21,8,12)              ' Define B
dist = Location.Distance(a,b) ' dist set equal to 34.45287
```

See Also

[Location Class](#)

location_object.Here Method

Sets the “total position” of a **Location Object** equal to the current position and orientation of the **Selected** robot.

location_object.Here

Prerequisites

A robot must be currently **Selected**, but need not be **Attached**.

Parameters

None

Remarks

The **Here** method provides a very convenient means for defining or updating the “total position” of a *location_object* by moving the robot to the desired position and then executing this method to record the position and orientation.

This method works properly for both Cartesian and Angles **Locations**. If the *location_object* is an Angles type, the values of the *location_object*’s axes positions are set equal to the current axes positions of the **Selected** robot. For Cartesian types, the “total position” is set equal to the current Cartesian position and orientation of the **Selected** robot. If the *location_object* does not have an associated reference frame, the **PosWrtRef** is set equal to the current Cartesian location of the robot. If the *location_object* has a reference frame, the **PosWrtRef** is set such that the combination of the new **PosWrtRef** and the reference frame will be equal to the current location of the robot.

While the **Here** method is similar to assigning a *location_object* to the value of the **Robot.Where()** method, it is important to understand the differences. The statement:

```
location_object = Robot.Where() ' Works okay
```

assigns a new block of data to the *location_object*. While it does save the current robot location in the *location_object*, the values previously set for **ZClearance**, **ZWorld**, and **RefFrame** are effectively lost. On the other hand, the statement:

```
location_object.Here ' Even better
```

alters the **PosWrtRef** value in the *location_object* with less overhead while still preserving the values for **ZClearance**, **ZWorld**, and **RefFrame**. So, in most situations, the **Here** method produces the expected results and should be employed instead of an assignment statement with **Robot.Where()**.

Examples

```
Dim loc1 As New Location ' Create new Location set to default values
loc1.Here                 ' Sets "total position" of loc1 to present
                           ' location of Selected robot.
```

See Also

[Location Class](#) | [location_object.Here3](#) | [location_object.Inverse](#) | [location_object.Mul](#) | [Robot.Selected](#)
| [Robot.Where](#) | [Robot.WhereAngles](#)

location_object.Here3 Method

Defines the "total position" of a **Location Object** based upon the XYZ coordinates of three specified **Locations**.

```
location_object.Here3(location_0, location_x, location_y)
```

Prerequisites

location_0, *location_x* and *location_y* must be Cartesian **Location Objects**.

Parameters

location_0

A required Cartesian **Location Object** or a method or property that returns a Cartesian **Location Object** value.

location_x

A required Cartesian **Location Object** or a method or property that returns a Cartesian **Location Object** value.

location_y

A required Cartesian **Location Object** or a method or property that returns a Cartesian **Location Object** value.

Remarks

This method is utilized for setting the "total position" of *location_object* based upon the XYZ position coordinates of three **Locations**. This is convenient if you wish to define the orientation and position of a **Location** or reference frame by teaching three **Locations**.

The total position of the *location_object* is computed as follows:

-
- The XYZ coordinates of the *location_object* are set equal to the XYZ coordinates of the total position of *location_0*. That is, the XYZ coordinates of *location_0* define the 0,0,0 position of the coordinate system defined by the new value of *location_object*.
- The direction of the x-axis of *location_object* is defined to be parallel to the vector from the XYZ coordinate of *location_0* to the XYZ coordinate of *location_x*. That is, if the XYZ position of *location_0* is equivalent to the 0,0,0 position of the coordinate frame defined by the new value of *location_object*, then the XYZ position of *location_x* will be a point on the x-axis of the coordinate system defined by the new value of *location_object*.
- The XY plane of the new *location_object* value is defined by the XYZ coordinates of *location_0*, *location_x*, and *location_y*. Normally, *location_y* is defined such that its XYZ position will be a point on the y-axis of the coordinate system defined by the new value of *location_object*.

At the completion of this method, the **PosWrtRef** value of the *location_object* will be set such that the total position of *location_object* corresponds to the position and orientation defined by three points represented by the three **Location** arguments. Also, as a convenience, the **Type** of the *location_object* is always set to indicate it is a Cartesian **Location Object**.

Examples

```
Dim loc1 As New Location      ' Define position of this Location
Dim loc0 As New Location
Dim locx As New Location
Dim locy As New Location
loc0.XYZ(10,20,30)           ' Define 0,0,0
locx.XYZ(10,25,30)           ' Define point on X-axis
locy.XYZ(5,20,30)            ' Define point on Y-axis
loc1.Here3(loc0,locx,locy)    ' Will define loc1 to same as
                              ' loc1.XYZ(10,20,30,0,0,90)
```

See Also

[Location Class](#) | [location_object.Here](#) | [location_object.XYZ](#)

location_object.Inverse Method

Returns the inverse of the “total position” of the Cartesian *location_object*.

...location_object.Inverse

Prerequisites

The *location_object* must be a Cartesian **Location Object**.

Parameters

None

Remarks

This method evaluates the “total position” of the *location_object* and then inverts the value. As defined in the description of GPL, the “total position” is the combination of the *location_object*’s **PosWrtRef** with the “total position” of any reference frame(s) associated with the *location_object*.

As an example, if the “total position” of the *location_object* represents the position and orientation of part B with respect to part A, then the **Inverse** will give the position and orientation of A with respect to B. As another way to think about this operation, if the *location_object* defines how to get from A to B then the **Inverse** will define how to get from B to A.

Assuming that the *location_object* is a Cartesian type, the **Inverse** method returns a **Location Object** with the following properties:

Property	Returned Location Object value
Type	Cartesian Location
PosWrtRef	Inverse of the “total position” of the <i>location_object</i>
RefFrame	Null
All other properties	Same as <i>location_object</i>

Examples

```
Dim loc1 As New Location          ' Create new Location set to defaults
Dim loc2, loc3 As Location
Dim dy As Double
loc1.XYZ(11, -23, 45, 0, 180, 42) ' Define "position" of loc1
loc2 = loc1.Inverse
loc3 = loc2.Inverse               ' loc3 will have same "position" as loc1
dy = loc3.Y                       ' dy will be equal to -23
```

See Also

[Location Class](#) | [location_object.Pos](#) | [location_object.Mul](#) | [location_object.PosWrtRef](#)

location_object.KineSol Method

Returns a Cartesian **Location Object** equivalent to an Angles **Location Object** for a specific kinematic model or vice versa.

```
...location_object.KineSol
```

Prerequisites

A robot must be currently **Selected**, but need not be **Attached**.

Parameters

None

Remarks

This method converts a set of axes positions to an equivalent Cartesian position and orientation or converts a Cartesian position and orientation to an equivalent set of axes positions based upon the **Selected** robot's geometry (kinematics). These operations are typically called the "forward and reverse kinematic solutions" and require an optional kinematic module.

Specifically, if the *location_object* is an Angles type, the **KineSol** method returns a **Location Object** with the following properties:

Property	Returned Location Object value
Type	Cartesian Location
PosWrtRef	Equivalent to <i>location_object</i> Angles values
Config	Appropriate for <i>location_object</i> Angles values
RefFrame	Null
All other properties	Same as <i>location_object</i>

Alternatively, if the *location_object* is a Cartesian type, the **KineSol** method returns a **Location Object** with the following properties:

Property	Returned Location Object value
Type	Angles Location
Angles	Equivalent to <i>location_object</i> 's "total position"
Config	0
RefFrame	Null
All other properties	Same as <i>location_object</i>

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
Dim loc2, loc3 As Location
Dim axis2 As Double
loc1.Angles(12, 42, 17)      ' Assume these values legal values for robot
loc2 = loc1.KineSol          ' Set loc2 to equivalent Cartesian Location
loc3 = loc2.KineSol          ' Regenerate Angles Location
axis2 = loc3.Angle(2)        ' axis2 should be 42 as in loc1
```

See Also

[Location Class](#) | [location_object.Inverse](#) | [location_object.Mul](#) | [Robot.Selected](#)

location_object.Mul Method

Returns the combination of the position and orientation of a Cartesian *location_object* with another Cartesian **Location Object**.

```
...location_object.Mul(location_object2)
```

Prerequisites

location_object and *location_object2* must both be Cartesian **Location Objects**.

Parameters

location_object2

A required Cartesian **Location Object** or a method or property that returns a Cartesian **Location Object** value.

Remarks

This method combines the “total position” of *location_object* and the “total position” of *location_object2*. As described in the Introduction to GPL, the “total position” of a **Location Object** is the combination of the **Location Object’s PosWrtRef** with the “total position” of any reference frame(s) associated with the **Location Object**.

More specifically, the **Mul** method returns the result of evaluating the “total position” of *location_object2* with respect to the **PosWrtRef** value of the *location_object*. If defined, the reference frame pointer for the *location_object* is copied to the returned **Location** and is not included in the mathematic operation. This is done to preserve the explicit reference frame relationship of the *location_object*.

For example, let’s consider the simple case without rotations where the *location_object* has an X, Y, Z value of (10,25,-40) and *location_object2* has an X, Y, Z value of (0,5,0). If we now combined the values, *location_object2*’s incremental displacement of 5 mm along the Y-axis would be interpreted with respect to *location_object*’s prior translations and the combined result would be (10,30,-40). Now, we can see what happens if we change *location_object* so it includes a 90-degree rotation about the Z-axis (10,25,-40,0,0,90). In this case, when we combine the two values, *location_object2*’s Y-axis has been rotated to point along *location_object*’s negative X-axis. So, the resulting combination would be (5, 25,-40,0,0,90).

Assuming that *location_object* and *location_object2* are both Cartesian **Locations**, the **Mul** method returns a **Location Object** with the following properties:

Property	Returned Location Object value
Type	Cartesian Location
PosWrtRef	“total position” of the <i>location_object2</i> evaluated with respect to the PosWrtRef of the <i>location_object</i> . In terms of matrix

	<p>operations, this could be written as:</p> <pre> returned.PosWrtRef = [location_object.PosWrtRef] *[location_object2.RefFrame] *[location_object2.PosWrtRef] </pre>
RefFrame	Same as <i>location_object</i>
All other properties	Same as <i>location_object</i>

Examples

```

Dim a As New Location      ' Create new Location set to default values
Dim b As New Location
Dim c As Location
Dim dx, dy As Double
a.XYZ(10,25,-40,0,0,90)   ' Define A
b.XYZ(0,5,0)              ' Define B
c = a.Mul(b)
dx = c.X                  ' dx will be 5
dy = c.Y                  ' dy will be equal to 25

```

See Also

[Location Class](#) | [location_object.Inverse](#) | [location_object.Pos](#) | [location_object.PosWrtRef](#)

location_object.Normalize Method

Corrects the **PosWrtRef** value of a Cartesian **Location Object** for any mathematical inconsistencies in the value.

location_object.Normalize

Prerequisites

The *location_object* must be a Cartesian **Location Object**.

Parameters

None

Remarks

After many sequential mathematics operations (e.g. **Inverse**, **Mul**) have been performed on a Cartesian **Location Object**, it is possible for the homogeneous transformation that is used to internally store the **PosWrtRef** value to suffer from mathematical inconsistencies. For example, certain rows and columns of the 4x4 matrix are vectors that must have unit values and be orthogonal to other vectors in the matrix. Given that all of the elements of a transformation are stored as double precision floating-point numbers, this problem is not very likely to occur.

Nonetheless, as a convenience, the **Normalize** method can be executed on a Cartesian *location_object* and it will correct any mathematic errors that may have accumulated in the **PosWrtRef** value.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
loc1.XYZ(10,20,30,0,180,25)  ' Set PosWrtRef value of loc1
loc1.Normalize                ' Won't alter loc1 since it is already correct
```

See Also

[Location Class](#) | [location_object.Inverse](#) | [location_object.Mul](#)

location_object.Pitch Property

Sets and gets the Pitch angle, in units of degrees, for the **PosWrtRef** value of a Cartesian **Location Object**.

```
location_object.Pitch = <new_value>
-or-
...location_object.Pitch
```

Prerequisites

The *location_object* must be a Cartesian **Location Object**.

Parameters

None

Remarks

Internally, the **PosWrtRef** value of a Cartesian **Location Object** is stored as a sparse 4 by 4 matrix called a “homogeneous transformation”. This matrix represents the three positional degrees-of-freedom and the three rotational degrees-of-freedom needed to fully specify a robot or part position and orientation in Cartesian coordinates. This internal representation has several computational advantages. However, entering the values for the elements of a homogeneous transformation is not very convenient. To simplify data entry, transformation values are converted to X, Y, and Z position displacement components and three Euler angles. The three Euler angles consist of a rotation about the Z-axis, followed by a rotation about the new Y-axis, followed by a rotation about the new Z-axis. This set of displacements and angles is often referred to as X, Y, Z, Yaw, Pitch, and Roll.

The property described on this page allows read and write access to one of the six components used to specify the **PosWrtRef** value of the Cartesian *location_object*.

Accessing the X, Y, and Z properties is an efficient operation. However, accessing the Yaw, Pitch, and Roll properties requires some computational overhead. Therefore, if you wish to set multiple angles, it is more efficient to utilize the **XYZ** method.

When a “New” Cartesian **Location Object** is created, all six components are initially set to 0.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
Dim ang As Double
loc1.XYZ(10,20,30,0,180,25)  ' Set PosWrtRef value of loc1
ang = loc1.Roll              ' ang will be set to 25
loc1.Roll += 5                ' loc1's Roll angle will now be 30 deg.
```

See Also

[Location Class](#) | [location_object.X](#) | [location_object.Y](#) | [location_object.Z](#) | [location_object.Yaw](#) | [location_object.Roll](#) | [location_object.XYZ](#)

location_object.Pos Property

Sets and gets the “total position” of the *location_object*.

```
location_object.Pos = <specified_location_value>
-or-
...location_object.Pos
```

Prerequisites

None

Parameters

None

Remarks

The **Pos** operation accesses the “total position” of both Cartesian and Angles **Location Objects**. For Cartesian **Locations** without reference frames, the “total position” is equal to the **PosWrtRef** value stored as a Cartesian position and orientation in the *location_object*. For Cartesian **Locations** with reference frames, the “total position” is equal to the **PosWrtRef** value of the *location_object* evaluated with respect to the “total position” of its reference frames. For Angles **Locations**, the “total value” is the equal to the set of axes positions stored in the *location_object*.

The **Pos** set operation works properly on all varieties of **Locations**. However, the type of the *<specified_location_value>* must match the type of the *location_object*, i.e. they must both either be Cartesian or Angles.

For Cartesian **Locations**, the “total position” of the *location_object* is set equal to the “total position” of the *<specified_location_value>*. If the *location_object* does not have an associated reference frame, the **PosWrtRef** value is set equal to the “total position” of the *<specified_location_value>*. If the *location_object* has a reference frame, the **PosWrtRef** value of the *location_object* is set such that the combination of the new **PosWrtRef** value of the *location_object* and its reference frame will be equal to the “total position” of the *<specified_location_value>*. If the *location_object* is an Angles type, the value of the *location_object*’s axes positions are set equal to the axes positions of the *<specified_location_value>*.

While the **Pos** method is similar to assigning a *location_object* to the value of another **Location Object**, it is important to understand the differences. The statement:

```
location_object = location_object2
```

assigns a pointer to *location_object2*’s data to the *location_object*. Not only does this operation supercede any reference frame you may have assigned to *location_object*, it also supercedes any other data assigned, such as its **ZClearance** information.

Furthermore, if you subsequently make a change to the data of either *location_object* or *location_object2*, the data for both objects will be effected. Alternatively, you could use the following assignment statement:

```
location_object = location_object2.Clone
```

This statement makes a copy of *location_object2*'s value before assigning it to *location_object*. This statement does eliminate the potential problem of having two variables inadvertently referencing the same data. However, it does not address superceding the original reference frame specification and other data. Also, one additional downside of this operation is that creating a copy of an object's value does incur a certain amount of system overhead.

On the other hand, the statement:

```
location_object.Pos = location_object2
```

alters the **PosWrtRef** or **Angles** values of *location_object* with low overhead and preserves all of the other properties of the *location_object*.

If the goal of a statement is simply to update the existing "total position" or **PosWrtRef** value of a **Location** without regard to the reference frame, you should normally make use of either the **Pos** or **PosWrtRef** set properties.

Regarding the **Pos** get operation, this property returns a **Location Object** that contains only the "total position" of the *location_object* with no reference frame or other data. Please note that if the *location_object* is a Cartesian type with a reference frame, the position and orientation of the **PosWrtRef** value and the "total position" of the reference frame are combined and returned as the **PosWrtRef** value of the returned **Object**.

For all cases the value of the returned **Object** from the **Pos** get operation is as follows:

Property	Returned Location Object value
Type	Cartesian or Angles Location as appropriate
PosWrtRef or Angles	"total position" of the <i>location_object</i>
RefFrame	Always NULL
ZClearance	1.0e32 to indicate not initialized
All other properties	Always zeroed.

Examples

```
Dim loc1 As New Location      ' Create new Location set to defaults
Dim loc2 As New Location
loc1.ZClearance = 12
loc2.XYZ(10,20,30,0,180,23)  ' Define PosWrtRef value for loc2
loc1.Pos = loc2              ' Use same "total position" for loc1
```

See Also

[Location Class](#) | [location_object.Inverse](#) | [location_object.Mul](#) | [location_object.PosWrtRef](#)

location_object.PosWrtRef Property

Sets and gets the “position with respect to the reference frame” value of a Cartesian **Location Object** while ignoring the reference frame.

```
location_object.PosWrtRef = <specified_location_value>
-or-
...location_object.PosWrtRef
```

Prerequisites

The *location_object* must be a Cartesian **Location Object**.

Parameters

None

Remarks

This property accesses the “position with respect to the reference frame” of a Cartesian **Location Object**. Normally, the **PosWrtRef** value is evaluated in combination with the reference frame to compute the “total position” of a **Location**. However, this property accesses the “position with respect to the reference frame” data ignoring any specified reference frame data.

The **PosWrtRef** set operation allows a statement to assign a new value to the “position with respect to the reference frame” of the *location_object* without affecting or considering the value of any reference frame or any other data of the *location_object*. The new value is set equal to the “total position” of the *<specified_location_value>* on the right hand side of the equal sign.

The **PosWrtRef** get operation returns a Cartesian **Location Object** that contains only the “position with respect to the reference frame” of the *location_object* with no reference frame or other data. In particular, the value of the returned **Object** is as follows:

Property	Returned Location Object value
Type	Cartesian Location
PosWrtRef	PosWrtRef of the <i>location_object</i>
RefFrame	Always NULL
ZClearance	1.0e32 to indicate not initialized
All other properties	Always zeroed.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
Dim loc2 As New Location
loc1.ZClearance = 12
loc2.XYZ(10,20,30,0,180,23)  ' Define position for loc2
loc1.PosWrtRef = loc2.PosWrtRef ' Use same PosWrtRef for loc1
```

See Also

[Location Class](#) | [location_object.Inverse](#) | [location_object.Mul](#) | [location_object.Pos](#)

location_object.RefFrame Property

Sets and gets a pointer to the reference frame object that the *location_object* is defined relative to.

```
location_object.RefFrame = <reference_frame_object>
-or-
... location_object.RefFrame
```

Prerequisites

The *location_object* must be a Cartesian **Location**.

Parameters

None

Remarks

Sets or gets the pointer to a reference frame object that the *location_object*'s position and orientation is to be defined relative to. Whenever the *location_object*'s total position and orientation are computed, the position and orientation of the **RefFrame** are automatically taken into consideration.

When a new **Location Object** is defined, its pointer to a reference frame object is zeroed by default.

Examples

```
Dim refl As New RefFrame           ' Also allocates Loc
Dim loc1 As New Location
ref1.Loc.XYZ(100,90,-80,0,0,45)    ' Define base frame
loc1.RefFrame = refl               ' Define loc1 wrt refl
loc1.XYZ(10,0,0,0,180,0)           ' Define loc1 poswrtref
Console.WriteLine(loc1.Pos.X)      ' Displays 107.07
Console.WriteLine(loc1.Pos.Y)      ' Displays 97.07
Console.WriteLine(loc1.Pos.Z)      ' Displays -80
```

See Also

[Location Class](#) | [RefFrame Class](#)

location_object.Roll Property

Sets and gets the Roll angle, in units of degrees, for the **PosWrtRef** value of a Cartesian **Location Object**.

```
location_object.Roll = <new_value>
-or-
...location_object.Roll
```

Prerequisites

The *location_object* must be a Cartesian **Location Object**.

Parameters

None

Remarks

Internally, the **PosWrtRef** value of a Cartesian **Location Object** is stored as a sparse 4 by 4 matrix called a “homogeneous transformation”. This matrix represents the three positional degrees-of-freedom and the three rotational degrees-of-freedom needed to fully specify a robot or part position and orientation in Cartesian coordinates. This internal representation has several computational advantages. However, entering the values for the elements of a homogeneous transformation is not very convenient. To simplify data entry, transformation values are converted to X, Y, and Z position displacement components and three Euler angles. The three Euler angles consist of a rotation about the Z-axis, followed by a rotation about the new Y-axis, followed by a rotation about the new Z-axis. This set of displacements and angles is often referred to as X, Y, Z, Yaw, Pitch, and Roll.

The property described on this page allows read and write access to one of the six components used to specify the **PosWrtRef** value of the Cartesian *location_object*.

Accessing the X, Y, and Z properties is an efficient operation. However, accessing the Yaw, Pitch, and Roll properties requires some computational overhead. Therefore, if you wish to set multiple angles, it is more efficient to utilize the **XYZ** method.

When a “New” Cartesian **Location Object** is created, all six components are initially set to 0.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
Dim ang As Double
loc1.XYZ(10,20,30,0,180,25)   ' Set PosWrtRef value of loc1
ang = loc1.Roll               ' ang will be set to 25
loc1.Roll += 5                 ' loc1's Roll angle will now be 30 deg.
```

See Also

Location Class | *location object.X* | *location object.Y* | *location object.Z* | *location object.Yaw* |
location object.Pitch | *location object.XYZ*

location_object.Type Property

Sets and gets the **Integer Type** of a **Location Object**, which indicates if the **Location Object** holds Cartesian or Angles data.

```
location_object.Type = <new_Integer_value>
-or-
...location_object.Type
```

Prerequisites

None

Parameters

None

Remarks

The **Type** property indicates if the *location_object* contains Cartesian or Angles position and orientation data. The possible values for this property are as follows:

Type Value	Description
0	Location contains Cartesian position and orientation data.
1	Location contains a set of axes position values ("Angles").

Many of the other **Location Object** properties and methods will generate an error if you attempt to access values that are not meaningful for the current **Type** of the *location_object*.

As a convenience, some methods, e.g. **Angles** and **XYZ**, automatically set the **Type** of a **Location Object**.

When a "New" Cartesian **Location** is created, its **Type** is automatically set to Cartesian.

Examples

```
Dim loc1 As New Location      ' Create new Cartesian Location
Dim iType As Integer
iType = loc1.Type             ' iType will be set to 0
loc1.Angles(10.2,-3.2)        ' Will automatically set Type to 1
```

See Also

[Location Class](#)

location_object.X Property

Sets and gets the displacement along the X-axis, in units of millimeters, for the **PosWrtRef** value of a Cartesian **Location Object**.

```
location_object.X = <new_value>
-or-
...location_object.X
```

Prerequisites

The *location_object* must be a Cartesian **Location Object**.

Parameters

None

Remarks

Internally, the **PosWrtRef** value of a Cartesian **Location Object** is stored as a sparse 4 by 4 matrix called a “homogeneous transformation”. This matrix represents the three positional degrees-of-freedom and the three rotational degrees-of-freedom needed to fully specify a robot or part position and orientation in Cartesian coordinates. This internal representation has several computational advantages. However, entering the values for the elements of a homogeneous transformation is not very convenient. To simplify data entry, transformation values are converted to X, Y, and Z position displacement components and three Euler angles. The three Euler angles consist of a rotation about the Z-axis, followed by a rotation about the new Y-axis, followed by a rotation about the new Z-axis. This set of displacements and angles is often referred to as X, Y, Z, Yaw, Pitch, and Roll.

The property described on this page allows read and write access to one of the six components used to specify the **PosWrtRef** value of the Cartesian *location_object*.

Accessing the X, Y, and Z properties is an efficient operation. However, accessing the Yaw, Pitch, and Roll properties requires some computational overhead. Therefore, if you wish to set multiple angles, it is more efficient to utilize the **XYZ** method.

When a “New” Cartesian **Location Object** is created, all six components are initially set to 0.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
Dim dx As Double
loc1.XYZ(10,20,30,0,180,25)   ' Set PosWrtRef value of loc1
dx = loc1.X                   ' dx will be set to 10
loc1.X -= 2                    ' loc1's X value will now be 8
```

See Also

[Location Class](#) | [location object.Y](#) | [location object.Z](#) | [location object.Yaw](#) | [location object.Pitch](#) |
[location object.Roll](#) | [location object.XYZ](#)

location_object.XYZ Method

Changes all six components of the **PosWrtRef** value of a Cartesian **Location Object** to a specified set of values.

```
location_object.XYZ(x,y,z,yaw,pitch.roll)
```

Prerequisites

None

Parameters

x

An optional numeric expression that specifies the X-axis displacement. If this value is not specified, a default value of 0 is assumed.

y

An optional numeric expression that specifies the Y-axis displacement. If this value is not specified, a default value of 0 is assumed.

z

An optional numeric expression that specifies the Z-axis displacement. If this value is not specified, a default value of 0 is assumed.

yaw

An optional numeric expression that specifies the Yaw angle rotation. If this value is not specified, a default value of 0 is assumed.

pitch

An optional numeric expression that specifies the Pitch angle rotation. If this value is not specified, a default value of 0 is assumed.

roll

An optional numeric expression that specifies the Roll angle rotation. If this value is not specified, a default value of 0 is assumed.

Remarks

Internally, the **PosWrtRef** value of a Cartesian **Location Object** is stored as a sparse 4 by 4 matrix called a “homogeneous transformation”. This matrix represents the 3 positional degrees-of-freedom and the 3 rotational degrees-of-freedom needed to fully specify a robot or part position and orientation in Cartesian coordinates. This internal representation has several computational advantages. However, entering the values for the elements of a homogeneous transformation is not very convenient. To simplify data entry, transformation values are entered as X, Y, and Z position displacement components and three Euler angles. The three Euler angles consist of a rotation about the Z-axis, followed by a rotation about the new Y-axis, followed by a rotation about the new Z-axis. This set of displacements and angles is often referred to as X, Y, Z, Yaw, Pitch, and Roll.

The **XYZ** method sets all six Cartesian components of the *location_object*'s **PosWrtRef** value in a single operation. Any unspecified values are set to 0. This operation is much more efficient than using the **X**, **Y**, **Z**, **Yaw**, **Pitch**, and **Roll** properties to individually set the component values.

As a convenience, independent of the initial **Type** of the *location_object*, at the conclusion of this operation, the **Type** will be set to indicate it is a Cartesian **Location Object**.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
Dim dy As Double
loc1.XYZ(10,20,30,0,180,25)   ' Set PosWrtRef value of loc1
dy = loc1.Y                   ' dy will be set to 20
loc1.Y += 7                    ' loc1's Y value will now be 27
```

See Also

[Location Class](#) | [location_object.X](#) | [location_object.Y](#) | [location_object.Z](#) | [location_object.Yaw](#) | [location_object.Pitch](#) | [location_object.Roll](#) | [location_object.XYZInc](#) | [Location.XYZValue](#)

location_object.XYZInc Method

Increments the X/Y/Z components of the **PosWrtRef** value of a Cartesian **Location Object** by specified amounts.

```
location_object.XYZInc(x,y,z)
```

Prerequisites

The *location_object* must be a Cartesian **Location Object**.

Parameters

x

An optional numeric expression that specifies the amount by which the X value is incremented. If this value is not specified, a default value of 0 is assumed.

y

An optional numeric expression that specifies the amount by which the Y value is incremented. If this value is not specified, a default value of 0 is assumed.

z

An optional numeric expression that specifies the amount by which the Z value is incremented. If this value is not specified, a default value of 0 is assumed.

Remarks

This method increments the **X**, **Y**, and **Z** Cartesian displacement components of the *location_object*'s **PosWrtRef** value in a single operation. Any unspecified increments leave the corresponding displacement values unchanged.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
loc1.XYZ(10,20,30,0,180,25)  ' Set PosWrtRef value of loc1
loc1.XYZInc(-3,,2)           ' Changes X to 7 and Z to 32
```

See Also

[Location Class](#) | [location_object.X](#) | [location_object.Y](#) | [location_object.Z](#) | [location_object.Yaw](#) | [Location.XYZValue](#)

Location.XYZValue Method

Returns a Cartesian **Location** with a "total position" equal to specified X, Y, Z, Yaw, Pitch, and Roll coordinates.

```
...Location.XYZValue(x,y,z,yaw,pitch.roll)
```

Prerequisites

None

Parameters

x

An optional numeric expression that specifies the X-axis displacement. If this value is not specified, a default value of 0 is assumed.

y

An optional numeric expression that specifies the Y-axis displacement. If this value is not specified, a default value of 0 is assumed.

z

An optional numeric expression that specifies the Z-axis displacement. If this value is not specified, a default value of 0 is assumed.

yaw

An optional numeric expression that specifies the Yaw angle rotation. If this value is not specified, a default value of 0 is assumed.

pitch

An optional numeric expression that specifies the Pitch angle rotation. If this value is not specified, a default value of 0 is assumed.

roll

An optional numeric expression that specifies the Roll angle rotation. If this value is not specified, a default value of 0 is assumed.

Remarks

The **XYZValue** method computes and returns a Cartesian **Location Object** that has a "total position" value whose displacement and orientation is equivalent to that specified by the *x*, *y*, *z*, *yaw*, *pitch*, and *roll* arguments. This method is provided as a convenience for constructing **Location** expressions.

If you wish to set the **PosWrtRef** value of a Cartesian **Location Object** equal to a set of displacement and orientation values, it is more efficient to utilize the **XYZ** method instead of **XYZValue**.

The following table describes the data returned in the **Location Object**.

Property	Returned Location Object value
Type	Cartesian Location
PosWrtRef	Set equal to the displacement and orientation defined by x, y, z, <i>yaw</i> , <i>pitch</i> , and <i>roll</i> arguments.
RefFrame	Always Null
ZClearance	1.0e32 to indicate not initialized
All other properties	Always zeroed.

Examples

```
Dim loc1 As Location          ' Locations default to Cartesian
loc1.PosWrtRef = Location.XYZValue(10,20,30,0,180,25)
                             ' Equivalent to "loc1.XYZ(10,20,30,0,180,25)"
```

See Also

[Location Class](#) | [location_object.XYZ](#)

location_object.Y Property

Sets and gets the displacement along the Y-axis, in units of millimeters, for the **PosWrtRef** value of a Cartesian **Location Object**.

```
location_object.Y = <new_value>
-or-
...location_object.Y
```

Prerequisites

The *location_object* must be a Cartesian **Location Object**.

Parameters

None

Remarks

Internally, the **PosWrtRef** value of a Cartesian **Location Object** is stored as a sparse 4 by 4 matrix called a “homogeneous transformation”. This matrix represents the three positional degrees-of-freedom and the three rotational degrees-of-freedom needed to fully specify a robot or part position and orientation in Cartesian coordinates. This internal representation has several computational advantages. However, entering the values for the elements of a homogeneous transformation is not very convenient. To simplify data entry, transformation values are converted to X, Y, and Z position displacement components and three Euler angles. The three Euler angles consist of a rotation about the Z-axis, followed by a rotation about the new Y-axis, followed by a rotation about the new Z-axis. This set of displacements and angles is often referred to as X, Y, Z, Yaw, Pitch, and Roll.

The property described on this page allows read and write access to one of the six components used to specify the **PosWrtRef** value of the Cartesian *location_object*.

Accessing the X, Y, and Z properties is an efficient operation. However, accessing the Yaw, Pitch, and Roll properties requires some computational overhead. Therefore, if you wish to set multiple angles, it is more efficient to utilize the **XYZ** method.

When a “New” Cartesian **Location Object** is created, all six components are initially set to 0.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
Dim dy As Double
loc1.XYZ(10,20,30,0,180,25)   ' Set PosWrtRef value of loc1
dy = loc1.Y                   ' dy will be set to 20
loc1.Y += 7                    ' loc1's Y value will now be 27
```

See Also

[Location Class](#) | [location_object.X](#) | [location_object.Z](#) | [location_object.Yaw](#) | [location_object.Pitch](#) | [location_object.Roll](#) | [location_object.XYZ](#)

location_object.Yaw Property

Sets and gets the displacement along the Y-axis, in units of millimeters, for the **PosWrtRef** value of a Cartesian **Location Object**.

```
location_object.Yaw = <new_value>
-or-
...location_object.Yaw
```

Prerequisites

The *location_object* must be a Cartesian **Location Object**.

Parameters

None

Remarks

Internally, the **PosWrtRef** value of a Cartesian **Location Object** is stored as a sparse 4 by 4 matrix called a “homogeneous transformation”. This matrix represents the three positional degrees-of-freedom and the three rotational degrees-of-freedom needed to fully specify a robot or part position and orientation in Cartesian coordinates. This internal representation has several computational advantages. However, entering the values for the elements of a homogeneous transformation is not very convenient. To simplify data entry, transformation values are converted to X, Y, and Z position displacement components and three Euler angles. The three Euler angles consist of a rotation about the Z-axis, followed by a rotation about the new Y-axis, followed by a rotation about the new Z-axis. This set of displacements and angles is often referred to as X, Y, Z, Yaw, Pitch, and Roll.

The property described on this page allows read and write access to one of the six components used to specify the **PosWrtRef** value of the Cartesian *location_object*.

Accessing the X, Y, and Z properties is an efficient operation. However, accessing the Yaw, Pitch, and Roll properties requires some computational overhead. Therefore, if you wish to set multiple angles, it is more efficient to utilize the **XYZ** method.

When a “New” Cartesian **Location Object** is created, all six components are initially set to 0.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
Dim ang As Double
loc1.XYZ(10,20,30,0,180,25)   ' Set PosWrtRef value of loc1
ang = loc1.Roll               ' ang will be set to 25
loc1.Roll += 5                ' loc1's Roll angle will now be 30 deg.
```

See Also

[Location Class](#) | [location object.X](#) | [location object.Y](#) | [location object.Z](#) | [location object.Pitch](#) |
[location object.Roll](#) | [location object.XYZ](#)

location_object.Z Property

Sets and gets the displacement along the Z-axis, in units of millimeters, for the **PosWrtRef** value of a Cartesian **Location Object**.

```
location_object.Z = <new_value>
-or-
...location_object.Z
```

Prerequisites

The *location_object* must be a Cartesian **Location Object**.

Parameters

None

Remarks

Internally, the **PosWrtRef** value of a Cartesian **Location Object** is stored as a sparse 4 by 4 matrix called a “homogeneous transformation”. This matrix represents the three positional degrees-of-freedom and the three rotational degrees-of-freedom needed to fully specify a robot or part position and orientation in Cartesian coordinates. This internal representation has several computational advantages. However, entering the values for the elements of a homogeneous transformation is not very convenient. To simplify data entry, transformation values are converted to X, Y, and Z position displacement components and three Euler angles. The three Euler angles consist of a rotation about the Z-axis, followed by a rotation about the new Y-axis, followed by a rotation about the new Z-axis. This set of displacements and angles is often referred to as X, Y, Z, Yaw, Pitch, and Roll.

The property described on this page allows read and write access to one of the six components used to specify the **PosWrtRef** value of the Cartesian *location_object*.

Accessing the X, Y, and Z properties is an efficient operation. However, accessing the Yaw, Pitch, and Roll properties requires some computational overhead. Therefore, if you wish to set multiple angles, it is more efficient to utilize the **XYZ** method.

When a “New” Cartesian **Location Object** is created, all six components are initially set to 0.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
Dim dz As Double
loc1.XYZ(10,20,30,0,180,25)   ' Set PosWrtRef value of loc1
dz = loc1.z                   ' dz will be set to 30
loc1.z += 7                    ' loc1's Z value will now be 37
```

See Also

[Location Class](#) | [location_object.X](#) | [location_object.Y](#) | [location_object.Yaw](#) | [location_object.Pitch](#) | [location_object.Roll](#) | [location_object.XYZ](#)

location_object.ZClearance Property

Sets and gets the distance in millimeters along a Z-axis that defines the safe approach position to a **Location Object**.

```
location_object.ZClearance = <new_value>
-or-
...location_object.ZClearance
```

Prerequisites

None

Parameters

None

Remarks

For most applications, it is not possible for the robot to move a part directly to its final destination. Normally, the destination must be approached from an intermediate position that allows the robot and part to avoid obstacles. Likewise, after picking up a part, it is typically required that the part be retracted a small distance to avoid dragging the part across the mating surface. To implement these motions to and from a final destination, GPL includes a **Move.Approach** method. Instead of moving to the “total position” of the *location_object*, this method moves the robot to a clearance position that is relative to the *location_object*.

To simplify the specification of the “approach” or “clearance” position, each *location_object* includes a **ZClearance** distance. This specifies the distance along a Z-axis for the approach position.

If the **ZWorld** property of the *location_object* is **True**, the clearance position is interpreted as being directly above (or below) the “total position” of the *location_object* in the world coordinate system at the Z value specified by **ZClearance**. For example, if the “total position” of the *location_object* is at an X, Y, Z value of (10,20,30) and **ZClearance** is 52.3 and **ZWorld** is **True**, the approach position would be (10,20,52.3).

A world Z clearance position is often used if the robot is loading or unloading a box and the robot must clear the edge of the box independent of how far into the box it must reach.

If the **ZWorld** property of the *location_object* is **False**, the clearance position is a relative distance along the negative Z-axis of the robot’s tool. This clearance distance corresponds to having the robot retract an incremental distance along the major axis of its tool or gripper. For example, if the “total position” of the *location_object* is at an X, Y, Z value of (10,20,30) and **ZClearance** is 52.3 and **ZWorld** is **False** and the robot’s tool is pointed along the positive world X-axis, the approach position would be (-42.3,20,30).

A tool Z clearance position is typically utilized if the robot is tending a number of machines and you always want to retract the gripper a fixed distance from each machine before moving to the next **Location**.

By making use of GPL's robot kinematics option, Cartesian approach specifications can be automatically applied to both Cartesian and Angles *location_objects*.

Examples

```
Dim loc1 As New Location      ' Create new Location set to default values
loc1.XYZ(10,20,30,0,180,0)    ' Define destination
loc1.ZWorld = True            ' Normally defaults to False
loc1.ZClearance = 52.3
Move.Approach (loc1, prof1)    ' Use global Profile to move to (10,20,52.3)
```

See Also

[Location Class](#) | [location_object.ZWorld](#) | [Move.Approach](#)

location_object.ZWorld Property

Sets and gets the **Boolean** flag that indicates if the **ZClearance** distance is interpreted as being along the world or tool Z-axis of a **Location Object**.

```
location_object.ZWorld = <new_Boolean_value>
-or-
...location_object.ZWorld
```

Prerequisites

None

Parameters

None

Remarks

For most applications, it is not possible for the robot to move a part directly to its final destination. Normally, the destination must be approached from an intermediate position that allows the robot and part to avoid obstacles. Likewise, after picking up a part, it is typically required that the part be retracted a small distance to avoid dragging the part across the mating surface. To implement these motions to and from a final destination, GPL includes a **Move.Approach** method. Instead of moving to the “total position” of the *location_object*, this method moves the robot to a clearance position that is relative to the *location_object*.

To simplify the specification of the “approach” or “clearance” position, each *location_object* includes a **ZClearance** distance. This specifies the distance along a Z-axis for the approach position.

If the **ZWorld** property of the *location_object* is **True**, the clearance position is interpreted as being directly above (or below) the “total position” of the *location_object* in the world coordinate system at the Z value specified by **ZClearance**. For example, if the “total position” of the *location_object* is at an X, Y, Z value of (10,20,30) and **ZClearance** is 52.3 and **ZWorld** is **True**, the approach position would be (10,20,52.3).

A world Z clearance position is often used if the robot is loading or unloading a box and the robot must clear the edge of the box independent of how far into the box it must reach.

If the **ZWorld** property of the *location_object* is **False**, the clearance position is a relative distance along the negative Z-axis of the robot’s tool. This clearance distance corresponds to having the robot retract an incremental distance along the major axis of its tool or gripper. For example, if the “total position” of the *location_object* is at an X, Y, Z value of (10,20,30) and **ZClearance** is 52.3 and **ZWorld** is **False** and the robot’s tool is pointed along the positive world X-axis, the approach position would be (-42.3,20,30).

A tool Z clearance position is typically utilized if the robot is tending a number of machines and you always want to retract the gripper a fixed distance from each machine before moving to the next **Location**.

By making use of GPL's robot kinematics option, Cartesian approach specifications can be automatically applied to both Cartesian and Angles *location_objects*.

Examples

```
Dim loc1 As New Location      ' Create new Location set to defaults
loc1.XYZ(10,20,30,0,180,0)    ' Define destination
loc1.ZWorld = True            ' Normally defaults to False
loc1.ZClearance = 52.3
Move.Approach (loc1, prof1)    ' Use global Profile, move to (10,20,52.3)
```

See Also

[Location Class](#) | [location_object.ZClearance](#) | [Move.Approach](#)

Math Class

Math Class Summary

The following sections present detailed information on the standard arithmetic and trigonometric operations that are built into GPL. As a convenience during editing, all of these operations are provided as methods to the **Math Class**. This allows programmers to display a pick list of the **Math** methods and easily see all of operations that are available.

As is standard in GPL, conversions between different arithmetic types, e.g. **Boolean**, **Integer**, **Single**, **Double**, are automatically performed as required. So, it is not necessary to have different variations on these methods to deal with the different possible mixes of input parameter data types. Also, these methods generally produce results that are formatted as **Double**'s. These results will automatically be converted to smaller data types as necessary, e.g. **Double** -> **Integer**, and will not generate an error so long as numeric overflow does not occur.

The table below briefly summarizes the methods that are described in greater detail in the following sections.

Method	Description
Math.Abs (<i>expression</i>)	Returns the absolute value of any arithmetic expression.
Math.Acos (<i>cosine</i>)	Returns the angle that corresponds to a specified cosine value.
Math.Asin (<i>sine</i>)	Returns the angle that corresponds to a specified sine value.
Math.Atan (<i>tangent</i>)	Returns the angle that corresponds to a specified tangent value.
Math.Atan2 (<i>sine_factor</i> , <i>cosine_factor</i>)	Returns the angle that corresponds to the quotient of two values.
Math.Ceiling (<i>value</i>)	Returns the smallest integer number that is greater than or equal to a value.
Math.Cos (<i>angle</i>)	Returns the cosine of a specified angle.
Math.Cosh (<i>angle</i>)	Returns the hyperbolic cosine of a specified angle.
Math.E	Returns the natural logarithmic base constant.
Math.Exp (<i>exponent</i>)	Returns the natural logarithmic constant, e , raised to a specified power.
Math.Floor (<i>value</i>)	Returns the largest integer number that is less than or equal to a value.
Math.Log (<i>value</i>)	Returns the natural logarithm (base-e logarithm) of a specified value.
Math.Log10 (<i>value</i>)	Returns the base-10 logarithm of a specified value.
Math.Max (<i>value_1</i> , <i>value_2</i>)	Returns the larger of two values.
Math.Min (<i>value_1</i> , <i>value_2</i>)	Returns the smaller of two values.
Math.PI	Returns the constant π .
Math.Pow (<i>base</i> , <i>exponent</i>)	Returns a specified base value raised to a specified power.
Math.Sign (<i>value</i>)	Returns a number that indicates the sign of a specified value.

<u>Math.Sin</u> (<i>angle</i>)	Returns the sine of a specified angle.
<u>Math.Sinh</u> (<i>angle</i>)	Returns the hyperbolic sine of a specified angle.
<u>Math.Sqrt</u> (<i>value</i>)	Returns the square root of a value.
<u>Math.Tan</u> (<i>angle</i>)	Returns the tangent of a specified angle.
<u>Math.Tanh</u> (<i>angle</i>)	Returns the hyperbolic tangent of a specified angle.

Math.Abs Method

Returns the absolute value of any arithmetic expression.

```
...Math.Abs( expression )
```

Prerequisites

None

Parameters

expression

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns the absolute value (i.e. the magnitude) of any numerical expression. That is, if the expression has a value greater than or equal to zero, its value is returned unchanged. If the expression value is negative, it is negated and returned as a positive value.

Examples

```
Dim value As Single  
value = Math.Abs(-1.23)      ' Sets value to 1.23  
value = Math.Abs(0)         ' Sets value to 0  
value = Math.Abs(3)         ' Sets value to 3
```

See Also

[Math Class](#)

Math.Acos Method

Returns the angle that corresponds to a specified cosine value

```
...Math.Acos( cosine )
```

Prerequisites

None

Parameters

cosine

A required expression that evaluates to the cosine of an angle. This value must be in the range $-1 \leq \text{cosine} \leq 1$.

Remarks

Returns the angle, in radians, that corresponds to a specified *cosine* value. That is, if the cosine of an angle A is B, then this arc cosine function returns A when given a value of B.

Since the cosine function generates the same value for both positive and negative angles, the **Math.Acos** method returns a value between 0 and π for any valid input expression. If the full range of angles is required, the **Math.Atan2** method should be used whenever possible.

To convert radians to degrees, multiply the radians times $180/\pi$. }

Examples

```
Dim angle As Single
angle = Math.Acos(-1)           ' Sets angle to Pi
angle = Math.Acos(Math.Sqrt(2)/2) ' Sets angle to Pi/4
angle = Math.Acos(Math.Cos(-.5)) ' Sets angle to 0.5 radians
```

See Also

[Math Class](#) | [Math.Atan2](#)

Math.Asin Method

Returns the angle that corresponds to a specified sine value.

```
...Math.Asin( sine )
```

Prerequisites

None

Parameters

sine

A required expression that evaluates to the sine of an angle. This value must be in the range $-1 \leq \textit{sine} \leq 1$.

Remarks

Returns the angle, in radians, that corresponds to a specified *sine* value. That is, if the sine of an angle A is B, then this arc sine function returns A when given a value of B.

Since the sine function repeats the same series of answers when an angle traverses from $\pi/2$ to 0 to $-\pi/2$ as when an angle moves from $\pi/2$ to $-\pi$ to $-\pi/2$, the **Math.Asin** function cannot distinguish these two cases and always returns values that range from $\pi/2$ to $-\pi/2$. If the full range of angles is required, the **Math.Atan2** method should be used whenever possible.

To convert radians to degrees, multiply the radians times $180/\pi$.

Examples

```
Dim angle As Single
angle = Math.Asin(-1)           ' Sets angle to -Pi/2
angle = Math.Asin(Math.Sqrt(2)/2) ' Sets angle to Pi/4
angle = Math.Asin(Math.Sin(Math.PI-.5)) ' Sets angle to 0.5 radians
```

See Also

[Math Class](#) | [Math.Atan2](#)

Math.Atan Method

Returns the angle that corresponds to a specified tangent value.

```
...Math.Atan( tangent )
```

Prerequisites

None

Parameters

tangent

A required expression that evaluates to the tangent of an angle.

Remarks

Returns the angle, in radians, that corresponds to a specified *tangent* value. That is, if the tangent of an angle A is B, then this arc tangent function returns A when given a value of B.

Since the tangent function repeats the same series of answers over two ranges of angles: when an angle traverses from 0 to $\pi/2$ as when an angle moves from $-\pi$ to $-\pi/2$ and then again when an angle traverses from 0 to $-\pi/2$ as when an angle moves from $-\pi$ to $\pi/2$, the **Math.Atan** function cannot distinguish these cases and always returns values that range from $\pi/2$ to $-\pi/2$.

In addition, as the angle gets close to $\pi/2$ or $-\pi/2$, the input parameter for this method must approach positive or negative infinity.

To deal with both of these problems, the **Math.Atan2** method should be used whenever possible.

To convert radians to degrees, multiply the radians times $180/\pi$.

Examples

```
Dim angle As Single
angle = Math.Atan(1)           ' Sets angle to Pi/4
angle = Math.Atan(0)           ' Sets angle to 0
angle = Math.Atan(Math.Tan(-3*Math.PI/4)) ' Sets angle to Pi/4
```

See Also

Math Class | [Math.Atan2](#) Method

Math.Atan2 Method

Returns the angle that corresponds to the quotient of two values.

```
...Math.Atan2( sine_factor, cosine_factor )
```

Prerequisites

None

Parameters

sine_factor

A required expression, which when divided by *cosine_factor*, is equal to the tangent of the angle.

cosine_factor

A required expression, which when divided into *sine_factor*, is equal to the tangent of the angle.

Remarks

Returns the angle, in radians, that corresponds to the tangent value computed from *sine_factor/cosine_factor* and using the signs of *sine_factor* and *cosine_factor* to uniquely determine the quadrant of the angle.

As a simplified example, if A is the sine of an angle C and B is the cosine of the angle, then this arc tangent function returns C when given the values A and B.

Unlike the **Math.Atan** method, this method can return the full range of angles from $+\pi$ to $-\pi$. In addition, it does not suffer from requiring infinite valued parameters in order to represent any angular value. So, **Math.Atan2** should be used whenever possible instead of **Math.Atan**.

To convert radians to degrees, multiply the radians times $180/\pi$.

Examples

```
Dim angle As Single
angle = Math.Atan2(1,0)           ' Sets angle to Pi/2
angle = Math.Atan2(.5,-.5)        ' Sets angle to 3*Pi/4
angle = Math.Atan2(-.707,.707)    ' Sets angle to -Pi/4
```

See Also

Math Class

Math.Ceiling Method

Returns the smallest integer number that is greater than or equal to a value.

...Math.Ceiling (*value*)

Prerequisites

None

Parameters

value

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns the smallest integer number that is greater than or equal to the *value*. This is sometimes referred to as rounding towards positive infinity.

Examples

```
Dim bigger As Single
bigger = Math.Ceiling(10.9999) ' Sets bigger equal to 11
bigger = Math.Ceiling(11)      ' Sets bigger equal to 11
bigger = Math.Ceiling(11.0001) ' Sets bigger equal to 12
```

See Also

[Math Class](#)

Math.Cos Method

Returns the cosine of a specified angle.

```
...Math.Cos( angle )
```

Prerequisites

None

Parameters

angle

A required expression that evaluates to an angle in units of radians. This angle is not limited to values between $-\pi$ and $+\pi$ and can be arbitrarily large.

Remarks

Returns the cosine of the *angle* that is specified in radians. The result of this method ranges from -1 to $+1$.

To convert degrees to radians, multiply the degrees times $\pi/180$.

Examples

```
Dim cos_val As Single
cos_val = Math.Cos(0)           ' Sets cos_val to 1
cos_val = Math.Cos(21*Math.PI)  ' Sets cos_val to -1
cos_val = Math.Cos(45*Math.PI/180) ' Sets cos_val to 0.7071
```

See Also

[Math Class](#)

Math.Cosh Method

Returns the hyperbolic cosine of a specified angle.

```
...Math.Cosh( angle )
```

Prerequisites

None

Parameters

angle

A required expression that evaluates to an angle in units of radians. This angle is not limited to values between $-\pi$ and $+\pi$ and can be arbitrarily large.

Remarks

Returns the hyperbolic cosine of the *angle* that is specified in radians.

To convert degrees to radians, multiply the degrees times $\pi/180$.

See Also

[Math Class](#)

Math.E Method

Returns the natural logarithmic base constant.

...Math.E

Prerequisites

None

Parameters

None

Remarks

Returns the constant that is the base value for the natural logarithmic functions,
2.7182818284590452354

Examples

```
Dim value As Single  
value = Math.Pow(Math.E, 2)
```

See Also

[Math Class](#)

Math.Exp Method

Returns the natural logarithmic constant, **e**, raised to a specified power.

```
...Math.Exp( exponent )
```

Prerequisites

None

Parameters

exponent

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns the value of the natural logarithmic constant, **Math.E**, raised to the *exponent* power (i.e. **Math.E**^{*exponent*}).

Examples

```
Dim e_val As Single
e_val = Math.Exp(2)           ' Sets e_val to 7.3891
e_val = Math.Exp(-2.2)        ' Sets e_val to 0.1108
e_val = Math.Exp(Math.Log(17.1)) ' Sets e_val to 17.1
```

See Also

[Math Class](#)

Math.Floor Method

Returns the largest integer number that is less than or equal to a value.

...Math.Floor (*value*)

Prerequisites

None

Parameters

value

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns the largest integer number that is less than or equal to the *value*. This is sometimes referred to as rounding towards negative infinity.

Examples

```
Dim smaller As Single
smaller = Math.Floor(10.9999) ' Sets smaller equal to 10
smaller = Math.Floor(11)     ' Sets smaller equal to 11
smaller = Math.Floor(11.0001) ' Sets smaller equal to 11
```

See Also

[Math Class](#)

Math.Log Method

Returns the natural logarithm (base-e logarithm) of a specified value.

```
...Math.Log( value )
```

Prerequisites

None

Parameters

value

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns the exponent to which the natural logarithmic constant, **Math.E**, must be raised in order to produce the *value*.

Examples

```
Dim ln_exp As Single
ln_exp = Math.Log(10)           ' Sets ln_exp to 2.3026
ln_exp = Math.Log(Math.E)       ' Sets ln_exp to 1
ln_exp = Math.Log(Math.Exp(3.4)) ' Sets ln_exp to 3.4
```

See Also

[Math Class](#)

Math.Log10 Method

Returns the base-10 logarithm of a specified value.

```
...Math.Log10( value )
```

Prerequisites

None

Parameters

value

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns the exponent to which the number 10 must be raised in order to produce the *value*.

Examples

```
Dim l_exp As Single
l_exp = Math.Log10(10)           ' Sets l_exp to 1
l_exp = Math.Log10(0.01)        ' Sets l_exp to -2
l_exp = Math.Log10(Math.Pow(10,3.4)) ' Sets l_exp to 3.4
```

See Also

[Math Class](#)

Math.Max Method

Returns the larger of two values.

```
...Math.Max( value_1, value_2 )
```

Prerequisites

None

Parameters

value_1

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

value_2

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns the larger of two numerical values, *value_1* or *value_2*.

Examples

```
Dim bigger As Single
bigger = Math.Max(-5, -4.9)           ' Sets bigger to -4.9
bigger = Math.Max(-20/-4, 3)         ' Sets bigger to 5
bigger = Math.Max(Math.Min(100, 33), 55) ' Sets bigger to 55
```

See Also

[Math Class](#)

Math.Min Method

Returns the smaller of two values.

```
...Math.Min( value_1, value_2 )
```

Prerequisites

None

Parameters

value_1

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

value_2

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns the smaller of two numerical values, *value_1* or *value_2*.

Examples

```
Dim smaller As Single
smaller = Math.Min(-5, -4.9)           ' Sets smaller to -5
smaller = Math.Min(-20/-4, 3)         ' Sets smaller to 3
smaller = Math.Min(Math.Max(100, 33), 55) ' Sets smaller to 55
```

See Also

[Math Class](#)

Math.PI Method

Returns the constant π .

...Math.PI

Prerequisites

None

Parameters

None

Remarks

Returns the value of π , 3.14159265358979323846.

Examples

```
Dim to_deg, to_rad As Double  
to_deg = 180/Math.PI           ' Conversion factor from radians to degrees  
to_rad = Math.PI/180           ' Conversion factor from degrees to radians
```

See Also

[Math Class](#)

Math.Pow Method

Returns a specified base value raised to a specified power.

```
...Math.Pow( base, exponent )
```

Prerequisites

None

Parameters

base

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

exponent

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns the value of *base* raised to the *exponent* power (i.e. $base^{exponent}$). The *base* cannot be negative if the *exponent* is a fractional value. Also, the *base* cannot be zero if the *exponent* is less than or equal to zero.

Examples

```
Dim p_val As Single
p_val = Math.Pow(2, 3)           ' Sets p_val to 8
p_val = Math.Pow(3, -2.2)       ' Sets p_val to 0.08919
p_val = Math.Pow(Math.E, Math.Log(17.1)) ' Sets p_val to 17.1
```

See Also

[Math Class](#)

Math.Sign Method

Returns a number that indicates the sign of a specified value.

...Math.Sign (*value*)

Prerequisites

None

Parameters

value

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns a 1.0 if the *value* is greater than zero, 0 if the *value* is equal to zero, otherwise – 1.0 to indicate that the *value* is negative.

Examples

```
Dim v_sign As Single, int_v_sign As Integer
v_sign = Math.Sign(-21.2/(-2.3))      ' Sets v_sign equal to 1.0
int_v_sign = Math.Sign(-7.2)         ' Sets int_v_sign equal to -1
```

See Also

[Math Class](#)

Math.Sin Method

Returns the sine of a specified angle

...Math.Sin(*angle*)

Prerequisites

None

Parameters

angle

A required expression that evaluates to an angle in units of radians. This angle is not limited to values between $-\pi$ and $+\pi$ and can be arbitrarily large.

Remarks

Returns the sine of the *angle* that is specified in radians. The result of this method ranges from -1 to $+1$.

To convert degrees to radians, multiply the degrees times $\pi/180$.

Examples

```
Dim sin_val As Single
sin_val = Math.Sin(-Math.PI/2)      ' Sets sin_val to -1
sin_val = Math.Sin(20.5*Math.PI)    ' Sets sin_val to 1
sin_val = Math.Sin(45*Math.PI/180) ' Sets sin_val to 0.7071
```

See Also

[Math Class](#)

Math.Sinh Method

Returns the hyperbolic sine of a specified angle.

```
...Math.Sinh( angle )
```

Prerequisites

None

Parameters

angle

A required expression that evaluates to an angle in units of radians. This angle is not limited to values between $-\pi$ and $+\pi$ and can be arbitrarily large.

Remarks

Returns the hyperbolic sine of the *angle* that is specified in radians.

To convert degrees to radians, multiply the degrees times $\pi/180$.

See Also

[Math Class](#)

Math.Sqrt Method

Returns the square root of a value.

```
...Math.Sqrt (value)
```

Prerequisites

None

Parameters

value

A required expression that evaluates to any numerical data type, e.g. **Integer**, **Single**, **Double**.

Remarks

Returns the square root of any positive number as a double precision value.

Examples

```
Dim root As Single, int_root As Integer
root = Math.Sqrt(1.44)      ' Sets root equal to 1.2
int_root = Math.Sqrt(1.69)  ' Sets int_root equal to 1
```

See Also

[Math Class](#)

Math.Tan Method

Returns the tangent of a specified angle.

```
...Math.Tan( angle )
```

Prerequisites

None

Parameters

angle

A required expression that evaluates to an angle in units of radians. This angle is not limited to values between $-\pi$ and $+\pi$ and can be arbitrarily large.

Remarks

Returns the tangent of the *angle* that is specified in radians. Since the returned value will be extremely large as the *angle* approaches $\pi/2$ or $-\pi/2$, it is normally desirable to use the **Math.Sin** and **Math.Cos** methods in place of this operation.

To convert degrees to radians, multiply the degrees times $\pi/180$.

Examples

```
Dim tan_val As Single
tan_val = Math.Tan(0)           ' Sets tan_val to 0
tan_val = Math.Tan(Math.PI/4)   ' Sets tan_val to 1
tan_val = Math.Tan(-45*Math.PI/180) ' Sets tan_val to -1
```

See Also

[Math Class](#)

Math.Tanh Method

Returns the hyperbolic tangent of a specified angle.

```
...Math.Tanh( angle )
```

Prerequisites

None

Parameters

angle

A required expression that evaluates to an angle in units of radians. This angle is not limited to values between $-\pi$ and $+\pi$ and can be arbitrarily large.

Remarks

Returns the hyperbolic tangent of the *angle* that is specified in radians.

To convert degrees to radians, multiply the degrees times $\pi/180$.

See Also

[Math Class](#)

Modbus Class

Modbus Class Summary

The **Modbus Class** in GPL supports master access to MODBUS/TCP slave devices connected to the local Ethernet network. MODBUS/TCP is an "open" de facto standard protocol that is widely used in the industrial manufacturing environment to communicate between intelligent devices. It has been implemented by hundreds of vendors on thousands of different products to communicate digital and analog I/O and register data between devices.

The tables below briefly summarize the properties and methods for this Class, which are described in greater detail in the following sections.

Modbus Class Member	Type	Description
New Modbus	Constructor Method	Creates an object for a MODBUS connection and specifies the IP address.
<u>modbus_obj.Close</u>	Method	Closes any connections associated with this object.
<u>modbus_obj.ReadCoils</u>	Method	Reads one or more outputs.
<u>modbus_obj.ReadDeviceId</u>	Method	Reads the device ID strings.
<u>modbus_obj.ReadDiscreteInputs</u>	Method	Reads one or more inputs.
<u>modbus_obj.ReadHoldingRegisters</u>	Method	Reads one or more holding registers.
<u>modbus_obj.ReadInputRegisters</u>	Method	Reads one or more input registers.
<u>modbus_obj.Timeout</u>	Get/Set Property	Gets or sets the timeout, in milliseconds, that this connection will wait for a reply before throwing an exception.
<u>modbus_obj.WriteMultipleCoils</u>	Method	Writes multiple outputs.
<u>modbus_obj.WriteMultipleRegisters</u>	Method	Writes multiple holding registers.
<u>modbus_obj.WriteSingleCoil</u>	Method	Writes a single output.
<u>modbus_obj.WriteSingleRegister</u>	Method	Writes a single holding register.

modbus_object.Close Method

Closes the network connection associated with a **Modbus** object.

```
modbus_object.Close
```

Prerequisites

None

Parameters

None

Remarks

The **Close** method may be used to close the network connection and free up resources.

If no Modbus connection is active, no error occurs.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")
Dim mb As New Modbus(ep)
...
mb.Close()
```

See Also

[Modbus Class](#)

modbus_object.ReadCoils Method

Reads one or more outputs from a MODBUS slave and returns the values in a **Boolean** array.

```
modbus_object.ReadCoils(start, number, value_array)
```

Prerequisites

None

Parameters

start

A required **Integer** expression that specifies the number of the first coil to be read.

number

A required **Integer** expression that defines the number of coils to be read.

value_array

A required **Boolean** array that receives the output values. The length of the array is changed to reflect the number of values read.

Remarks

This method issues a MODBUS/TCP Read Coils request (function 1).

A new connection to the MODBUS slave is made if none currently exists.

If any network errors occur, this method throws an exception.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")
Dim mb As New Modbus(ep)
Dim bool() As Boolean
mb.ReadCoils(1, 16, bool)           ' Read 16 outputs
```

See Also

[Modbus Class](#) | [modbus_object.WriteMultipleCoils](#) | [modbus_object.WriteSingleCoil](#)

modbus_object.ReadDeviceID Method

Reads device identification information from a MODBUS slave and returns as a **String** value.

```
... modbus_object.ReadDeviceId(object_id)
```

Prerequisites

None

Parameters

object_id

A required **Integer** expression that evaluates to a number from 0 to 255 that selects the identification information to be returned.

Remarks

This method issues a MODBUS Read Device Identification request (MEI-type 13) using the Encapsulated Interface Transport (function 43) to retrieve identification information from the slave. The Read Device ID code is always set to 1.

The *object_id* parameter selects the identification information to be returned. Some standard values are:

Object ID	Description
0	Vendor name
1	Product code
2	Major and Minor Revision

Consult the MODBUS/TCP standard for the meaning of other *object_id* values.

Not all MODBUS devices support this function. The **String** value returned by this method depends on the particular device being referenced. Consult the manual for your MODBUS slave device for details.

A new connection to the MODBUS slave is made if none currently exists.

If any network errors occur, this method throws an exception.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")
Dim mb As New Modbus(ep)
Dim id As String
id = mb.ReadDeviceId(0)           ' Read vendor name
```

See Also

[Modbus Class](#)

modbus_object.ReadDiscreteInputs Method

Reads one or more inputs from a MODBUS slave and returns the values in a **Boolean** array.

```
modbus_object.ReadDiscreteInputs(start, number, value_array)
```

Prerequisites

None

Parameters

start

A required **Integer** expression that specifies the number of the first input to be read.

number

A required **Integer** expression that defines the number of inputs to be read.

value_array

A required **Boolean** array that receives the input values. The length of the array is changed to reflect the number of values read.

Remarks

This method issues a MODBUS/TCP Read Discrete Inputs request (function 2).

A new connection to the MODBUS slave is made if none currently exists.

If any network errors occur, this method throws an exception.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")
Dim mb As New Modbus(ep)
Dim bool() As Boolean
mb.ReadDiscreteInputs(1, 16, bool) ' Read 16 inputs
```

See Also

[Modbus Class](#) | [modbus_object.ReadInputRegisters](#)

modbus_object.ReadHoldingRegisters Method

Reads one or more holding registers from a MODBUS slave and returns the values in an **Integer** array.

```
modbus_object.ReadHoldingRegisters(start, number, value_array)
```

Prerequisites

None

Parameters

start

A required **Integer** expression that specifies the number of the first register to be read.

number

A required **Integer** expression that defines the number of registers to be read.

value_array

A required **Integer** array that receives the register values. The length of the array is changed to reflect the number of values read.

Remarks

This method issues a MODBUS/TCP Read Holding Registers request (function 3).

Each holding register contains a 16-bit unsigned integer value, from 0 to 65535.

A new connection to the MODBUS slave is made if none currently exists.

If any network errors occur, this method throws an exception.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")
Dim mb As New Modbus(ep)
Dim regs() As Integer
mb.ReadHoldingRegisters(1, 16, regs) ' Read 16 values
```

See Also

[Modbus Class](#) | [*modbus object*.ReadInputRegisters](#) | [*modbus object*.WriteMultipleRegisters](#) | [*modbus object*.WriteSingleRegister](#)

modbus_object.ReadInputRegisters Method

Reads one or more input registers from a MODBUS slave and returns the values in an **Integer** array.

```
modbus_object.ReadInputRegisters(start, number, value_array)
```

Prerequisites

None

Parameters

start

A required **Integer** expression that specifies the number of the first register to be read.

number

A required **Integer** expression that defines the number of registers to be read.

value_array

A required **Integer** array that receives the register values. The length of the array is changed to reflect the number of values read.

Remarks

This method issues a MODBUS/TCP Read Input Registers request (function 4).

Each input register contains a 16-bit unsigned integer value, from 0 to 65535.

A new connection to the MODBUS slave is made if none currently exists.

If any network errors occur, this method throws an exception.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")
Dim mb As New Modbus(ep)
Dim regs() As Integer
mb.ReadInputRegisters(1, 16, regs) ' Read 16 values
```

See Also

[Modbus Class](#) | [*modbus object*.ReadHoldingRegisters](#) | [*modbus object*.WriteMultipleRegisters](#) | [*modbus object*.WriteSingleRegister](#)

modbus_object.Timeout Property

Sets or gets the timeout period, in milliseconds, that GPL waits for a response from a MODBUS slave.

```
modbus_object.Timeout = <timeout>  
-or-  
... modbus_object.Timeout
```

Prerequisites

None

Parameters

None

Remarks

The property allows you to set the timeout period for all **Modbus** methods that perform I/O with the MODBUS slave.

If this time is exceeded, the method throws an exception. If the *timeout* period is set to 0, the timeout is disabled and a request may wait indefinitely.

Each *modbus_object* has an independent timeout value.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")  
Dim mb As New Modbus(ep)  
mb.Timeout = 2000           ' Timeout in 2 seconds
```

See Also

[Modbus Class](#)

modbus_object.WriteMultipleCoils Method

Writes one or more outputs to a MODBUS slave.

```
modbus_object.WriteMultipleCoils(start, value_array)
```

Prerequisites

None

Parameters

start

A required **Integer** expression that specifies the number of the first coil to be written.

value_array

A required **Boolean** array that contains the output values to be written. The length of the array determines the number of coils written.

Remarks

This method issues a MODBUS/TCP Write Multiple Coils request (function 15).

A new connection to the MODBUS slave is made if none currently exists.

If any network errors occur, this method throws an exception.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")
Dim mb As New Modbus(ep)
Dim bool(15) As Boolean           ' Array length is 16
bool(0) = True                    ' First output set, rest clear
mb.WriteMultipleCoils(1, bool)   ' Write 16 outputs
```

See Also

[Modbus Class](#) | [modbus_object.WriteSingleCoil](#)

modbus_object.WriteMultipleRegisters Method

Writes one or more holding register values to a MODBUS slave.

```
modbus_object.WriteMultipleRegisters(start, value_array)
```

Prerequisites

None

Parameters

start

A required **Integer** expression that specifies the number of the first holding register to be written.

value_array

A required **Integer** array that contains the register values to be written. The length of the array determines the number of registers written.

Remarks

This method issues a MODBUS/TCP Write Multiple Registers request (function 16).

The holding registers are 16-bit unsigned integer values. Only the low 16-bits of values in *value_array* are used. No error is reported if values are too big to fit in 16 bits.

A new connection to the MODBUS slave is made if none currently exists.

If any network errors occur, this method throws an exception.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")
Dim mb As New Modbus(ep)
Dim value() As Integer
Redim value(7) ' Set array length to 8
value(0) = 111 ' First reg is 111, rest are zero
mb.WriteMultipleRegisters(1, value) ' Write 8 registers
```

See Also

[Modbus Class](#) | [modbus_object.WriteSingleRegister](#)

modbus_object.WriteSingleCoil Method

Writes a single output to a MODBUS slave.

```
modbus_object.WriteSingleCoil(coil, value)
```

Prerequisites

None

Parameters

coil

A required **Integer** expression that specifies the number of the coil to be written.

value

A required **Boolean** expression that determines the output to be written. Any non-zero value is considered **True**.

Remarks

This method issues a MODBUS/TCP Write Single Coil request (function 5).

If more than one coil is to be changed, it is much more efficient to use the **WriteMultipleCoils** method than multiple **WriteSingleCoil** methods.

A new connection to the MODBUS slave is made if none currently exists.

If any network errors occur, this method throws an exception.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")
Dim mb As New Modbus(ep)
mb.WriteSingleCoil(1, True)      ' Turn on coil 1
mb.WriteSingleCoil(2, False)    ' Turn off coil 2
```

See Also

[Modbus Class](#) | [modbus_object.WriteMultipleCoils](#)

modbus_object.WriteSingleRegister Method

Writes a single holding register value to a MODBUS slave.

```
modbus_object.WriteSingleRegister(register, value)
```

Prerequisites

None

Parameters

register

A required **Integer** expression that specifies the number of the holding register to be written.

value

A required **Integer** expression that determines the output to be written to the holding register.

Remarks

This method issues a MODBUS/TCP Write Single Register request (function 6).

The holding registers are 16-bit unsigned integer values. Only the low 16-bits of *value* are used. No error is reported if *value* is too big to fit in 16 bits.

If more than one register is to be changed, it is much more efficient to use the **WriteMultipleRegisters** method than multiple **WriteSingleRegister** methods.

A new connection to the MODBUS slave is made if none currently exists.

If any network errors occur, this method throws an exception.

Examples

```
Dim ep As New IPEndPoint("192.168.0.150")
Dim mb As New Modbus(ep)
mb.WriteSingleRegister(1, 123)
```

See Also

[Modbus Class](#) | [modbus_object.WriteMultipleRegisters](#)

Move Class

Move Class Summary

The following pages provide detailed information on the methods of the **Move Class**. This class provides the means for issuing motion commands to a robot.

The GPL system supports position, velocity, and torque-controlled motions. In the standard case of position-controlled motions, a **Move** method requires two arguments: a motion destination and a motion performance specification. Typically, a **Location Object** specifies the destination and a **Profile Object** defines the performance parameters. The **Location** can specify the destination in either Cartesian or joint coordinates and includes clearance position information that is utilized by selected **Move** methods. The **Profile** specifies the type of path to follow, i.e. straight-line or joint interpolated and how fast the robot is to move.

As an ease-of-use feature, several **Move** methods are provided for defining the destination of a motion. For example, methods are provided for specifying if the robot is to move directly to a destination, move to the clearance position of a destination, move relative to the previous destination, or move a single axis.

The table below briefly summarized the methods that are described in greater detail in the following sections.

Member	Type	Description
Move.Approach	Method	Moves to the clearance position for a specified Location .
Move.Arc	Method	Moves the tool tip of the robot along an arc path defined by three Locations .
Move.Circle	Method	Moves the tool tip of the robot around a complete circle defined by three Locations .
Move.Delay	Method	Pauses execution of motions for a specified period of time, in seconds.
Move.Extra	Method	Moves extra, independent axes during the next motion to a Cartesian Location .
Move.ForceOverlap	Method	Bypasses the system's normal motion blending features and explicitly defines how the execution of two sequential motions are to be overlapped.
Move.Loc	Method	Basic instruction to move to a specified destination Location .
Move.OneAxis	Method	Convenience method to move a single axis of a robot.
Move.Rel	Method	Moves to a Location that is relative to the final position and orientation of the previous motion.
Move.SetJogCommand	Method	Sets or changes the specific mode, axis and speed during jog (manual) control mode.
Move.SetSpeeds	Method	Sets new target speeds and accelerations for all axes during velocity control mode.
Move.SetTorques	Method	Sets new target torque output levels for all

		motors in torque control mode.
<u>Move.StartJogMode</u>	Method	Initiates execution of jog (manual) control mode.
<u>Move.StartTorqueCntrl</u>	Method	Initiates execution of torque control mode for one or more motors.
<u>Move.StartVelocityCntrl</u>	Method	Switches all axes of a robot to velocity control mode in place of position control mode.
<u>Move.StopSpecialModes</u>	Method	Terminates execution of any active special trajectory control modes.
<u>Move.Trigger</u>	Method	Primes the system to automatically assert a digital output signal at a prescribed trigger position during the next motion.
<u>Move.WaitForEOM</u>	Method	Pauses GPL program execution until the current motion is completed.

Move.Approach Method

Moves the robot in a position-controlled motion to the clearance position for a specified **Location**.

Move.Approach (*location_1*, *profile_1*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be homed.
- The robot must be **Attached** by the thread.

Parameters

location_1

A required **Location Object** or an expression that evaluates to a **Location Object** value. Can be either a Cartesian or an Angles type value.

profile_1

A required **Profile Object** or an expression that evaluates to a **Profile Object** value. Can specify either Cartesian straight-line or joint interpolated motions.

Remarks

This method simultaneously moves all of the axes of the robot in a coordinated, position-controlled motion to a clearance position for a specified **Location**.

In many cases, as the robot moves towards a part position or is being retracted from a part position, it must first move through an intermediate clearance position. For example, when picking up a part, it is often necessary to position the robot's gripper directly over the part before moving down to pick it up. Likewise, after gripping a part, it is often necessary to retract the robot's end effector and the part in order to clear other parts or to avoid scrapping the part along it's supporting surface.

Since this is such a common operation, all **Location Objects** contain information on their required clearance position. The **Approach** method automatically makes use of this clearance data to compute an intermediate "approach position" that is taken as the destination for the **Approach** method's motion.

Specifically, each **Location** contains a **ZClearance** distance and a **ZWorld Boolean** flag. The **ZClearance** property specifies the Z-axis offset distance for the approach position in millimeters. If the **ZWorld** property is **True**, the clearance position is interpreted as being directly above (or below) the "total position" of the **Location** in the world coordinate

system at the Z value specified by **ZClearance**. For example, if the “total position” of a **Location** is at an X, Y, Z value of (10,20,30) and **ZClearance** is 52.3 and **ZWorld** is **True**, the approach position would be (10,20,52.3).

A world Z clearance position is often used if the robot is loading or unloading a box and the robot must clear the edge of the box independent of how far into the box it must reach.

If the **ZWorld** property of a **Location** is **False**, the clearance position is a relative distance along the negative Z-axis of the robot’s tool. This clearance distance corresponds to having the robot retract an incremental distance along the major axis of its tool or gripper. For example, if the “total position” of a **Location** is at an X, Y, Z value of (10,20,30) and **ZClearance** is 52.3 and **ZWorld** is **False** and the robot’s tool is pointed along the positive world X-axis, the approach position would be (-42.3,20,30).

A tool Z clearance position is typically utilized if the robot is tending a number of machines and you always wish to retract the gripper a fixed distance from each machine before moving to the next **Location**.

By making use of GPL’s robot kinematics option, approach specifications can be automatically applied to both **Cartesian** and **Angles Location Objects**.

Once the **Approach** method computes the desired motion destination, the motion execution is identical to the **Move.Loc** method. The motion can be a **Straight**-line or joint interpolated motion, can be blended with the previous and the next motions as desired, and the performance parameters are defined by *profile_1* (e.g. **Speed**, **Accel**, **Decel**, **AccelRamp**, **DecelRamp**, **InRange**).

Examples

```
Dim prof1 As New Profile      ' Create new profile initialized to default values
Dim loc1 As New Location     ' Create new location value
loc1.XYZ(10,20,30,0,180,20)  ' Define position to move to
loc1.ZClearance = 10         ' Require 10 mm clearance in Tool
Move.Approach(loc1,prof1)    ' Move to clearance position
Move.Loc(loc1, prof1)        ' Move to loc1 using prof1
```

See Also

[Location Class](#) | [Move Class](#) | [Move.Loc](#) | [Move.Rel](#) | [Profile Class](#)

Move.Arc Method

Moves the robot's tool tip in a circular arc defined by three **Location** values.

Move.Arc (*location_1*, *location_2*, *profile_1*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be homed.
- The robot must be **Attached** by the thread.

Parameters

location_1

A required **Location Object** or an expression that evaluates to a **Location Object** value. Can be either a Cartesian or an Angles type value.

location_2

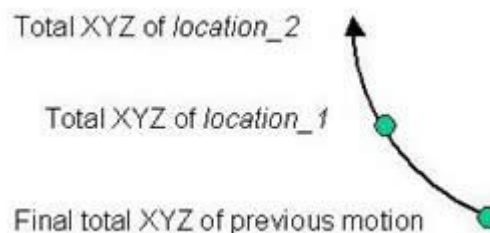
A required **Location Object** or an expression that evaluates to a **Location Object** value. Can be either a Cartesian or an Angles type value.

profile_1

A required **Profile Object** or an expression that evaluates to a **Profile Object** value. The **Straight** property that specifies a Cartesian straight-line or a joint interpolated motion is ignored since the motion is always performed in Cartesian coordinates.

Remarks

This method simultaneously moves all of the axes of a robot in a coordinated, position controlled motion such that the robot's tool tip follows a circular arc path. The arc is defined by the XYZ values of the final position of the previous motion and *location_1* and *location_2*. The performance parameters for the motion are defined by the **Profile Object**, *profile_1* (e.g. **Speed**, **Accel**, **Decel**, **AccelRamp**, **DecelRamp**).



The circular arc begins at the final "total XYZ position" of the previous motion, goes through the "total" XYZ position of *location_1* and terminates at the "total" XYZ position of *location_2*. The "total position" of *location_1* and *location_2* are computed as the results of evaluating each **Location's PosWrtRef** value relative to the "total position" of their respective reference frames, if any. If a **Location** is specified as an **Angles** type, its XYZ position is computed using the kinematic model for the attached robot.

If *profile_1* has its **InRange** property set to zero or a positive value, the system will bring the robot to a stop at *location_2*. If this property is negative and the next motion statement is executed before this motion reaches its destination, GPL will attempt to blend the two motions together into a "continuous path". Circular interpolated motions can be blended with any of the motion types, i.e. Cartesian straight-line, joint interpolated or other circular interpolated motions.

If the previous motion is still in process when the **Move.Arc** instruction is executed, the **Move.Arc** instruction will temporarily suspend execution of its thread. At the conclusion of the previous motion or as soon as the new **Arc** motion starts to be blended with the previous motion, the thread will automatically continue execution at the next instruction in the GPL procedure.

The following are special notes regarding the use of the **Arc** method.

- The circular arc can be defined in any arbitrary orientation and need not lie in an cardinal plane.
- The XYZ value of *location_1* need not be halfway between the starting and ending positions of the arc although values closer to the mid point will more accurately define the plane of the arc.
- If the three XYZ points that define the arc lie in a straight-line, the **Arc** method is automatically converted to a Cartesian straight-line motion to *location_2*.
- When blending two **Arc** motions, the s-curve **AccelRamp** and **DecelRamp** should be set to 0 and the **Accel** and **Decel** properties should be set high to ensure that the path tracks the circular path as closely as possible.
- As with straight-line motions, the orientation of the tool of the robot is smoothly rotated from the final orientation of the previous motion to the orientation of the final position, *location_2*. The specific rotation method is a function of the kinematic module being utilized.

Examples

```
Dim p0 As New Location      ' Create location objects
Dim p1 As New Location
Dim p2 As New Location
Dim p3 As New Location
Dim p4 As New Location

p0.XYZ(100,200,-100,0,180,0) ' Define two semi-circles
p1.XYZ(200,100,-100,0,180,0) '   that form an "S"
p2.XYZ(300,200,-100,0,180,0)
p3.XYZ(400,300,-100,0,180,0)
p4.XYZ(500,200,-100,0,180,0)

Move.Loc(p0,pf_start)        ' Move to start position
Move.Arc(p1,p2,pf_on_path)   ' Follow first semi-circle
Move.Arc(p3,p4,pf_on_path)   ' Follow second semi-circle
Move.WaitForEOM              ' Pause thread until motion done
```

See Also

[Location Class](#) | [Move Class](#) | [Move.Circle](#) | [Move.Loc](#) | [Profile Class](#)

Move.Circle Method

Moves the robot's tool tip in a complete circle defined by three **Location** values.

Move.Circle (*location_1*, *location_2*, *profile_1*)

Prerequisites

-
- High power to the robot must be enabled.
- The robot must be homed.
- The robot must be **Attached** by the thread.

Parameters

location_1

A required **Location Object** or an expression that evaluates to a **Location Object** value. Can be either a Cartesian or an Angles type value.

location_2

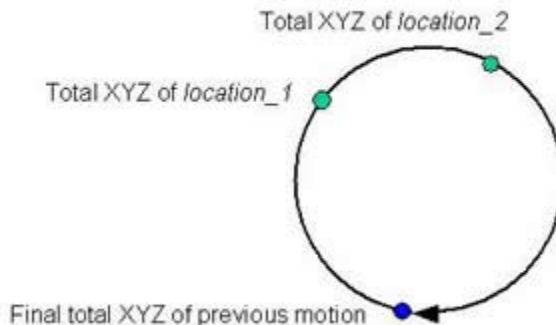
A required **Location Object** or an expression that evaluates to a **Location Object** value. Can be either a Cartesian or an Angles type value.

profile_1

A required **Profile Object** or an expression that evaluates to a **Profile Object** value. The **Straight** property that specifies a Cartesian straight-line or a joint interpolated motion is ignored since the motion is always performed in Cartesian coordinates.

Remarks

This method simultaneously moves all of the axes of a robot in a coordinated, position controlled motion such that the robot's tool tip follows an arc path around a complete circle. The circle is defined by the XYZ values of the final position of the previous motion and *location_1* and *location_2*. The performance parameters for the motion are defined by the **Profile Object**, *profile_1* (e.g. **Speed**, **Accel**, **Decel**, **AccelRamp**, **DecelRamp**).



The circle begins at the final "total XYZ position" of the previous motion, goes through the "total" XYZ position of *location_1* and the "total" XYZ position of *location_2* and terminates at the starting position. The "total positions" of *location_1* and *location_2* are computed as the results of evaluating each **Location's PosWrtRef** value relative to the "total position" of their respective reference frames, if any. If a **Location** is specified as an **Angles** type, its XYZ position is computed using the kinematic model for the attached robot.

If *profile_1* has its **InRange** property set to zero or a positive value, the system will bring the robot to a stop at the final position. If this property is negative and the next motion statement is executed before this motion reaches its destination, GPL will attempt to blend the two motions together into a "continuous path". Circular interpolated motions can be blended with any of the motion types, i.e. Cartesian straight-line, joint interpolated or other circular interpolated motions.

If the previous motion is still in process when the **Move.Circle** instruction is executed, the **Move.Circle** instruction will temporarily suspend execution of its thread. At the conclusion of the previous motion or as soon as the new **Circle** motion starts to be blended with the previous motion, the thread will automatically continue execution at the next instruction in the GPL procedure.

The following are special notes regarding the use of the **Circle** method.

-
- The circle can be defined in any arbitrary orientation and need not lie in an cardinal plane.
- The XYZ values of *location_1* and *location_2* need not be equal distance between the starting and ending positions of the circle although values closer to 120 degrees apart will increase the accuracy of the plane of the circle.
- If the three XYZ points that define the circle lie in a straight-line, the **Circle** method motion is automatically converted to a short move to nowhere.
- When blending a **Circle** motion with another motion, the s-curve **AccelRamp** and **DecelRamp** should be set to 0 and the **Accel** and **Decel** properties should be set high to ensure that the path tracks the circular path as closely as possible.
- During the circular motion, the orientation of the tool is held constant.

Examples

```
Dim p0 As New Location      ' Create location objects
Dim p1 As New Location
```

```
Dim p2 As New Location

p0.XYZ(100,200,-100,0,180,0) ' Center on (200,200), radius 100
p1.XYZ(200,300,-100,0,180,0)
p2.XYZ(200,100,-100,0,180,0)

Move.Loc(p0,pf_start)      ' Move to start position
Move.Circle(p1,p2,pf_on_path) ' Move in a circle
Move.WaitForEOM           ' Pause thread until motion done
```

See Also

[Location Class](#) | [Move Class](#) | [Move.Arc](#) | [Move.Loc](#) | [Profile Class](#)

Move.Delay Method

Pauses execution of a robot's motions for a specified period of time, in seconds.

Move.Delay (*seconds*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be homed.
- The robot must be **Attached** by the thread.

Parameters

seconds

A required numeric expression that specifies the number of seconds to delay any further robot motions, interpreted as a **Double** value.

Remarks

This method delays any further motions for the attached robot for the specified number of seconds. This delay starts immediately if the robot is not moving or starts at the completion of any in-process motions if the robot is moving. Unlike other methods that simply suspend execution of a thread, this delay is synchronized with the movement of the robot. So, it is very useful of inserting process delays in order to allow other equipment to complete their operations before the robot moves to its next step. For example, this method can be used after the robot has come to a complete halt to pick up a part, to insert a fixed delay to allow the robot's gripper to close and engage the part.

Another advantage of this method is that it is implemented like a command to "move to the current position for a fixed amount of time". This means that as soon as the delay period begins, execution of the thread continues. This allows the thread to monitor other activities or plan the next motion. Also, since the **Delay** method behaves like any other motion, the **Delay** can be prematurely terminated by a RapidDecel command.

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
Move.Loc(loc1, prof1)        ' Move to global loc1
Move.Delay(0.2)               ' Delay for .2 seconds after we reach loc1
```

See Also

[Move Class](#) | [Move.WaitForEOM](#)

Move.Extra Method

Move extra, independent axes during the next motion to a **Cartesian Location**.

Move.Extra (*axis_1_position*, *axis_2_position*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be **Attached** by the thread.

Parameters

axis_1_position

A required numeric expression that specifies the new position of the first extra axis as an absolute position in units of either millimeters or degrees as appropriate.

axis_2_position

An optional numeric expression that specifies the new position of the second extra axis as an absolute position in units of either millimeters or degrees as appropriate.

Remarks

Selected kinematic modules include extra, independent axes that are physically part of the robot but that do not logically factor into the calculation of the Cartesian position and orientation of the robot. For example, the "Dual RPR Robot" and the "XYZ Plus Extra Axis Robot" both include an extra axis that does not affect the Cartesian location of the robot.

For these types of robots, if a motion instruction is executed to a **Cartesian Location** value, there is no information available to define where the extra axis is to be moved. So, in general, the extra axis will remain in its current position during such a motion.

To address this need, the **Move.Extra** method can be executed prior to the execution of a motion to a **Cartesian Location**. During the motion, any extra axes will be moved to the positions specified by the **Move.Extra** method simultaneously with the other axes of the robot. If the next motion is not to a **Cartesian Location**, the information specified in the **Move.Extra** method is ignored.

As an alternative to using the **Move.Extra** method, a motion specified to an **Angles Location** will move all of the axes of the robot including the extra axis. However, in this case, the benefits of utilizing a **Cartesian Location** will be lost.

Please see the documentation for your specific Robot Kinematic Module to determine if this instruction has any affect.

Examples

```
Dim pf1 As New Profile      ' Create new profile set to default values
Move.Extra(20)              ' Move extra axis to 20 during next motion
Move.Loc(Location.XYZValue(300,0,100),pf1) ' Move robot and extra axis
```

See Also

[Move Class](#) | [Move.Loc](#) | [Move.Rel](#)

Move.ForceOverlap Method

Bypasses the system's normal motion blending features and explicitly defines how the execution of two sequential motions are to be overlapped.

Move.ForceOverlap (*mode*, *criterion*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be **Attached** by the thread.

Parameters

mode

A required arithmetic expression that defines how the overlapping is specified and the *criterion* is interpreted. Currently, **this value must be 0** and specifies that the *criterion* is a percentage of overlap.

criterion

A required arithmetic expression that defines how much the next motion is to be overlapped with the currently executing motion. The interpretation of this parameter is a function of the *mode*.

Remarks

In most applications, the system automatically optimizes the execution of sequential motions by blending (overlapping) the deceleration of the previous motion with the acceleration of the next motion. For example, if a motion in the X direction is split into two separate motion instructions and the robot is instructed not to stop between the motions, the system will automatically blend the deceleration of the first segment with the acceleration of the second segment such that the two motions will appear as though they were a single continuous motion. This blending can significantly improve the performance of a robot since the time required for accelerating and decelerating adversely effects cycle time.

When the system automatically computes the amount by which sequential motions are blended, it takes into account the maximum allowable acceleration and deceleration of the robot. This permits the cycle time to be optimized without exceeding the capabilities of the mechanical system.

However, in some special cases, it is desirable to override the system's standard blending computations by using the **ForceOverlap** method to explicitly define how much two motions are to be overlapped.

This method has the following benefits as compared to automatic blending:

- Allows all segments of the current motion to be overlapped with the next motion, not just the current motion's deceleration and the next motion's acceleration segments. This permits a much greater overlapping of the two motions.
- Provides explicit overlapping specification in cases where the automatic blending may not result in optimal performance.

This method has the following disadvantages as compared to automatic blending:

- No checking is performed to ensure that the maximum acceleration and deceleration capabilities of the robot are not exceeded.
- The system's blending algorithms automatically reduce the deceleration of the current motion and the acceleration of the next motion when this will not adversely effect cycle time to increase the smoothness of the motion transition.
- The **ForceOverlap** method places more burden on the application programmer for optimizing motion cycle time.

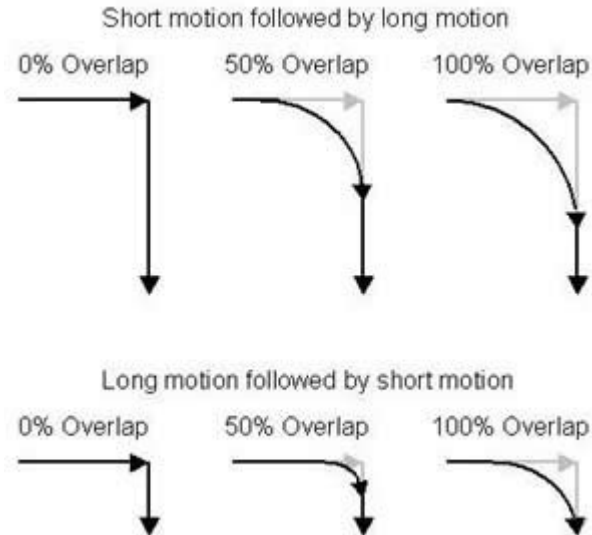
The interpretation of the *mode* and *criterion* parameters are described in the following table.

<i>mode</i>	<i>criterion</i>	Resulting Overlap
0	% (0-100)	% of the total execution time of the next motion that is to be overlapped with the currently executing motion. A value of 0 indicates that the two motions are not overlapped. A value of 100 indicates that all of the next motion is to be overlapped with the currently executing motion if possible.

The motion overlap generated by this method is subject to the following limitations.

- Since the overlap is with respect to the currently executing motion, the next motion will never be started prior to the execution of the current motion.
- The overlap is limited to ensure that the next motion never terminates before the end of the currently executing motion.
- If the current motion is defined to stop, i.e. has a **Profile Inrange** parameter of 0 or greater than 0, no overlapping will be performed.

The following simplified drawings graphically illustrate how the overlapping is performed. In the first set of drawings, the current motion is shorter than the next motion. In the second set of drawings, the current motion is longer than the next motion.



Note that when the next motion is longer than the current motion, the overlap can be extended to almost the start of the current motion. If the next motion is shorter than the current motion, the next motion will always be started sufficiently after the start of the current motion to ensure that the next motion does not terminate before the current motion.

By comparison, the following picture illustrates the amount of overlapping that can be expected as a result of the system's automatic blending algorithm. The automatic blending is very easy to use and ensures that the robot's dynamic capabilities are not exceeded. However, the overlapping is generally limited to the deceleration segment of the previous motion and the acceleration segment of the next motion.

Standard automatic blending



Examples

```
Dim pf1 As New Profile
Robot.Attached = 1           ' Get control of robot #1
pf1.Inrange = -1            ' Don't stop at end of motion
Move.Rel(Location.XYZValue(10), pf1) ' Move 10 mm in X-tool coordinates
Move.ForceOverlap(0, 50)      ' Overlap 50% of the next motion's time
Move.Rel(Location.XYZValue(0,10), pf1) ' Move 10 mm in Y-tool coordinates
Robot.Attached = 0           ' Release control of robot
```

See Also

[Move Class](#)

Move.Loc Method

Basic method for moving the robot to a specified destination in a position-controlled motion.

Move.Loc (*location_1*, *profile_1*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be homed.
- The robot must be **Attached** by the thread.

Parameters

location_1

A required **Location Object** or an expression that evaluates to a **Location Object** value. Can be either a Cartesian or an Angles type value.

profile_1

A required **Profile Object** or an expression that evaluates to a **Profile Object** value. Can specify either Cartesian straight-line or joint interpolated motions.

Remarks

This is the basic method for simultaneously moving all of the axes of a robot in a coordinated, position controlled motion to a destination specified by a **Location Object**, *location_1*, using performance parameters defined by a **Profile Object**, *profile_1* (e.g. **Speed**, **Accel**, **Decel**, **AccelRamp**, **DecelRamp**).

The destination of the motion will be the “total position” defined by *location_1*. For the various forms for the **Location Object**, the motion destination will be computed as follows:

- If *location_1* is a **Cartesian Location** with a reference frame, the “total position” is computed as the position and orientation that is a result of evaluating *location_1*’s **PosWrtRef** value relative to the “total position” of the reference frame.
- If *location_1* is a **Cartesian Location** without a reference frame, *location_1*’s **PosWrtRef** value is interpreted as the absolute coordinates for the destination.
- Otherwise, *location_1* is an **Angles Location** and the motion destination will be the axes positions specified by *location_1*.

If *profile_1* specifies a **Straight**-line motion, the robot will move along a straight path in Cartesian space. Otherwise, a joint-interpolated motion will be generated. If *profile_1* has

its **InRange** property set to zero or a positive value, the system will bring the robot to a stop at *location_1*. If this property is negative and the next motion statement is executed before this motion reaches its destination, GPL will attempt to blend the two motions together into a “continuous path”.

If the previous motion is still in process when the **Move.Loc** instruction is executed, the **Move.Loc** instruction will temporarily suspend execution of its thread. At the conclusion of the previous motion or as soon as the new motion starts to be blended with the previous motion, the thread will automatically continue execution at the next instruction in the GPL procedure.

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
Dim loc1 As New Location      ' Create new location value
loc1.XYZ(10,20,30,0,180,20)   ' Define position to move to
Move.Loc(loc1, prof1)         ' Move to loc1 using prof1
```

See Also

[Location Class](#) | [Move Class](#) | [Move.Approach](#) | [Move.Arc](#) | [Move.Extra](#) | [Move.Rel](#) | [Profile Class](#)

Move.OneAxis Method

Convenience method to move a single axis of a robot.

Move.OneAxis (*axis*, *axis_position*, *relative_flag*, *profile_1*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be homed.
- The robot must be **Attached** by the thread.

Parameters

axis

A required numeric expression that specifies the number of the robot's axis that is to be moved, 1-n.

axis_position

A required numeric expression that specifies the new position of the axis as either an absolute position or a relative position, in units of either millimeters or degrees as appropriate.

relative_flag

A required numeric expression that is interpreted as a **Boolean** that indicates if the *axis_position* is an absolute axis position (**False**) or a relative value (**True**).

profile_1

A required **Profile Object** or an expression that evaluates to a **Profile Object** value. Can specify either Cartesian straight-line or joint interpolated motions.

Remarks

This method is primarily a convenience and diagnostic function that moves a single axis of the **Attached** robot. If the *relative_flag* is **True**, the new axis position is computed by adding the *axis_position* value to the final axis position of the previous motion. Otherwise, the *axis_position* is taken as the new absolute position for the axis.

When this motion is generated, the positions of all of the other axes of the robot remain unchanged.

Once the **OneAxis** method computes the desired position for each axis, the motion execution is identical to the **Move.Loc** method except that **Straight**-line motions are not

permitted. However, the motion can be blended with the previous and the next motions as desired, and the performance parameters are defined by *profile_1* (e.g. **Speed**, **Accel**, **Decel**, **AccelRamp**, **DecelRamp**, **InRange**).

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
Move.OneAxis(1,20,True,prof1) ' Increment axis 1 by 20 mm or deg
```

See Also

[Move Class](#) | [Move.Loc](#) | [Move.Rel](#)

Move.Rel Method

Moves the robot in a position-controlled motion to a **Location** that is relative to the final position and orientation of the previous motion.

Move.Rel (*location_1*, *profile_1*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be homed.
- The robot must be **Attached** by the thread.

Parameters

location_1

A required **Location Object** or an expression that evaluates to a **Location Object** value. Can be either a Cartesian or an Angles type value.

profile_1

A required **Profile Object** or an expression that evaluates to a **Profile Object** value. Can specify either Cartesian straight-line or joint interpolated motions.

Remarks

This method simultaneously moves all of the axes of the robot in a coordinated, position controlled motion to a destination specified by the “total position” of *location_1*, which is interpreted as an incremental change relative to the final position and orientation of the previous motion. If *location_1* is a **Cartesian Location**, the “total position” of *location_1* is evaluated relative to the final Cartesian position and orientation of the previous motion. If *location_1* is a **Angles Location**, the motion’s destination is computed by adding *location_1*’s set of angles to the final angles of the previous motion.

Note, that this motion is relative to the actual final position and orientation of the previous motion and not the planned destination of the previous motion (**Robot.Dest**, **Robot.DestAngles**). The planned destination remains the same even if the motion prematurely terminates execution. This was designed to allow a motion to be retried. However, the actual final position and orientation is modified by a Soft E-Stop, a Hard E-Stop, a RapidDecel command or other conditions. So, the **Rel** method is designed to allow a program to do an incremental motion from wherever the robot actually stopped.

For **Cartesian Locations**, it should be keep in mind that the incremental motion is performed in the tool coordinate system of the robot. For example, a positive incremental Z motion will not necessarily move up vertically in the world coordinate system. It will move along the Z-axis of the robot’s end effector.

Once the **Rel** method computes the desired motion destination, the motion execution is identical to the **Move.Loc** method. The motion can be a **Straight**-line or joint interpolated motion, can be blended with the previous and the next motions as desired, and the performance parameters are defined by *profile_1* (e.g. **Speed**, **Accel**, **Decel**, **AccelRamp**, **DecelRamp**, **InRange**).

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
Dim loc1 As New Location     ' Create new location value
loc1.XYZ(10,20,30,0,180,20) ' Define position to move to
Move.Loc(loc1, prof1)        ' Move to loc1 using prof1
loc1.XYZ(10)                  ' Define incremental motion in X
Move.Rel(loc1, prof1)         ' Move 10 mm in Tool X, not World
```

See Also

[Location Class](#) | [Move Class](#) | [Move.Approach](#) | [Move.Extra](#) | [Move.Loc](#) | [Profile Class](#)

Move.SetJogCommand Method

Sets or changes the specific mode, axis and speed during jog (manual) control mode.

Move.SetJogCommand (*jog_mode, jog_axis, jog_speed*)

Prerequisites

- High power to the robot must be enabled.
- The robot *does not* need to be homed.
- The robot must be **Attached** by the thread.
- The robot must be in jog control mode.

Parameters

jog_mode

A required expression that evaluates to an **Integer** value. This value specifies the manual control mode that should now be in effect.

jog_axis

A required expression that evaluates to an **Integer** value. This defines the robot or Cartesian axis that is to be moved under manual control.

jog_speed

A required expression that evaluates to a percentage value between +100 and -100. This specifies the target speed and direction for the manual control motion. The system automatically generates a motion profile to accelerate up to this speed and to decelerate to a stop after the manual motion is completed.

Remarks

After a robot has been placed into jog (manual) control mode, this method must be executed to define the manual control mode, the axis to be manually controlled and the speed at which the axis is to be moved. This method can be executed at any time during jog control mode and as many times as desired. It simply posts the parameters to the trajectory generator for execution. The trajectory generator automatically takes care of smoothly transitioning between modes and target speeds.

For example, if the robot is being moved in World manual control mode and a new command to move in joint manual mode is received, the trajectory generator will decelerate the World manual mode motion to a stop prior to starting the acceleration up to the target joint manual mode speed. As another example, if the robot is being moved in any mode and a new command is posted that changes the target speed, the trajectory generator will smoothly accelerate or decelerate to achieve the new speed.

The interpretation of the parameters to this method are as follows:

Jog_Mode	Jog_Axis	Jog_Speed	Description
0	Ignored.	Ignored.	Idle , robot not moving.
1	Robot joint number, 1-n	Joint speed and direction.	Joint manual control mode. A single robot axis can be moved. The robot does not need to be homed. Axes that are out-of-range can be moved into range.
2	Cartesian axis: 1:X, 2:Y, 3:Z, 4:RX, 5:RY, 6:RZ	Cartesian speed and direction.	World manual control mode. Translates or rotates along or about a single world (base) Cartesian coordinate axis. The robot must be homed.
3	Cartesian axis: 1:X, 2:Y, 3:Z, 4:RX, 5:RY, 6:RZ	Cartesian speed and direction.	Tool manual control mode. Translates or rotates along or about a single tool (gripper) Cartesian coordinate axis. The robot must be homed.
4	Robot joint number, 1-n	Positive values free the joint and negative values lock the joint.	Free manual control mode. Puts any number of axes into torque control mode to permit the axes to be manually pushed into position.

For Joint, World and Tool control modes, if the magnitude of the speed is set to 5% or less, the robot will move a discrete increment and then stop rather than move continuously. In order to move an additional small increment, the speed must be set to 0 and then to a value of 5% or less. This is very convenient for fine positioning the robot.



WARNING: Any axis commanded to move at greater than 5% speed will continue to do so until stopped. It is responsibility of the GPL Project to have suitable safe guards and time outs to ensure that a motion is terminated when required.

Examples

```
Robot.Attached = 1           ' Get control of robot #1
Move.StartJogMode()          ' Initiate jog control mode
Move.SetJogCommand(3, 3, 50) ' Set tool mode, Z-axis, 50% speed
Thread.Sleep(4000)
Move.SetJogCommand(2, 1, -50) ' Change to world mode, X-axis, -50% speed
Thread.Sleep(4000)
Move.StopSpecialModes        ' Terminate jog mode
Robot.Attached = 0           ' Release control of robot
```

See Also

[Move Class](#) | [Move.StartJogMode](#) | [Move.StopSpecialModes](#)

Move.SetSpeeds Method

Sets new target speeds for all axes of a robot in velocity control mode.

Move.SetSpeeds (*speed_array*, *profile_1*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be homed.
- The robot must be **Attached** by the thread.
- The robot must be in velocity control mode.

Parameters

speeds_array

A required array of **Doubles** that contains a speed specification for each axis of the robot. The first array element (0) corresponds to the target speed for the robot's first axis. One value must be provided for each axis of the robot. Each array element is interpreted in units of mm/sec (linear axes) or deg/sec (rotary axes). These values are limited by the maximum permitted joint speed, e.g. 100%_joint_speeds * max_%_speed_allowed.

profile_1

An optional **Profile Object** or an expression that evaluates to a **Profile Object** value. This value defines the acceleration, deceleration and acceleration/deceleration ramp times to be use to change the speed of each axes. In certain cases, it may not be possible to honor the ramp times without over-shooting the target velocity, but the acceleration and deceleration limits are adhered to. For example, this can occur if an axis is accelerating to a high velocity and suddenly a new, lower velocity target is specified. If this parameter is not specified, the **Profile** specified by the last executed **Move.SetSpeeds** or **Move.StartVelocityCtrl** method will be utilized.

Remarks

After a robot has been placed into velocity control mode, this method can be used to modify the target speed levels for each axis. This method can be executed at any time and as many times as desired. It simply posts the desired target speeds to the trajectory generator. The next time that the trajectory generator executes, the specified speeds will be taken as the new target values. If this method is executed multiple times before the trajectory generator executes again, only the last values posted will have an effect.

Examples

```

Dim speeds(12) As Double           ' All Double speeds will be 0
Dim pfl As New Profile             ' Use default accel/decel
Dim ii As Integer
Robot.Attached = 1                 ' Get control of robot #1
Move.StartVelocityCtrl(0, 0, speeds, pfl) ' Set to velocity control mode
For ii = 36 To 360 Step 36
    speeds(0) = ii                 ' New speed value
    Move.SetSpeeds(speeds)         ' Ramp axis 1 speed
    Controller.SleepTick(30)       ' Wait a little while
Next ii
Move.StopSpecialModes              ' Terminate velocity mode
Robot.Attached = 0                 ' Release control of robot

```

See Also

[Move Class](#) | [Move.StartVelocityCtrl](#) | [Move.StopSpecialModes](#)

Move.SetTorques Method

Sets new target torque output levels for all motors in torque control mode.

Move.SetTorques (*torques_array*)

Prerequisites

- High power to the robot must be enabled.
- The robot *does not* need to be homed.
- The robot must be **Attached** by the thread.
- One or more motors of the robot must be operating in torque control mode.

Parameters

torques_array

A required array of **Doubles** that contains a torque specification for each motor of the robot. The first array element (0) corresponds to the torque value for the robot's first motor. Array elements for motors that are not torque controlled are ignored. Each array element is interpreted as a percentage, where a value of +100 or -100 indicates that the torque output should be equivalent to the full positive or negative rated motor torque. Since the peak motor torque can usually be higher than the rated torque, values greater than +/- 100% are permitted.

Remarks

After selected motors of a robot have been placed into torque control mode, this method can be used to modify the target torque levels. This method can be executed at any time and as many times as desired. It simply posts the desired torque levels to the trajectory generator. The next time that the trajectory generator executes, the specified torque levels will be taken as the new target values. If this method is executed multiple times before the trajectory generator executes again, only the last values posted will have an effect.

Examples

```
Dim torques(12) As Double          ' All Double torques will be 0
Dim ii, jj As Integer

Robot.Attached = 1                  ' Get control of robot #1
Move.StartTorqueCtrl(1, 0, torques) ' Set motor 1 to torque mode
For jj = 1 To 10
    For ii = 0 To 100
        Controller.SleepTick()      ' Wait till next trajectory cycle
        torques(0) = ii/10          ' New torque value
        Move.SetTorques(torques)    ' Ramp torque from 0% to 10%
    Next ii
Next jj
Move.StopSpecialModes              ' Terminate torque mode
```

```
Robot.Attached = 0           ' Release control of robot
```

See Also

[Move Class](#) | [Move.StartTorqueCntrl](#) | [Move.StopSpecialModes](#)

Move.StartJogMode Method

Initiates execution of jog (manual) control mode.

Move.StartJogMode ()

Prerequisites

- High power to the robot must be enabled.
- The robot does not need to be homed.
- The robot must be **Attached** by the thread.
- This mode is not compatible with torque, velocity or other special control modes.

Parameters

None

Remarks

This method switches all of the axes of a robot from the standard position controlled mode to jog (manual) control mode. This is the mode that is utilized by the Virtual and Hardware Manual Control Pendants (MCP) to implement joint, world, tool and free manual control modes. This method and the **Move.SetJogCommand** method are provided to permit these same manual modes to be easily implemented by a GPL Project. For example, these methods can be used by a GPL program to implement manual control modes via a graphics HMI or a joystick.

When a robot is placed into this mode, it is moved in a manner similar to velocity control mode in that a specified axis or group of axes are accelerated and moved at a specified continuous speed until they are instructed to change their speed.



WARNING: Any axis commanded to move will continue to do so until stopped. So, it is responsibility of the GPL Project to have suitable safe guards and time outs to ensure that a motion is terminated when required.

When this method is executed, it first waits for any in-process position controlled motions to be completed. It then transitions all axes into jog control mode. Once in this mode, the **Move.SetJogCommand** method must be executed to set and change the specific manual mode, axis and motion speed.

When an axis speed is specified, the setting of the "System Test Speed" is ignored to permit the robot to be moved in a consistent manner when debugging applications.

To permit the axes of a robot to be moved back into range if they are accidentally moved beyond their stop limits, joint control mode permits out-of-range axes to be moved back in range, but not further out-of-range. In addition, the robot does not need to be homed in order to move the axes in joint control mode to permit it to be manually repositioned.

The robot will remain in jog control mode until one of the following occurs:

1. The **Move.StopSpecialModes** method is executed to terminate this mode.
2. A hardware error or hard E-stop or soft E-stop occurs.
3. A **RapidDecel** is issued.
4. The robot is detached by the user program either by issuing a detach command or by halting user program execution for any reason.

Examples

```
Robot.Attached = 1           ' Get control of robot #1
Move.StartJogMode()          ' Initiate jog control mode
Move.SetJogCommand(3, 3, 50) ' Set tool mode, Z-axis, 50% speed
Thread.Sleep(4000)
Move.SetJogCommand(2, 1, -50) ' Change to world mode, X-axis, -50% speed
Thread.Sleep(4000)
Move.StopSpecialModes         ' Terminate jog mode
Robot.Attached = 0           ' Release control of robot
```

See Also

[Move Class](#) | [Move.SetJogCommand](#) | [Move.StopSpecialModes](#)

Move.StartTorqueCntrl Method

Initiates execution of torque control mode for one or more motors.

Move.StartTorqueCntrl (*motor_mask*, *adc_mask*, *torques_array*)

Prerequisites

- High power to the robot must be enabled.
- The robot *does not* need to be homed.
- The robot must be **Attached** by the thread.

Parameters

motor_mask

A required numeric expression that evaluates to a bit mask that specifies the motors to be placed into torque control mode. The least significant bit corresponds to the first motor for the attached robot.

adc_mask

A required numeric expression that evaluates to a bit mask that specifies the single motor whose torque is to be directly controlled by the first ADC input channel. This value should be zero if no motor is to be ADC controlled. A scaled ADC reading of +1.0 or −1.0 will drive the corresponding motor at its full positive or negative rated motor torque. Since the peak motor torque can usually be higher than the rated torque, ADC values greater than +- 1.0 are permitted.

torques_array

A required array of **Doubles** that contains a torque specification for each motor of the robot. The first array element (0) corresponds to the torque value for the robot's first motor. Array elements for motors that are not torque controlled are ignored. Each array element is interpreted as a percentage, where a value of +100 or −100 indicates that the torque output should be equivalent to the full positive or negative rated motor torque. Since the peak motor torque can usually be higher than the rated torque, values greater than +- 100% are permitted.

Remarks

This method places the specified motors into torque control. Motors that are not placed into torque control mode continue to operate in position control mode and can be moved by the standard Move Class Methods. Thus, some axes of the robot can continue to follow a position-controlled path while others can exert a force or can move freely if their torque output is set to zero.

If a motor is specified in the *adc_mask*, that motor's torque output level is the sum of the percentage of rated motor torque specified in the *torques_array* and the value defined by the ADC input.

When this method is executed, it first waits for any in-process motions to be completed. It then transitions the specified motors into torque control and sets their initial torque levels to the values specified in the *torques_array*. The torque levels can subsequently be changed by executing a **Move.SetTorques** method or by a change in the ADC signal.

Since torque control does not close the position loop around a motor, the torque applied is unaffected by the current setting of the "System Test Speed". This is the speed value that can be set via the web Operator Control Panel or the "System wide test speed in %" (DataID 601) database parameter.

The specified motors will remain in torque control mode until one of the following occurs:

1. The **Move.StopSpecialModes** method is executed to terminate torque control mode for all motors.
2. A hardware error or hard E-stop or soft E-stop occurs.
3. A **RapidDecel** is issued.
4. The robot is detached by the user program either by issuing a detach command or by halting user program execution for any reason.

Torque control mode is compatible with both position and velocity control modes. However, torque control mode can only be initiated when in position control mode.

Examples

```
Dim torques(12) As Double      ' All Double torques will be 0
Dim ii, jj As Integer
Robot.Attached = 1            ' Get control of robot #1
Move.StartTorqueCntrl(1, 0, torques) ' Set motor 1 to torque mode
For jj = 1 To 10
    For ii = 0 To 100
        Controller.SleepTick() ' Wait till next trajectory cycle
        torques(0) = ii/10     ' New torque value
        Move.SetTorques(torques) ' Ramp torque from 0% to 10%
    Next ii
Next jj
Move.StopSpecialModes         ' Terminate torque mode
Robot.Attached = 0            ' Release control of robot
```

See Also

[Move Class](#) | [Move.SetTorques](#) | [Move.StopSpecialModes](#)

Move.StartVelocityCntrl Method

Switches all axes of a robot from position to velocity control mode.

Move.StartVelocityCntrl (*mode*, *adc_mask*, *speeds_array*, *profile_1*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be homed.
- The robot must be **Attached** by the thread.

Parameters

mode

A required numeric expression that evaluates to the mode of velocity control to be executed. Currently, this parameter is unused and should be set to 0 for compatibility with future software releases.

adc_mask

A required numeric expression that evaluates to a bit mask that specifies the single axis whose speed is to be directly controlled by the first ADC input channel. This value should be zero if no axis is to be ADC controlled. A scaled ADC reading of +1.0 or -1.0 will drive the corresponding axis at its full 100% speed.

speeds_array

A required array of **Doubles** that contains a speed specification for each axis of the robot. The first array element (0) corresponds to the target speed for the robot's first axis. One value must be provided for each axis of the robot. Each array element is interpreted in units of mm/sec (linear axes) or deg/sec (rotary axes). These values are limited by the maximum permitted joint speed, e.g. `100%_joint_speeds * max_%_speed_allowed`.

profile_1

A required **Profile Object** or an expression that evaluates to a **Profile Object** value. This value defines the acceleration, deceleration and acceleration/deceleration ramp times to be use to change the speed of each axes. In certain cases, it may not be possible to honor the ramp times without over-shooting the target velocity, but the acceleration and deceleration limits are adhered to. For example, this can occur if an axis is accelerating to a high velocity and suddenly a new, lower velocity target is specified.

Remarks

This method switches all of the axes of a robot from the standard position controlled mode to velocity controlled mode. When in velocity controlled mode, each axis accepts a target speed as its command rather than a position. The target speeds can be set by this method or can be updated at any time using the **Move.SetSpeeds** method. Once each axis has accelerated, it will continue to rotate at its target speed until the speed is explicitly changed, velocity control mode is terminated or an error occurs.

As with position control mode, velocity control mode is compatible with torque control mode. That is, when in velocity control mode, one or more motors can be in torque control mode. (Note: Motors must be placed into torque control mode when the robot is in position control mode. After motors are placed into torque control, the position-controlled joints can then be switched to velocity control mode.)

If an axis is specified in the *adc_mask*, that axis' target speed is the sum of the appropriate value in the *speeds_array* plus the value defined by the ADC input.

When this method is executed, it first waits for any in-process position controlled motions to be completed. It then transitions all axes into velocity control mode and sets the initial target speeds to the values specified in the *speeds_array*. The speed targets can subsequently be changed by executing a **Move.SetSpeeds** method or by a change in the ADC signal.

As a convenience in debugging applications, the velocity control target speed is affected by the current setting of the "System Test Speed". This is the speed value that can be set via the web Operator Control Panel or the "System wide test speed in %" (DataID 601) database parameter. In addition, software and hardware limit stop checking is still performed during this mode of operation. If an axis is to be rotated continuously, motors can be configured for continuous turn capability assuming that this capability is supported by the robot's kinematic module.

The robot will remain in velocity control mode until one of the following occurs:

1. The **Move.StopSpecialModes** method is executed to terminate velocity control mode.
2. A hardware error or hard E-stop or soft E-stop occurs.
3. A **RapidDecel** is issued.
4. The robot is detached by the user program either by issuing a detach command or by halting user program execution for any reason.

Examples

```
Dim speeds(12) As Double          ' All Double speeds will be 0
Dim pfl As New Profile           ' Use default accel/decel
Dim ii As Integer
Robot.Attached = 1               ' Get control of robot #1
Move.StartVelocityCtrl(0, 0, speeds, pfl) ' Set to velocity control mode
For ii = 36 To 360 Step 36
    speeds(0) = ii               ' New speed value
    Move.SetSpeeds(speeds)       ' Ramp axis 1 speed
    Controller.SleepTick(30)     ' Wait a little while
Next ii
Move.StopSpecialModes            ' Terminate velocity mode
Robot.Attached = 0              ' Release control of robot
```

See Also

[Move Class](#) | [Move.SetSpeeds](#) | [Move.StopSpecialModes](#) | [Move.StartTorqueCntrl](#)

Move.StopSpecialModes Method

Terminates execution of any active special trajectory control modes.

Move.StopSpecialModes

Prerequisites

- High power to the robot must be enabled.
- The robot must be **Attached** by the thread.

Parameters

None

Remarks

If any special, i.e. non-position control, trajectory modes are in effect, this method executes the equivalent of a **Robot.RapidDecel** to immediately decelerate any moving axes of the attached robot to a stop. At the completion of this operation, all special trajectory generation modes will be terminated and the robot will be in the standard position control mode. If no special modes are in effect, this method performs no operation and does not signal an error.

In particular, the following modes of execution will be terminated:

- External trajectory control mode
- Jog (manual) control mode
- Master/slave mode
- Torque control mode
- Velocity control mode

Examples

```
Move.StopSpecialModes      ' Halts any special control modes in effect
```

See Also

[Move Class](#) | [Move.StartJogMode](#) | [Move.StartTorqueCntrl](#) | [Move.StartVelocityCntrl](#) | [Robot.RapidDecel](#)

Move.Trigger Method

Primes the system to automatically assert a digital output signal at a prescribed trigger position during the next motion.

Move.Trigger (*mode*, *trigger_pt*, *channel*)

Prerequisites

- High power to the robot must be enabled.
- The robot must be **Attached** by the thread.

Parameters

mode

A required arithmetic expression that defines the manner in which the trigger position is defined.

trigger_pt

A required arithmetic expression that defines the trigger position. The interpretation of this value is a function of the *mode*.

channel

A required arithmetic expression that specifies the digital I/O channel whose output is set at the trigger point. If the channel number is positive, the output is turned ON at the trigger point. If the channel number is negative, the output is turned OFF at the trigger point. If the value is 0, any previous **Move.Trigger** operation is disabled.

Remarks

After this instruction is executed, the digital output signal defined by the *channel* will be asserted when the next motion reaches a specified trigger position. The trigger position is defined by the *mode* and the *trigger_pt* values as described in the following table:

<i>mode</i>	<i>trigger_pt</i>	Resulting Trigger Point
0	% (0-100)	% of change in position of the motion measured from the start of the motion.
1	% (0-100)	% of change in position of the motion measured from the end of the motion.
2	mm	Distance in millimeters from the start of the motion. Only valid for straight-line and arc motions.
3	mm	Distance in millimeters before the end of the motion. Only valid for straight-line and arc motions.
4	seconds	Time after the start of the motion.

5	seconds	Time before the end of the motion.
---	---------	------------------------------------

For example, if the *mode* is "1" and the *trigger_pt* is "10", if the next motion is joint interpolated, the *channel* signal will be asserted when the joints are 90% of the way to their final values.

For *modes* 4 & 5, the trigger point is computed assuming that the system is operating with the System Speed (as set via the Operator Control Panel) at a value of 100%. If the System Speed is set to 50%, the motion time is doubled and the effective trigger point time is doubled as well. To set the time value to be independent of the System Speed, the *trigger_pt* value should be adjusted by the value of the "System wide test speed in %" (DataID 601).

If the next motion is blended with the subsequent motion and a *mode* is selected that is relative to the end of the next motion, the trigger point will be relative to the end of the blending period. Since the start and end of the blending period are a function of both the next and the subsequent motion, the trigger point will vary as a function of both motions. Likewise, if the next motion is blended with the previous motion, trigger points defined relative to the start of the next motion will vary as a function of the motion blending.

If a motion terminates in the standard manner, the *channel* is guaranteed to be asserted at some point during the motion. However, if an error or **RapidDecel** function prematurely terminates a motion, the digital output signal may not be asserted.

Examples

```
Dim pfl As New Profile          ' Use default accel/decel
Robot.Attached = 1             ' Get control of robot #1
Signal.DIO(20001) = 0          ' Turn off signal
Move.Trigger(1, 10, 20001)     ' Turn on 90% into motion
Move.Rel(Location.XYZValue(10), pfl) ' Move 10 mm in tool coordinates
Robot.Attached = 0             ' Release control of robot
```

See Also

[Move Class](#)

Move.WaitForEOM Method

Suspends execution of the current thread until the robot completes its current motion.

Move.WaitForEOM

Prerequisites

- High power to the robot must be enabled.
- The robot must be **Attached** by the thread.

Parameters

None

Remarks

This allows a program that is controlling a robot (i.e. **Attached** to) to synchronizing its execution with the robot by suspending execution of the thread until any current robot motion has been completed. This method is valid for waiting until the completion of both position and velocity controlled motions.

Examples

```
Dim prof1 As New Profile ' Create new profile set to default values
Move.Loc(loc1, prof1)    ' Move to global loc1
Move.WaitForEOM          ' Execution suspended until robot at loc1
:                        ' Execution continues here after robot stops
```

See Also

[Move Class](#) | [Move.Approach](#) | [Move.Loc](#) | [Move.OneAxis](#) | [Move.Rel](#)

Networking Classes

Networking Classes Summary

The following pages provide detailed information on the properties and methods for the various classes that implement Ethernet networking communications.

The networking classes include: a **IPEndPoint Class** for specifying IP and port addresses; a **Socket Class** that is the basis for most networking I/O operations and contains the basic send and receive methods; a **TcpListener Class** that is used for implementing TCP server applications; a **TcpClient Class** for implementing TCP client applications; and finally a **UdpClient Class** for implementing both the server and client side of UDP based communications.

The tables below briefly summarize the properties and methods for each Class, which are described in greater detail in the following sections.

IPEndPoint Member	Type	Description
New IPEndPoint	Constructor Method	Creates an Endpoint and allows the IP Address and Port to be specified.
ipendpoint_obj.IPAddress	Property	Sets or gets the IP Address of an Endpoint.
ipendpoint_obj.Port	Property	Sets or gets the Port of an Endpoint.

Socket Member	Type	Description
socket_obj.Available	Property	Gets the number of data bytes currently available to receive from a Socket .
socket_obj.Blocking	Property	Sets or gets the blocking mode for a Socket . If True , the Socket blocks. If False , it does not block.
socket_obj.Close	Method	Closes any connections associated with a Socket .
socket_obj.Connect	Method	Requests a TCP Client connection with a remote TCP Server.
socket_obj.Receive	Method	Receives a datagram from an open TCP connection.
socket_obj.ReceiveFrom	Method	Receives a datagram from an open UDP connection.
socket_obj.ReceiveTimeout	Property	Sets or gets the receive timeout, in milliseconds, for a Socket .
socket_obj.Send	Method	Sends a datagram on an open TCP connection.
socket_obj.SendTimeout	Property	Sets or gets the send timeout, in milliseconds, for a Socket .
socket_obj.SendTo	Method	Sends a datagram to an open UDP connection.

TcpClient Member	Type	Description
New TcpClient	Constructor Method	Creates an Object for a TCP Client and optionally requests a connection.
tcpclient_obj.Client	Method	Returns the embedded Socket for performing I/O.
tcpclient_obj.Close	Method	Closes a Client Socket and breaks any connection.

TcpListener Member	Type	Description
New TcpListener	Constructor Method	Creates an Object for a TCP Server to listen for connections.
tcplistener_obj.AcceptSocket	Method	Accepts a connection and returns a new Socket Object for use by the TCP Server.
tcplistener_obj.Close	Method	Stops listening and closes the listener Socket .
tcplistener_obj.Pending	Property	True if there is a pending connection and AcceptSocket will succeed. Otherwise False .
tcplistener_obj.Start	Method	Starts listening for connection requests.
tcplistener_obj.Stop	Method	Stops listening and closes the listener Socket . Same as Close method.

UdpClient Member	Type	Description
New UdpClient	Constructor Method	Creates an Object for I/O using UDP.
udpclient_obj.Client	Method	Returns the embedded Socket for performing I/O.
udpclient_obj.Close	Method	Closes a Socket .

New IPEndPoint Constructor

Constructor for creating an IP endpoint object and optionally initializing it.

New IPEndPoint (*IP_address*, *port_number*)

Prerequisites

None

Parameters

IP_address

An optional string containing a standard IP address in the form "nnn.nnn.nnn.nnn". This address identifies a computer or computer-based device on the network. If omitted, or empty, the IP address is assumed to be a "wild card", matching any address.

port_number

An optional number specifying the port number, from 0 to 65536 of a process, protocol, or connection. If omitted, the port number is assigned automatically.

Remarks

The combination of IP address and port uniquely specifies a computer and process on a network. When messages are exchanged, both the sender and the receiver have an endpoint address consisting of these two items.

Examples

```
Dim ep As New IPEndPoint("192.168.0.2", 1234) ' Port 1234 at address 192.168.0.2
Dim ep As New IPEndPoint("", 69)           ' Port 69 at any address
```

See Also

[Networking Classes](#) | [ipendpoint_object.IPAddress](#) | [ipendpoint_object.Port](#)

ipendpoint_object.IPAddress Property

Sets or gets the IP address associated with an **IPEndPoint** object.

```
ipendpoint_object.IPAddress = <ip_address_string>
-or-
...ipendpoint_object.IPAddress
```

Prerequisites

None

Parameters

None

Remarks

The IP Address identifies a computer or computer-based device on the network. If empty, the IP address is assumed to be a “wild card”, matching any address.

This property converts the IP Address part of an **IPEndPoint Object** to or from a string value. The string value contains the address in the form nnn.nnn.nnn.nnn where each nnn field is a decimal number representing 8 bits of the 32-bit IP address.

Examples

```
Dim ep As New IPEndPoint()
ep.IPAddress = "192.168.0.2"      ' Assign the IP Address to the endpoint
Console.WriteLine(ep.IPAddress)  ' Display the IP Address of the endpoint
```

See Also

[Networking Classes](#) | [NewIPEndPoint](#) | [ipendpoint_object.Port](#)

ipendpoint_object.Port Property

Sets or gets the port number associated with an **IPEndPoint Object**.

```
ipendpoint_object.Port= <port_number>  
-or-  
...ipendpoint_object.Port
```

Prerequisites

None

Parameters

None

Remarks

The port number specifies a process, protocol, or connection at an endpoint. This number may range from 0 to 65536.

This property sets or gets the port number of an **IPEndPoint Object**.

Examples

```
Dim ep As New IPEndPoint()  
ep.Port = 1234           ' Set the port of an endpoint object  
Console.WriteLine(ep.Port) ' Display the port of the endpoint
```

See Also

[Networking Classes](#) | [NewIPEndPoint](#) | [ipendpoint_object.IPAddress](#)

socket_object.Available Property

Gets the number of data bytes currently available to receive from a **Socket**.

```
...socket_object.Available
```

Prerequisites

The **Socket** must be open and ready to receive data.

Parameters

None

Remarks

This property returns the number of bytes available on an open **Socket**. If this number is greater than zero, a **Receive** or **ReceiveFrom** method may be called to read data. Throws an **Exception** if the **Socket** is not open or an error occurs.

This method may be used to poll for data to read. A better solution is to set the **ReceiveTimeout** property for the **Socket**.

Examples

```
While ts.Available = 0      ' Test if anything to receive
    Thread.Sleep(1000)      ' Wait 1 second
End While
ts.Receive(recv, 1500)      ' Receive the data
```

See Also

[Networking Classes](#) | [socket_object.Blocking](#) | [socket_object.ReceiveTimeout](#)

socket_object.Blocking Property

Gets or sets the blocking I/O mode for a **Socket**.

```
socket_object.Blocking= <boolean_value>
-or-
...socket_object.Blocking
```

Prerequisites

The **Socket** must be open in order to set this flag.

Parameters

None

Remarks

This property sets or gets the state of the blocking mode for a **Socket**. If the **Socket** is in blocking mode, calls to receive data wait until data is available, and calls to send data wait if the output queue is full. If the **Socket** is not in blocking mode, calls to send or receive data throw an **Exception** if they would have to wait.

By default **Sockets** are created in blocking mode.

Non-blocking mode may be used to poll for data to read by repeatedly issuing receive requests and handling the **Exception**. A better solution is to use the **Available** property or to set the **ReceiveTimeout** or **SendTimeout** property for the **Socket**.

Examples

```
ts.Blocking = 0           ' Set to non-blocking mode
While ts.Available = 0    ' Test if anything to receive
    Thread.Sleep(1000)    ' Wait 1 second
End While
ts.Receive(recv, 1500)    ' Receive the data
```

See Also

[Networking Classes](#) | [socket_object.ReceiveTimeout](#) | [socket_object.SendTimeout](#)

socket_object.Close Method

Closes the network connection associated with a **Socket**, **TcpListener**, **TcpClient**, or **UdpClient Object**.

```
socket_object.Close  
-or-  
tcplistener_object.Close  
-or-  
tcpclient_object.Close  
-or-  
udpclient_object.Close
```

Prerequisites

None

Parameters

None

Remarks

The **Close** method may be used to close the network connection and free up resources. If it is called with a **TcpListener**, **TcpClient**, or **UdpClient Object**, the underlying **Socket** is actually closed.

If the **Socket** is not currently open, no error occurs.

Examples

```
Dim tl As New TcpListener(ep)  
Dim sock As Socket  
...  
tl.Close  
sock.Close
```

See Also

[Networking Classes](#)

socket_object.Connect Method

Initiates a TCP client connection with a remote TCP server.

```
socket_object.Connect ( remote_endpoint )
```

Prerequisites

The **Socket Object** must have been created by a *tcpclient_object.Client* method with the *endpoint* parameter omitted.

Parameters

remote_endpoint

A required **IPEndPoint Object** that specifies the IP address and port number of the remote endpoint to which you wish to connect.

Remarks

This method is only called when the remote endpoint of a connection was not specified in the constructor for the initial **TcpClient Object** from which the **Socket** was obtained.

Examples

```
Dim tc As New TcpClient() ' Optional endpoint not specified
Dim sock As Socket
Dim ep As New IPEndPoint("192.168.0.3", 1234)
sock = tc.Client
sock.Connect(ep)
```

See Also

[Networking Classes](#) | [New TcpClient Constructor](#)

socket_object.Receive Method

Receives a message from an open TCP connection.

```
...socket_object.Receive( input_buffer, max_length )
```

Prerequisites

An active TCP connection must exist for the **Socket**.

The **Socket Object** must have been created by the *tcpclient_object*.**Client** method or the *tcplistener_object*.**AcceptSocket** method.

Parameters

input_buffer

A **ByRef String** variable where the received data is stored.

max_length

The maximum number of data bytes that are read. If more bytes are available than this maximum, they must be read by subsequent **Receive** method calls.

Remarks

If blocking is enabled, this method blocks until some data is received. There is no guarantee that an entire datagram is received at once.

This method returns the number of bytes of data received. If the number is zero, this indicates that the TCP connection has been broken by either the local or remote endpoint. In this case, the program should close the **Socket**.

If any other network errors occur, this method throws an **Exception**.

Examples

```
Dim ep As New IPEndPoint("192.168.0.3", 1234)
Dim tc As New TcpClient(ep)
Dim sock As Socket
Dim input As String
Dim count As Integer
sock = tc.Client
count = sock.Receive(input, 2000)
```

See Also

[Networking Classes](#) | [socket_object.ReceiveFrom](#)

socket_object.ReceiveFrom Method

Receives a message from an open UDP **Socket**.

```
...socket_object.ReceiveFrom( input_buffer, max_length, remote_endpoint )
```

Prerequisites

The **Socket Object** must be open for UDP I/O.

The **Socket Object** must have been created by the *udpclient_object.Client* method.

Parameters

input_buffer

A **ByRef String** variable where the received data is stored.

max_length

The maximum number of data bytes that are read. If more bytes are available than this maximum, **they are lost**.

remote_endpoint

A **ByRef IPEndPoint Object** that receives endpoint information identifying the remote source of the received data. The original contents of *remote_endpoint* are ignored and replaced by the new information.

Remarks

If blocking is enabled, this method blocks until some data is received. The entire datagram is transferred by this method, if the *max_length* value is large enough.

Because of internal limitations on datagram size, *max_length* values greater than 1536 are not useful.

This method returns the number of bytes of data received. If that number is zero, this indicates that the **Socket** has been disconnect and should therefore be closed.

If any other network errors occur, this method throws an **Exception**.

Examples

```
Dim local_ep As New IPEndPoint("", 1234) ' Receive data for port 1234.
Dim uc As New UdpClient(local_ep)
Dim remote_ep As IPEndPoint
Dim sock As Socket
Dim input As String
```

GPL Dictionary Pages

```
Dim count As Integer
sock = uc.Client
count = sock.ReceiveFrom(input, 2000, remote_ep)
Console.WriteLine("Remote IP address: " & remote_ep.IPAddress)
Console.WriteLine("Remote Port: " & CStr(remote_ep.Port))
```

See Also

[Networking Classes](#) | [socket object.Receive](#)

socket_object.ReceiveTimeout Property

Sets or Gets the timeout period, in milliseconds, for a **Socket** to block while waiting to receive data.

```
socket_object.ReceiveTimeout = <timeout>  
-or-  
...socket_object.ReceiveTimeout
```

Prerequisites

The **Socket** must currently be open to set this property.

Parameters

None

Remarks

This property allows you to set the timeout period for a **Receive** or **ReceiveFrom** method. It only applies if the **Socket** is set to blocking. If a receive request blocks waiting for data, it will only wait for the specified timeout period. If that time is exceeded, the receive requests throws an **Exception**. If the timeout period is set to 0, the timeout is disabled and a request may block indefinitely.

Examples

```
ts.ReceiveTimeout = 30000      ' Timeout in 30 seconds  
ts.Receive(recv, 1500)        ' Receive the data
```

See Also

[Networking Classes](#) | [socket_object.Blocking](#) | [socket_object.SendTimeout](#)

socket_object.Send Method

Sends a message to an open TCP connection.

```
...socket_object.Send( output_buffer, max_length )
```

Prerequisites

An active TCP connection must exist for the **Socket**.

The **Socket Object** must have been created by the *tcpclient_object*.**Client** method or the *tcplistener_object*.**AcceptSocket** method.

Parameters

output_buffer

The **String** value that is sent.

max_length

An optional value indicating the maximum number of data bytes to send.
If omitted or zero, the entire *output_buffer* string is sent.

Remarks

If blocking is enabled, this method blocks if the output queue is full.

This method returns the number of bytes of data actually sent. If in blocking mode, the returned value is always equal to the number of bytes requested. In non-blocking mode, the value may be less than the number of bytes requested. In that case, you should re-issue the **Send** to output the remainder of the bytes.

If any network errors occur, this method throws an **Exception**.

Examples

```
Dim ep As New IPEndPoint("192.168.0.3", 1234)
Dim tc As New TcpClient(ep)
Dim sock As Socket
Dim output As String
Dim count As Integer
sock = tc.Client
...
count = sock.Send(output)
```

See Also

[Networking Classes](#) | [socket_object.SendTo](#)

socket_object.SendTimeout Property

Sets or Gets the timeout period, in milliseconds, for a **Socket** to block while waiting to send data.

```
socket_object.SendTimeout = <timeout>  
-or-  
...socket_object.SendTimeout
```

Prerequisites

None

Parameters

None

Remarks

The property allows you to set the timeout period for a **Send** or **SendTo** method. It only applies if the **Socket** is set to blocking. If a send request blocks waiting for the output queue, it will only wait for the specified timeout period. If that time is exceeded, the send request throws an **Exception**. If the timeout period is set to 0, the timeout is disabled and a send may block indefinitely.

Examples

```
ts.SendTimeout = 30000    ' Timeout in 30 seconds  
ts.Send(trns, 1500)      ' Send the data
```

See Also

[Networking Classes](#) | [socket_object.Blocking](#) | [socket_object.ReceiveTimeout](#)

socket_object.SendTo Method

Sends a message using an open UDP **Socket**.

```
...socket_object.SendTo( output_buffer, max_length, remote_endpoint )
```

Prerequisites

The **Socket Object** must be open for UDP I/O.

The **Socket Object** must have been created by the *udpclient_object*.**Client** method.

Parameters

output_buffer

The **String** value that is sent.

max_length

An optional value indicating the maximum number of data bytes to send. If omitted or zero, the entire *output_buffer* string is sent.

remote_endpoint

An **IPEndPoint Object** that contains endpoint information identifying the remote destination for the data sent.

Remarks

If blocking is enabled, this method blocks if the output queue is full.

This method returns the number of bytes of data actually sent. If that number is less than the number requested, you should re-issue the **SendTo** to output the remainder of the bytes.

If any network errors occur, this method throws an **Exception**.

Examples

```
Dim uc As New UdpClient()
Dim remote_ep As IPEndPoint("192.168.0.5")
Dim sock As Socket
Dim output As String
Dim count As Integer
sock = uc.Client
count = sock.SendTo(output, 0, remote_ep)
...
count = sock.ReceiveFrom(input, 2000, remote_ep) ' Get new remote endpoint
...
```

```
count = sock.SendTo(output, 0, remote_ep)      ' Reply to previous sender
```

See Also

[Networking Classes](#) | [socket_object.Send](#)

New TcpClient Constructor

Constructor for creating a **TcpClient Object** and optionally connecting to a remote TCP server.

New TcpClient (*endpoint*)

Prerequisites

None

Parameters

endpoint

An optional **IPEndPoint Object** that contains the IP address and port identifying the remote endpoint of a TCP server. If omitted, a **Connect** method must be called later for the TCP client **Socket** before I/O can be performed.

Remarks

This constructor creates a new **TcpClient Object** and creates the underlying **Socket**. If the optional *endpoint* parameter is specified, a connect request is sent immediately to the remote server. If it is omitted, a **Connect** method must be called for the TCP client **Socket** before I/O can be performed.

Examples

```
Dim ep As New IPEndPoint("192.168.0.2", 1234) ' Port 1234 at address 192.168.0.2
Dim tc As New TcpClient(ep)                  ' Connect to remote endpoint

Dim tc As New TcpClient()                     ' Create socket but do not connect
```

See Also

[Networking Classes](#) | [socket_object.Connect](#)

tcpclient_object.Client Method

Returns the **Socket Object** associated with a **TcpClient Object**.

```
...tcpclient_object.Client
```

Prerequisites

None

Parameters

None

Remarks

Since all I/O is performed on **Sockets**, this method allows the **Socket** associated with a **TcpClient** object to be accessed.

Examples

```
Dim tc As New TcpClient(ep)  
Dim sock As Socket  
sock = tc.Client
```

See Also

[Networking Classes](#) | [udpclient_object.Client](#)

tcpclient_object.Close Method

Closes the network connection associated with a **Socket**, **TcpListener**, **TcpClient**, or **UdpClient Object**.

```
socket_object.Close  
-or-  
tcplistener_object.Close  
-or-  
tcpclient_object.Close  
-or-  
udpclient_object.Close
```

Prerequisites

None

Parameters

None

Remarks

The **Close** method may be used to close the network connection and free up resources. If it is called with a **TcpListener**, **TcpClient**, or **UdpClient Object**, the underlying **Socket** is actually closed.

If the **Socket** is not currently open, no error occurs.

Examples

```
Dim tl As New TcpListener(ep)  
Dim sock As Socket  
...  
tl.Close  
sock.Close
```

See Also

[Networking Classes](#)

New TcpListener Constructor

Constructor for creating a **TcpListener Object** that allows a TCP server to be created.

New TcpListener (*endpoint*)

Prerequisites

None

Parameters

endpoint

An **IPEndPoint Object** that contains the IP address and port identifying the local endpoint for connections accepted by this TCP server. The IP address of this endpoint is ignored since GPL controllers only have a single IP address. The port number determines the port on which the server listens.

Remarks

This constructor creates a new **TcpListener Object** and creates the underlying **Socket**. It does not actually begin listening for connections until the **Start** method is called. These **Objects** are the basis for implementing TCP servers.

Examples

```
Dim ep As New IPEndPoint("", 1234)    ' Listen on port 1234
Dim tl As New TcpListener(ep)         ' Create listener object
```

See Also

[Networking Classes](#) | [tcplistener_object.Start](#)

tcplistener_object.AcceptSocket Method

Accepts a TCP connection and returns a new **Socket Object** for performing I/O on that connection.

```
...tcplistener_object.AcceptSocket
```

Prerequisites

The TCP listener associated with the *tcplistener_object* should have already been started.

Parameters

None

Remarks

This method is used by a TCP server to accept a connection request from a remote TCP client. It creates a new **Socket** for performing I/O with that client. If no connection requests are pending, this method blocks until one is received. To avoid blocking, use the **Pending** property before calling **AcceptSocket**.

If any network errors occur, this method throws an **Exception**.

Examples

```
Dim ep As New IPEndPoint("", 1234)    ' Listen on port 1234
Dim tl As New TcpListener(ep)         ' Create listener object
Dim sock As Socket
tl.Start
sock = tl.AcceptSocket
```

See Also

[Networking Classes](#) | [tcplistener_object.Pending](#)

tcplistener_object.Close Method

Closes the network connection associated with a **Socket**, **TcpListener**, **TcpClient**, or **UdpClient Object**.

```
socket_object.Close  
-or-  
tcplistener_object.Close  
-or-  
tcpclient_object.Close  
-or-  
udpclient_object.Close
```

Prerequisites

None

Parameters

None

Remarks

The **Close** method may be used to close the network connection and free up resources. If it is called with a **TcpListener**, **TcpClient**, or **UdpClient Object**, the underlying **Socket** is actually closed.

If the **Socket** is not currently open, no error occurs.

Examples

```
Dim tl As New TcpListener(ep)  
Dim sock As Socket  
...  
tl.Close  
sock.Close
```

See Also

[Networking Classes](#)

tcplistener_object.Pending Property

Gets a **Boolean** value that indicates if there are any TCP connection requests pending.

```
...tcplistener_object.Pending
```

Prerequisites

The TCP listener associated with the *tcplistener_object* must have already been started.

Parameters

None

Remarks

This property is used by a TCP server to test if there are any pending connection requests for a **TcpListener Object**. If so, it returns **True**. Otherwise it returns **False**. If there is a pending request, call the **AcceptSocket** method to accept it.

If any network errors occur, this property returns **False**.

Examples

```
Dim tl As New TcpListener(ep)      ' Create listener object
Dim sock As Socket
tl.Start
If tl.Pending Then
    sock = tl.AcceptSocket
End If
```

See Also

[Networking Classes](#) | [tcplistener_object.AcceptSocket](#)

tcplistener_object.Start Method

Start listening for TCP connection requests.

tcplistener_object.Start

Prerequisites

None

Parameters

None

Remarks

This method is used by TCP servers to start listening for connection requests from remote TCP clients. You can test if any requests are received by using the **Pending** property. After a request is received, it is accepted by calling the **AcceptSocket** method. After you accept a connection request, you can call the **Stop** method to cease accepting any further connection requests if you wish. Executing the **Stop** method does not effect your ability to continue to service datagrams for connections that have already been established.

If any network errors occur, this method throws an **Exception**.

Examples

```
Dim tl As New TcpListener(ep)      ' Create listener object
Dim sock As Socket
tl.Start
sock = tl.AcceptSocket
```

See Also

[Networking Classes](#) | [tcplistener_object.AcceptSocket](#)

tcplistener_object.Stop Method

Stop listening for TCP connection requests.

tcplistener_object.Stop

Prerequisites

None

Parameters

None

Remarks

This method is used by TCP servers when they are done listening for connection requests from remote TCP clients. Executing this method does not effect your ability to continue to service datagrams for connections that have already been established.

No error occurs if the listener is not active.

Examples

```
Dim tl As New TcpListener(ep)      ' Create listener object
Dim sock As Socket
tl.Start
sock = tl.AcceptSocket
tl.Stop
```

See Also

[Networking Classes](#) | [tcplistener_object.Start](#)

New UdpClient Constructor

Constructor for creating a **UdpClient Object**.

New UdpClient (*endpoint*)

Prerequisites

None

Parameters

endpoint

An optional **IPEndPoint Object** that contains the IP address and port identifying the local endpoint for datagrams recognized by this UDP **Socket**. The IP address of this endpoint is ignored since GPL controllers only have a single IP address. If the port is non-zero, only datagrams to the specified port can be received.

Remarks

This constructor creates a new **UdpClient Object** and creates the underlying **Socket**. No network I/O is generated by this method.

Examples

```
Dim ep As New IPEndPoint("", 1234) ' Port 1234
Dim uc As New UdpClient(ep)       ' Create a socket for UDP communications
```

See Also

[Networking Classes](#) | [udpclient_object.Client](#)

udpclient_object.Client Method

Returns the **Socket Object** associated with a **UdpClient Object**.

```
...udpclient_object.Client
```

Prerequisites

None

Parameters

None

Remarks

Since all I/O is performed on **Sockets**, this method allows the **Socket** associated with a **UdpClient Object** to be accessed.

Examples

```
Dim tc As New UdpClient(ep)
Dim sock As Socket
sock = tc.Client
```

See Also

[Networking Classes](#) | [tcpclient_object.Client](#)

udpclient_object.Close Method

Closes the network connection associated with a **Socket**, **TcpListener**, **TcpClient**, or **UdpClient Object**.

```
socket_object.Close  
-or-  
tcplistener_object.Close  
-or-  
tcpclient_object.Close  
-or-  
udpclient_object.Close
```

Prerequisites

None

Parameters

None

Remarks

The **Close** method may be used to close the network connection and free up resources. If it is called with a **TcpListener**, **TcpClient**, or **UdpClient Object**, the underlying **Socket** is actually closed.

If the **Socket** is not currently open, no error occurs.

Examples

```
Dim tl As New TcpListener(ep)  
Dim sock As Socket  
...  
tl.Close  
sock.Close
```

See Also

[Networking Classes](#)

Profile Class

Profile Class Summary

The following pages provide detailed information on the properties and methods of the **Profile Class**. This class defines the attributes of objects that are used to specify the performance parameters for a typical motion. That is, a **Profile Object** contains speed, acceleration, deceleration, in range criteria and other specifications that dictate how a motion is to be performed. The basic motion instruction, **Move.Loc**, takes as its two arguments a **Profile Object** and a **Location Object**. The **Location Object** specifies the destination for the robot motion and the **Profile Object** specifies how the robot is to get to the destination.

As is standard in GPL, conversions between different arithmetic types, e.g. **Boolean**, **Integer**, **Single**, **Double**, are automatically performed as required. So, for numeric properties and methods of the **Profile Class**, it is not necessary to have different variations of these members to deal with the different possible mixes of input parameter data types. Also, as appropriate, the properties and methods generally produce results that are formatted as **Double**'s. These results will automatically be converted to smaller data types as necessary, e.g. **Double** -> **Integer**, and will not generate an error so long as numeric overflow does not occur.

The table below briefly summarizes the properties and methods that are described in greater detail in the following sections.

Member	Type	Description
<u>profile_obj.Speed</u>	Property	Sets and gets peak motion speed specified as a percentage of the nominal speed.
<u>profile_obj.Speed2</u>	Property	Sets and gets the secondary peak motion speed specification as a percentage of their nominal speeds for selected axes during Cartesian motions.
<u>profile_obj.Accel</u>	Property	Sets and gets peak motion acceleration specified as a percentage of the nominal acceleration.
<u>profile_obj.Decel</u>	Property	Sets and gets peak motion deceleration specified as a percentage of the nominal deceleration.
<u>profile_obj.AccelRamp</u>	Property	Sets and gets duration for ramping up to the peak acceleration, specified in seconds.
<u>profile_obj.DecelRamp</u>	Property	Sets and gets duration for ramping up to the peak deceleration, specified in seconds.
<u>profile_obj.Straight</u>	Property	Sets and gets Boolean indicating if the robot is to follow a straight-line path.
<u>profile_obj.InRange</u>	Property	Sets and gets constraint that specifies if the robot should be stopped at the end of the motion and when the robot is close enough to the final destination to be considered at its final position.
<u>profile_obj.Clone</u>	Method	Method that returns a copy of the <i>profile_obj</i> .

profile_object.Accel Property

Sets and gets the peak motion acceleration defined as the percentage of the nominal acceleration.

```
profile_object.Accel = <new_value>
-or-
...profile_object.Accel
```

Prerequisites

Takes effect when the *profile_object* is passed as a parameter to a **Move Class** method and the specified motion segment is generated.

Parameters

None

Remarks

When generating a motion segment, the **Accel** property defines the peak acceleration that the motion can achieve. An **Accel** value of 100 corresponds to the nominal (100%) acceleration for the specified type of motion. The **Accel** value can range from 1.0 up to a maximum value permitted for the robot. For a **Straight**-line motion, the acceleration is computed along the path and about the Cartesian rotational angles defined by the robot's kinematic module. For joint motions, the acceleration percentage is applied to the joint angles.

The acceleration that the robot actually achieves for a given motion may be different than the **Accel** value for a number of reasons: if an **AccelRamp** (s-curve profile) value is specified, the motion may not be long enough to ramp up to the specified acceleration; the **Accel** value may be limited by the maximum permitted **Accel** value; or the **Accel** value may be automatically scaled if the Parameter Database "Couple %accel/%decel to %speed" parameter is set. The Parameter DB value is a convenience feature that automatically scales the specified **Accel** and **Decel** values with the **Speed** so that slow motions have gentler accelerations and decelerations and fast motions accelerate and decelerate as quickly as possible.

When a **New Profile** is created, its properties are automatically set to the default values specified in the controller's Configuration Database. Therefore, the **Accel** parameter only needs to be set if you wish to deviate from the default value.

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
prof1.Accel = 50              ' Only accelerate at 50% of nominal rate
Move.Loc (loc1, prof1)       ' Perform motion to previously defined
                              ' location, loc1 with performance "prof1"
```

See Also

[Profile Class](#) | [*profile_object.AccelRamp*](#) | [*profile_object.Decel*](#) | [*profile_object.DecelRamp*](#)

profile_object.AccelRamp Property

Sets and gets the duration for ramping up to the peak acceleration, specified in seconds.

```
profile_object.AccelRamp = <new_value>
-or-
...profile_object.AccelRamp
```

Prerequisites

Takes effect when the *profile_object* is passed as a parameter to a **Move Class** method and the specified motion segment is generated.

Parameters

None

Remarks

When generating a motion segment, the **AccelRamp** property specifies how long, in seconds, it takes for the **Accel** to achieve its specified value. Likewise, this time is also used for ramping the **Accel** down to zero. If the **AccelRamp** time is set to zero, at the start of a motion, the **Accel** command instantaneously jumps up to its specified value and then, at the end of acceleration period, instantaneously drops down to zero. A zero **AccelRamp** time corresponds to a square wave acceleration curve and commands an infinite jerk, i.e. rate of change of the acceleration. A non-zero **AccelRamp** time produces a trapezoidal acceleration curve, which is often referred to as an s-curve profile.

S-curve acceleration and deceleration profiles limit the impact of starting and stopping motions and help to reduce the excitation of resonances (or ringing) in the robot structure. An s-curve profile can often reduce the settling time at the end of the motion since each axis more smoothly glides into its final position with less oscillations. On the other hand, an s-curve profile will lengthen the planned duration of a motion since the average acceleration and deceleration will be less than a square wave profile. So, while most robots will benefit from s-curve profiles, for low accelerations or for very stiff robots, a square wave acceleration profile may be more beneficial.

The actual acceleration ramp time for a given motion may be different than the **AccelRamp** value for a number of reasons: if the motion is short, there may not be sufficient time to ramp all of the way up to the **Accel** value; or the **AccelRamp** value may be automatically scaled by with the **Accel** value if the Parameter Database “Couple %accel/%decel to %speed” parameter is set. The Parameter DB value is a convenience feature that automatically scales the specified **AccelRamp** and **Accel** values with the **Speed** so that slow motions have gentler accelerations with shorter ramp times and fast motions accelerate more quickly but have longer ramp times.

When a **New Profile** is created, its properties are automatically set to the default values specified in the controller's Configuration Database. Therefore, the **AccelRamp** parameter only needs to be set if you wish to deviate from the default value.

GPL Dictionary Pages

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
prof1.Accel = 50              ' Only accelerate at 50% of nominal rate
prof1.AccelRamp = 0.1         ' Take 0.1 sec to achieve 50% nominal accel
Move.Loc (loc1, prof1)       ' Perform motion to previously defined
                              ' location, loc1 with performance "prof1"
```

See Also

[Profile Class](#) | [profile_object.Accel](#) | [profile_object.Decel](#) | [profile_object.DecelRamp](#)

profile_object.Clone Method

Method that returns a copy of the *profile_object*.

...profile_object.Clone

Prerequisites

None

Parameters

None

Remarks

For objects, if a program contains a simple assignment statement:

object_1 = *object_2*

the result is that *object_1* points to the same data as *object_2*. Any subsequent change of a property in either *object_1* or *object_2* affects the data associated with both objects.

If you wish to make an independent copy of an object, the **Clone** method is the standard means for performing this operation:

object_1 = *object_2.Clone*

Examples

```
Dim prof1 As New Profile ' Create new profile set to default values
Dim prof2 As Profile     ' Create new profile with no data allocated
prof1.Decel = 25          ' Only decelerate at 25% of nominal rate
prof2 = prof1.Clone       ' Makes a copy of prof1 data
prof2.Accel = 50          ' Doesn't affect prof1 data
```

See Also

[Profile Class](#)

profile_object.Decel Property

Sets and gets the peak motion deceleration defined as the percentage of the nominal deceleration.

```
profile_object.Decel = <new_value>
-or-
...profile_object.Decel
```

Prerequisites

Takes effect when the *profile_object* is passed as a parameter to a **Move Class** method and the specified motion segment is generated.

Parameters

None

Remarks

When generating a motion segment, the **Decel** property defines the peak deceleration that the motion can achieve. An **Decel** value of 100 corresponds to the nominal (100%) deceleration for the specified type of motion. The **Decel** value can range from 1.0 up to a maximum value permitted for the robot. For a **Straight**-line motion, the Deceleration is computed along the path and about the Cartesian rotational angles defined by the robot's kinematic module. For joint motions, the deceleration percentage is applied to the joint angles.

The deceleration that the robot actually achieves for a given motion may be different than the **Decel** value for a number of reasons: if an **DecelRamp** (s-curve profile) value is specified, the motion may not be long enough to ramp up to the specified deceleration; the **Decel** value may be limited by the maximum permitted **Decel** value; or the **Decel** value may be automatically scaled if the Parameter Database "Couple %accel/%decel to %speed" parameter is set. The Parameter DB value is a convenience feature that automatically scales the specified **Accel** and **Decel** values with the **Speed** so that slow motions have gentler accelerations and decelerations and fast motions accelerate and decelerate as quickly as possible.

When a **New Profile** is created, its properties are automatically set to the default values specified in the controller's Configuration Database. Therefore, the **Decel** parameter only needs to be set if you wish to deviate from the default value.

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
prof1.Decel = 25              ' Only decelerate at 25% of nominal rate
Move.Loc (loc1, prof1)       ' Perform motion to previously defined
                              ' location, loc1 with performance "prof1"
```

See Also

[Profile Class](#) | [profile_object.Accel](#) | [profile_object.AccelRamp](#) | [profile_object.DecelRamp](#)

profile_object.DecelRamp Property

Sets and gets the duration for ramping up to the peak deceleration, specified in seconds.

```
profile_object.DecelRamp = <new_value>
-or-
...profile_object.DecelRamp
```

Prerequisites

Takes effect when the *profile_object* is passed as a parameter to a **Move Class** method and the specified motion segment is generated.

Parameters

None

Remarks

When generating a motion segment, the **DecelRamp** property specifies how long, in seconds, it takes for the **Decel** to achieve its specified value. Likewise, this time is also used for ramping the **Decel** down to zero. If the **DecelRamp** time is set to zero, at the start of the motion deceleration period, the **Decel** command instantaneously jumps up to its specified value and then, at the end of the motion, instantaneously drops down to zero. A zero **DecelRamp** time corresponds to a square wave deceleration curve and commands an infinite jerk, i.e. rate of change of the deceleration. A non-zero **DecelRamp** time produces a trapezoidal deceleration curve, which is often referred to as an s-curve profile.

S-curve acceleration and deceleration profiles limit the impact of starting and stopping motions and help to reduce the excitation of resonances (or ringing) in the robot structure. An s-curve profile can often reduce the settling time at the end of the motion since each axis more smoothly glides into its final position with less oscillations. On the other hand, an s-curve profile will lengthen the planned duration of a motion since the average acceleration and deceleration will be less than a square wave profile. So, while most robots will benefit from s-curve profiles, for low decelerations or for very stiff robots, a square wave deceleration profile may be more beneficial.

The actual deceleration ramp time for a given motion may be different than the **DecelRamp** value for a number of reasons: if the motion is short, there may not be sufficient time to ramp all of the way up to the **Decel** value; or the **DecelRamp** value may be automatically scaled by with the **Decel** value if the Parameter Database “Couple %accel/%decel to %speed” parameter is set. The Parameter DB value is a convenience feature that automatically scales the specified **DecelRamp** and **Decel** values with the **Speed** so that slow motions have gentler decelerations with shorter ramp times and fast motions decelerate more quickly but have longer ramp times.

When a **New Profile** is created, its properties are automatically set to the default values specified in the controller's Configuration Database. Therefore, the **DecelRamp** parameter only needs to be set if you wish to deviate from the default value.

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
prof1.Decel = 25              ' Only decelerate at 25% of nominal rate
prof1.DecelRamp = 0.1         ' Take 0.1 sec to achieve 50% nominal decel
Move.Loc (loc1, prof1)       ' Perform motion to previously defined
                              ' location, loc1 with performance "prof1"
```

See Also

[Profile Class](#) | [profile_object.Accel](#) | [profile_object.AccelRamp](#) | [profile_object.Decel](#)

profile_object.InRange Property

Gets and sets the constraint that specifies if the robot should be stopped at the end of the motion and when the robot is close enough to the final destination to be considered at its final position.

```
profile_object.InRange = <new_value>
-or-
...profile_object.InRange
```

Prerequisites

Takes effect when the *profile_object* is passed as a parameter to a **Move Class** method and the specified motion segment is generated.

Parameters

None

Remarks

Whenever the robot picks up a part or places it at its final destination, the robot should normally be brought to a complete stop and any small position errors should be eliminated (nulled) before the part is grasped or released. Conversely, if the robot is moving through intermediate (via) positions simply to clear obstacles, bringing the robot to a stop at these positions increases the cycle time without providing any benefit. Also, when the robot is to be brought to a stop, there are instances where it is beneficial to spend more time reducing the final positioning errors to the tightest possible position constraint for the robot and other times when a looser constraint is acceptable to save cycle time.

The **InRange** property specifies if the robot is to stop at the end of motion and, if so, how tight a position error constraint should be applied to determine when the robot has reached its final destination. The value of this property is interpreted as follows:

InRange Value	Interpretation
<0	Don't stop the robot at the end of the motion. Blend with the next motion if possible.
0	Stop the robot at the end of the motion, but do not apply any position error constraints. This means that as soon as the final set point command has been issued to the servos, GPL will signal that the motion has been completed.
Small number >0	Stop the robot at the end of the motion, but use a very small (loose) position error constraint. This will ensure that the robot has approximately reached the specified destination before GPL considers that the motion has been completed.
Large number <= 100	Stop the robot at the end of the motion and apply a stringent position error constraint. If this value is 100, the robot will have to be within its tightest error envelope before GPL considers the motion completed.

	Values greater than 100 can be specified, but these require smaller error tolerances than are recommended by the manufacturer of the robot.
--	---

When a **New Profile** is created, its properties are automatically set to reasonable default values. Normally, the **InRange** property defaults to 100. Therefore, the **InRange** parameter only needs to be altered if this default value is not appropriate.

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
prof1.InRange = 10           ' Stop at EOM, reduced requirement for inrange
Move.Loc (loc1, prof1)       ' Perform motion to previously defined
                              ' location, loc1
```

See Also

[Profile Class](#)

profile_object.Speed Property

Sets and gets the peak motion speed specified as a percentage of the nominal speed.

```
profile_object.Speed = <new_value>
-or-
...profile_object.Speed
```

Prerequisites

Takes effect when the *profile_object* is passed as a parameter to a **Move Class** method and the specified motion segment is generated.

Parameters

None

Remarks

When generating a motion segment, the **Speed** property defines the peak speed that the motion can achieve. A **Speed** value of 100 corresponds to the nominal (100%) speed for the specified type of motion. The **Speed** value can range from 1.0 up to a maximum value permitted for the robot. For a **Straight**-line motion, the speed is computed along the path and about the Cartesian rotational angles defined by the robot's kinematic module. For joint motions, the speed percentage is applied to the joint angles.

While 100% is normally the maximum operating speed recommended by the robot manufacturer, there are times that a greater **Speed** setting may be beneficial. Often, the 100% **Speed** setting is established for when the robot is carrying its maximum payload. Also, 100% **Speed** may be the sustained maximum speed setting, but higher burst speeds may be permitted.

The speed that the robot actually achieves for a given motion may be different than the specified **Speed** value for a number of reasons: the motion may not be long enough to ramp up to the specified speed given the available acceleration; the **Speed** value may be limited by the maximum permitted **Speed** value; or the operator may have set a slow "Test Speed" that scales down the specified **Speed** value.

When a **New Profile** is created, its properties are automatically set to the default values specified in the controller's Configuration Database. Therefore, the **Speed** parameter only needs to be set if you wish to deviate from the default value.

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
prof1.Speed = 50              ' Only go at half of the rated speed
Move.Loc (loc1, prof1)        ' Perform motion to previously defined
                               ' location, loc1 with performance "prof1"
```

See Also

[Profile Class](#) | [profile_object.Acce](#) | [profile_object.Decel](#) | [profile_object.Speed2](#)

profile_object.Speed2 Property

Sets and gets the secondary peak motion speed specification as a percentage of their nominal speeds for selected axes during Cartesian motions.

```
profile_object.Speed2 = <new_value>
-or-
...profile_object.Speed2
```

Prerequisites

Takes effect when the *profile_object* is passed as a parameter to a **Move Class** method and the specified Cartesian motion segment is generated.

Parameters

None

Remarks

For all joint interpolated and the majority of Cartesian motions, the standard **Speed** property is used to control the peak speed of the robot. However, for certain robot geometries and certain Cartesian (straight-line) motions, it is beneficial to have a secondary property to control motion speeds.

The **Speed2** property only applies to Cartesian motions and is generally used to specify a secondary speed setting to control the peak rotation speed for a motion. If **Speed2** is zero, both the peak translation and rotation are governed by the **Speed** property. If **Speed2** is non-zero, the peak Cartesian translation motion speed is limited by the **Speed** property and the peak Cartesian rotation speed is limited by **Speed2**. For a such a motion, the speed value that is more limiting will govern the overall motion timing.

For most motions, **Speed2** should be set to 0. However, if your robot has a wrist that can rotate very quickly and it is unpredictable as to whether the motion will be primarily a translation or a rotation, **Speed2** can be set low to limit the speed of a large rotation without negatively impacting motions that are primarily translations.

For some special kinematic modules, **Speed2** may also be applied to other degrees-of-freedom. Please see the Kinematic Library for specific information on these special uses.

Examples

```
Dim prof1 As New Profile      ' Create new profile set to default values
prof1.Straight = True         ' Limit Cartesian rotation speed
prof1.Speed2 = 25              ' Keep translation speed at full
prof1.Speed = 100              ' Perform motion to previously defined
Move.Loc (loc1, prof1)        ' location, loc1 with performance "prof1"
```

See Also

[**Profile Class**](#) | [*profile_object*.**Accel**](#) | [*profile_object*.**Decel**](#) | [*profile_object*.**Speed**](#)

profile_object.Straight Property

Sets and gets Boolean indicating if the robot's tool tip is to follow a straight-line path or if the path will be a function of the robot's geometry.

```
profile_object.Straight = <new_value>
-or-
...profile_object.Straight
```

Prerequisites

Takes effect when the *profile_object* is passed as a parameter to a **Move Class** method and the specified motion segment is generated.

Parameters

None

Remarks

For certain motions, the path of the robot's tool or the part being held by the robot is important and moving along a straight line is desirable. In other cases, the path may not be important. In the latter case, the robot may move faster if the path is defined by interpolating between the joint angles of the initial and final Locations.

If the **Straight** property is **True**, by making use of the system's built-in knowledge of the robot's geometry (i.e. kinematics), the robot's tool tip is moved along a straight-line path in Cartesian space. If **Straight** is **False**, the system will interpolate in joint angles to move the robot to its destination.

If the robot is a simple 1, 2, or 3 degree-of-freedom Cartesian mechanism with all linear axes, there is no difference between straight-line and joint interpolated motions. However, if the Cartesian robot has a rotary theta axis or if the robot is a non-Cartesian mechanism with rotary or parallel axes, the two motion types are quite different.

In situations where the path is not important, joint interpolated motions requires less processor time and the robot will often move more quickly.

By default, when a **New Profile** is created, Straight is set to **False**.

Examples

```
Dim prof1 As New Profile ' Create new profile set to default values
prof1.Straight = True
Move.Loc (loc1, prof1) ' Perform motion to previously defined
                        ' location, loc1 by moving along a straight path
```

See Also

Profile Class

Reference Frame Class

RefFrame Class Summary

The following pages provide detailed information on the properties and methods of the reference frame class, **RefFrame**. If one or more **Location Objects** are defined with respect to a **RefFrame Object**, when the position and/or orientation of the reference frame are altered, the position and orientation of all associated **Location Objects** are automatically adjusted as well.

RefFrame Objects are very useful when picking up or placing several parts that are at fixed positions relative to a base plate or when accessing pallets that have parts arranged in a rectangular grid. The assembly of a printed circuit board is a common example of the first situation. When a PCB enters into a machine for mounting electronic components, the position and orientation of the PCB is first accurately determined, typically using a vision system. The reference frame that represents the PCB is then updated and all of the positions and orientations of the components to be placed are automatically adjusted. The use of robots in the laboratory automation industry provides a good example for the use of pallet reference frames. In this case, samples to be tested are placed on a tray and arranged in a rectangular grid pattern. After the tray is located and its associated reference frame updated, the **RefFrame Class** provides a simple means for stepping from sample to sample.

To allow different types of static and dynamic reference frames to be represented, the **RefFrame Object** includes a **Type** property. At present, only a basic reference frame is supported and a pallet reference frame. In the future, additional types of reference frames will be added.

In general, each type of reference frame only makes use of a subset of the properties and methods of the **RefFrame Class**. The tables below summarize the properties and methods utilized for each type of reference frame.

Basic Reference Frame		
Member	Type	Description
<u>refframe_obj.Type</u>	Property	Set to 0 to indicate a basic reference frame.
<u>refframe_obj.Loc</u>	Property	Loc.Pos is set equal to the position and orientation of the reference frame by a GPL procedure.
<u>refframe_obj.Pos</u>	Method	Returns the absolute ("total") position and orientation for any type of reference frame object.
<u>refframe_obj.PosWrtRef</u>	Method	Returns the position for any type of reference frame while ignoring any further reference frames.

Pallet Reference Frame		
Member	Type	Description

<u>refframe_obj.Type</u>	Property	Set to 1 to indicate a pallet reference frame.
<u>refframe_obj.Loc</u>	Property	Loc.X , Y , and Z define the position of the first row, column and layer. The orientation of the X, Y, and Z axes of Loc define the direction for each row, column, and layer respectively.
<u>refframe_obj.Pos</u>	Method	Returns the absolute ("total") position and orientation for any type of reference frame object.
<u>refframe_obj.PosWrtRef</u>	Method	Returns the position for any type of reference frame while ignoring any further reference frames.
<u>refframe_obj.PalletIndex</u>	Property	Sets and gets the index for the next position along the pallet row, column, or layer (1 to n).
<u>refframe_obj.PalletMaxIndex</u>	Property	Sets and gets the maximum position index along the pallet row, column, or layer (1 to n).
<u>refframe_obj.PalletNextPos</u>	Method	Advances to the next pallet position.
<u>refframe_obj.PalletOrder</u>	Property	Sets and gets the parameter that specifies the order in which PalletNextPos indexes along the row, column, and layer indices.
<u>refframe_obj.PalletPitch</u>	Property	Sets and gets the step size for advancing along each row, column, or layer.
<u>refframe_obj.PalletRowColLay</u>	Method	Sets the next pallet position row, column, and layer indices in a single instruction.

refframe_object.Loc Property

Sets and gets a reference frame's **Location Object**, which typically contains the nominal position and orientation of the frame.

```
refframe_object.Loc = <Cartesian_location_object>
-or-
... refframe_object.Loc
```

Prerequisites

None

Parameters

None

Remarks

All reference frame types have an associated Cartesian **Location Object** that is pointed to by the **Loc** property. Typically, the nominal position and orientation of the reference frame is stored in this **Location** although the specific interpretation of this data is a function of the reference frame type.

Independent of the reference frame type, the *refframe_object.Loc.RefFrame* property always points to the next reference frame if *refframe_object* is itself relative to another frame.

The following table describes how to interpret the position and orientation data stored in the Cartesian **Location Object** pointed to by *refframe_object.Loc*.

RefFrame Type	<i>refframe_object.Loc</i> Contents
Basic	Contains the reference frame position and orientation. So, <i>refframe_object.Loc.Pos</i> represents the total position of <i>refframe_object</i> and <i>refframe_object.Loc.PosWrtRef</i> is the position and orientation of <i>refframe_object</i> with respect to any subsequent reference frames. If a program wishes to change the position and orientation of a basic frame, it must do so via <i>refframe_object.Loc</i> . However, if a program wishes to read the reference frame position and orientation, it is normally a better practice to use the <i>refframe_object.Pos</i> and <i>refframe_object.PosWrtRef</i> methods. These last two methods will return the current total and relative position for any type of reference frame.
Pallet	The XYZ position of the <i>refframe_object.Loc</i> defines the position of row 1, column 1, and layer 1 of the pallet. The orientation of <i>refframe_object.Loc</i> defines the direction of the rows, columns, and layers of the pallet. The X-axis of <i>refframe_object.Loc</i> defines the index direction for a row. The Y-axis defines the index direction for a

column. The Z-axis defines the index direction for layers.
--

As a convenience, when a new reference frame object is created, a Cartesian **Location Object** is automatically created and linked to the reference frame. By default, this **Location** will have its position and orientation angles set to zero.

Examples

```

Dim refl As New RefFrame          ' Also allocates Loc
Dim loc1 As New Location
refl.Loc.XYZ(100,90,-80,0,0,45)   ' Define base frame
loc1.RefFrame = refl              ' Define loc1 wrt refl
loc1.XYZ(10,0,0,0,180,0)          ' Define loc1 poswrtref
Console.WriteLine(loc1.Pos.X)     ' Displays 107.07
Console.WriteLine(loc1.Pos.Y)     ' Displays 97.07
Console.WriteLine(loc1.Pos.Z)     ' Displays -80

```

See Also

[RefFrame Class](#) | [reframe object.Pos](#) | [reframe object.PosWrtRef](#)

reframe_object.PalletIndex Property

For a pallet reference frame, sets or gets the row, column or layer index for the next grid position to be accessed.

```
reframe_object.PalletIndex( row_col_lay ) = <next_index>
-or-
... reframe_object.PalletIndex( row_col_lay )
```

Prerequisites

The *reframe_object* must be a pallet reference frame.

Parameters

row_col_lay

A required numerical expression that is equal to 1 if the row index is to be accessed, 2 if the column index is to be accessed, or 3 if the layer index is to be accessed.

Remarks

This property permits a program to set or get the next row, column, or layer index to be accessed in a pallet reference frame. Each index can range from 1 to the maximum value for that dimension as specified by the object's **PalletMaxIndex** property. The row, column, and layer indices are always positive integer numbers. If you wish to step in a negative direction, the appropriate **PalletPitch** property for the *reframe_object* can be set to a negative number.

If you wish to set all 3 index values at once, you can make use of the object's **PalletRowColLay** method. If you want to just advance to the next logical pallet position, the **PalletNextPos** method can be invoked.

By default, when a new pallet reference frame is created, the pallet indices are set to 1, 1, 1.

Examples

```
Dim refl As New RefFrame          ' Also allocates Loc
Dim loc1 As New Location

refl.Type = 1                     ' Change to pallet frame
refl.Loc.XYZ(100,50,-80,0,0,0)   ' Define pallet base
refl.PalletPitch(1) = 10          ' Spacing along row
refl.PalletPitch(2) = 20          ' Spacing along column
refl.PalletMaxIndex(1) = 3        ' Define grid size
refl.PalletMaxIndex(2) = 3        ' Define grid size

loc1.RefFrame = refl              ' loc1.PosWrtRef all 0's
refl.PalletIndex(2) = 2           ' Set grid (1,2,1)
Console.WriteLine(loc1.Pos.X)     ' Displays 100
```

```
Console.WriteLine(loc1.Pos.Y)      ' Displays 70
```

See Also

[RefFrame Class](#) | [reframe_object.PalletMaxIndex](#) | [reframe_object.PalletNextPos](#) | [reframe_object.PalletRowColLay](#)

reframe_object.PalletMaxIndex Property

For a pallet reference frame, sets or gets the number of rows, columns, or layers in the pallet.

```
reframe_object.PalletMaxIndex(row_col_lay) = <maximum_index>
-or-
... reframe_object.PalletMaxIndex( row_col_lay )
```

Prerequisites

The *reframe_object* must be a pallet reference frame.

Parameters

row_col_lay

A required numerical expression that is equal to 1 if the number of rows is to be accessed, 2 if the number of columns is to be accessed, or 3 if the number of layers is to be accessed.

Remarks

This property allows a program to set or get the number of rows, columns or layers for a given pallet reference frame. The number of rows, columns or layers is specified by an integer number greater than or equal to 1.

To specify a specific pallet position, the **PalletIndex** properties must be set to at least 1 and cannot be greater than the applicable maximum values defined by the **PalletMaxIndex** property.

By default, when a new pallet reference frame is created, the maximum pallet indices are each set to 1.

Examples

```
Dim refl As New RefFrame          ' Also allocates Loc
Dim loc1 As New Location

refl.Type = 1                     ' Change to pallet frame
refl.Loc.XYZ(100,50,-80,0,0,0)    ' Define pallet base
refl.PalletPitch(1) = 10          ' Spacing along row
refl.PalletPitch(2) = 20          ' Spacing along column
refl.PalletMaxIndex(1) = 3        ' Define grid size
refl.PalletMaxIndex(2) = 3        ' Define grid size

loc1.RefFrame = refl              ' loc1.PosWrtRef all 0's
refl.PalletRowColLay(2,3,1)       ' Set grid position
Console.WriteLine(loc1.Pos.X)     ' Displays 110
Console.WriteLine(loc1.Pos.Y)     ' Displays 90
```

See Also

[RefFrame Class](#) | [reframe_object.PalletIndex](#) | [reframe_object.PalletRowColLay](#)

reframe_object.PalletNextPos Method

For a pallet reference frame, advances the pallet position to the next logical position.

reframe_object.PalletNextPos

Prerequisites

The *reframe_object* must be a pallet reference frame.

Parameters

None

Remarks

Given the current pallet position and the **PalletOrder**, this method advances the pallet to the next logical position. For example, if the current pallet position is at the last element in a row, 3rd column position, and 2nd layer, and the **PalletOrder** indicates that the pallet should be incremented by row, column and layer, **PalletNextPos** will advance to the 1st row element, 4th column element and 2nd layer.

If the initial pallet position is at the last row, column, and layer position, **PalletNextPos** changes the pallet position indices to 1,1,1.

If you want to randomly select the next pallet position, a program can utilize **PalletIndex** or **PalletRowColLay** instead of the **PalletNextPos** method.

Examples

```
Dim ref1 As New RefFrame          ' Also allocates Loc
Dim loc1 As New Location

ref1.Type = 1                     ' Change to pallet frame
ref1.Loc.XYZ(100,50,-80,0,0,0)   ' Define pallet base
ref1.PalletPitch(1) = 10         ' Spacing along row
ref1.PalletPitch(2) = 20         ' Spacing along column
ref1.PalletMaxIndex(1) = 3       ' Define grid size
ref1.PalletMaxIndex(2) = 3       ' Define grid size
ref1.PalletOrder = 2             ' Col, row, layer order

loc1.RefFrame = ref1             ' loc1.PosWrtRef all 0's
ref1.PalletRowColLay(3,1,1)      ' Set grid position
ref1.PalletNextPos               ' Advance to 3,2,1
Console.WriteLine(loc1.Pos.X)    ' Displays 120
Console.WriteLine(loc1.Pos.Y)    ' Displays 70
```

See Also

[RefFrame Class](#) | [reframe_object.PalletIndex](#) | [reframe_object.PalletOrder](#) | [reframe_object.PalletRowColLay](#)

refframe_object.PalletOrder Property

For a pallet reference frame, sets or gets the parameter that specifies the order in which the row, column, and layer indices are incremented.

```
refframe_object.PalletOrder= <indexing_order>
-or-
... refframe_object.PalletOrder
```

Prerequisites

The *refframe_object* must be a pallet reference frame.

Parameters

None

Remarks

Normally, the rows and columns of a pallet are defined such that a layer of rows and columns lie in the world coordinate system X-Y plane. If the rows and columns are defined in such a manner, you may wish to increment from one pallet position to the next in a different order than the standard row first, then column, then layer pattern. For example, you may want to stack from the bottom layer to the top layer before incrementing to the next row or column. The **PalletOrder** parameter allows a program to define the order in which the row, column, and layer indices are incremented.

The interpretation of this parameter is presented in the following table.

PalletOrder Value	Incrementing Order
0	Row, column, layer
1	Row, layer, column
2	Column, row, layer
3	Column, layer, row
4	Layer, row, column
5	Layer, column, row

By default, when a new pallet reference frame is created, the **PalletOrder** is set to 0 (row,column,layer).

Examples

```
Dim refl As New RefFrame          ' Also allocates Loc
Dim loc1 As New Location

refl.Type = 1                     ' Change to pallet frame
refl.Loc.XYZ(100,50,-80,0,0,0)   ' Define pallet base
refl.PalletPitch(1) = 10          ' Spacing along row
refl.PalletPitch(2) = 20          ' Spacing along column
refl.PalletMaxIndex(1) = 3        ' Define grid size
```

GPL Dictionary Pages

```
refl.PalletMaxIndex(2) = 3      ' Define grid size
refl.PalletOrder = 2           ' Col, row, layer order

loc1.RefFrame = refl           ' loc1.PosWrtRef all 0's
refl.PalletRowColLay(3,1,1)    ' Set grid position
refl.PalletNextPos             ' Advance to 3,2,1
Console.WriteLine(loc1.Pos.X)   ' Displays 120
Console.WriteLine(loc1.Pos.Y)   ' Displays 70
```

See Also

[RefFrame Class](#) | [reframe object.PalletNextPos](#)

refframe_object.PalletPitch Property

For a pallet reference frame, sets or gets the step size (pitch) between adjacent rows, columns, or layers in a pallet.

```
refframe_object.PalletPitch( row_col_lay )= <pitch_size>
-Or-
... refframe_object.PalletPitch( row_col_lay )
```

Prerequisites

The *refframe_object* must be a pallet reference frame.

Parameters

row_col_lay

A required numerical expression that is equal to 1 if the row pitch is to be accessed, 2 if the column pitch is to be accessed, or 3 if the layer pitch is to be accessed.

Remarks

This property allows a program to set or get the step size (pitch) between sequential rows, columns or layers for a pallet reference frame. The step sizes are in units of millimeters and can be both positive and negative real numbers.

Examples

```
Dim ref1 As New RefFrame      ' Also allocates Loc
Dim loc1 As New Location

ref1.Type = 1                 ' Change to pallet frame
ref1.Loc.XYZ(100,50,-80,0,0,0) ' Define pallet base
ref1.PalletPitch(1) = 10      ' Spacing along row
ref1.PalletPitch(2) = 20      ' Spacing along column
ref1.PalletMaxIndex(1) = 3    ' Define grid size
ref1.PalletMaxIndex(2) = 3    ' Define grid size

loc1.RefFrame = ref1          ' loc1.PosWrtRef all 0's
ref1.PalletRowColLay(2,3,1)   ' Set grid position
Console.WriteLine(loc1.Pos.X)  ' Displays 110
Console.WriteLine(loc1.Pos.Y)  ' Displays 90
```

See Also

[RefFrame Class](#)

reframe_object.PalletRowColLay Method

For a pallet reference frame, sets the row, column, and layer indices for the next grid position to be accessed.

```
reframe_object.PalletRowColLay( row, column, layer)
```

Prerequisites

The *reframe_object* must be a pallet reference frame.

Parameters

row

A required numerical expression that specifies the index for the next row to be accessed, where the row number is interpreted as an integer value that ranges from 1 to the maximum permitted row index for this pallet, i.e. *reframe_object.PalletMaxIndex*(1).

column

A required numerical expression that specifies the index for the next column to be accessed, where the column number is interpreted as an integer value that ranges from 1 to the maximum permitted column index for this pallet, i.e. *reframe_object.PalletMaxIndex*(2).

layer

A required numerical expression that specifies the index for the next layer to be accessed, where the layer number is interpreted as an integer value that ranges from 1 to the maximum permitted layer index for this pallet, i.e. *reframe_object.PalletMaxIndex*(3).

Remarks

This is a convenience method that allows a program to explicitly set the row, column, and layer indices for the next pallet element to be accessed. This method permits a program to randomly set or reset the next element. For example, if values of 1,1,1 are specified as the arguments to this method, the first pallet position will be accessed next.

By default, when a new pallet reference frame is created, the pallet indices are set to 1, 1, 1.

The operation performed by this method can also be accomplished by utilizing the **PalletIndex** property once for each of the three pallet indices or the **PalletNextPos** method can be invoked to advance to the next logical pallet position.

Examples

```

Dim ref1 As New RefFrame          ' Also allocates Loc
Dim loc1 As New Location

ref1.Type = 1                     ' Change to pallet frame
ref1.Loc.XYZ(100,50,-80,0,0,0)   ' Define pallet base
ref1.PalletPitch(1) = 10         ' Spacing along row
ref1.PalletPitch(2) = 20         ' Spacing along column
ref1.PalletMaxIndex(1) = 3       ' Define grid size
ref1.PalletMaxIndex(2) = 3       ' Define grid size

loc1.RefFrame = ref1              ' loc1.PosWrtRef all 0's
ref1.PalletRowColLay(2,3,1)      ' Set grid position
Console.WriteLine(loc1.Pos.X)    ' Displays 110
Console.WriteLine(loc1.Pos.Y)    ' Displays 90

```

See Also

[RefFrame Class](#) | [reframe_object.PalletIndex](#) | [reframe_object.PalletMaxIndex](#) | [reframe_object.PalletNextPos](#)

refframe_object.Pos Method

Returns a Cartesian **Location** equal to the current total position and orientation for any type of **RefFrame Object**.

```
... refframe_object.Pos(location_object)
```

Prerequisites

None

Parameters

location_object

A required Cartesian **Location Object** or a method or property that returns a Cartesian **Location Object** value. This parameter is not currently utilized but is included to support planned future reference frame types.

Remarks

For any type of reference frame object, this method returns a Cartesian **Location** whose value is equal to the current (instantaneous) total position and orientation of the frame taking into account any additional linked reference frames. In the case of a “basic” reference frame, the current location is equal to the contents of *refframe_object.Loc.Pos*. In the case of a dynamic reference frame, such as a pallet, the current total position and orientation is computed based upon the object properties, e.g. nominal location, current row, column and layer numbers.

This method returns the reference frame’s total position and orientation that is equivalent to the value used to compute the total position and orientation of a Cartesian **Location** that is defined with respect to the reference frame. For example, if a Cartesian **Location**, *loc1*, has its **RefFrame** pointer set equal to a reference frame, *ref1*, then *loc1.Pos* is equal to:

$$ref1.Pos(dummy).Mul(loc1.PosWrtRef)$$

Examples

```
Dim ref1 As New RefFrame          ' Also allocates Loc
Dim dum As New Location
ref1.Loc.XYZ(100,90,-80,0,0,45)    ' Define base frame
Console.WriteLine(ref1.Pos(dum).X) ' Displays 100
Console.WriteLine(ref1.Pos(dum).Y) ' Displays 90
Console.WriteLine(ref1.Pos(dum).Z) ' Displays -80
```

See Also

[RefFrame Class](#) | [refframe_object.PosWrtRef](#)

refframe_object.PosWrtRef Method

Returns a Cartesian **Location** equal to the current position and orientation for any type of **RefFrame Object** ignoring any further reference frames.

```
... refframe_object.PosWrtRef( location_object )
```

Prerequisites

None

Parameters

location_object

A required Cartesian **Location Object** or a method or property that returns a Cartesian **Location Object** value. This parameter is not currently utilized but is included to support planned future reference frame types.

Remarks

For any type of reference frame object, this method returns a Cartesian **Location** whose value is equal to the current (instantaneous) position and orientation of the frame without taking into account any additional linked reference frames. In the case of a “basic” reference frame, the current location is equal to the contents of *refframe_object.Loc.PosWrtRef*. In the case of a dynamic reference frame, such as a pallet, the current position and orientation is computed based upon the object properties, e.g. nominal location, current row, column and layer numbers.

Examples

```
Dim refl As New RefFrame          ' Also allocates Loc
Dim dum As New Location
refl.Loc.XYZ(100,90,-80,0,0,45)   ' Define base frame
Console.WriteLine(refl.PosWrtRef(dum).X) ' Displays 100
Console.WriteLine(refl.PosWrtRef(dum).Y) ' Displays 90
Console.WriteLine(refl.PosWrtRef(dum).Z) ' Displays -80
```

See Also

[RefFrame Class](#) | [refframe_object.Pos](#)

refframe_object.Type Property

Sets and gets the **Integer Type** of a **RefFrame Object**, which indicates if the object is a basic type or one of the special types of reference frames.

```
refframe_object.Type = <new_Integer_value>
-or-
...refframe_object.Type
```

Prerequisites

None

Parameters

None

Remarks

There are several different types of reference frames that can be represented by a *refframe_object*. The **Type** property indicates which type of reference frame is stored in a specific object. The possible values for the **Type** property are as follows:

Type Value	Description
0	Basic RefFrame that simply stores the position and orientation of the reference frame in the Loc Location .
1	Pallet RefFrame that defines a one, two or three-dimensional rectangular grid of positions that are sequentially indexed.

For all reference frames, there are a few common properties that are always defined and accessible. These common properties include the **Type**, **Loc**, **Pos** and **PosWrtRef**. In addition, specific types of reference frames may have additional properties and methods that are only meaningful for a specific type of *refframe_object*. For example, a pallet reference frame has a **PalletOrder** property that is only relevant for that type of frame.

In general, if you attempt to access a property that is not relevant for a *refframe_object*, an error will be generated.

When a “New” **RefFrame** is created, its **Type** is automatically set to 0, i.e. the basic type.

Examples

```
Dim refl As New RefFrame ' Create new reference frame
Dim iType As Integer
iType = refl.Type         ' iType will be set to 0
```

See Also

[RefFrame Class](#)

Robot Class

Robot Class Summary

The following pages provide detailed information on the properties and methods of the global **Robot Class**. This class provides access to the features and status of each robot configured in the system, e.g. the current position of a robot, processes for establishing the position reference for each axes of each robot, functions for forcing an in-process motion to decelerate to a halt, methods for setting and getting the robot's base and tool offsets, etc.

The most important operations of the **Robot Class** are to associate a specific robot with a specific thread and to grant exclusive control of a robot to a thread. Most read-only robot operations require that a statement either explicitly specify a robot or have a previously **Selected** robot. For example, to read the current position of a robot, the **Selected** robot will be accessed if no robot is specified. More importantly, in order to control or move a robot, a thread must first be **Attached** to a robot in order to gain exclusive access to it.

As is standard in GPL, conversions between different arithmetic types, e.g. **Integer**, **Single**, **Double**, are automatically performed as required. So, for numeric properties and methods of the **Robot Class**, it is not necessary to have different variations of these members to deal with the different possible mixes of input parameter data types. Also, as appropriate, the properties and methods generally produce results that are formatted as **Double**'s. These results will automatically be converted to smaller data types as necessary, e.g. **Double** -> **Integer**, and will not generate an error so long as numeric overflow does not occur.

The table below briefly summarizes the properties and methods that are described in greater detail in the following sections.

Member	Type	Description
Robot.Attached	Property	Sets and gets the number of the robot that is exclusively controlled by a thread.
Robot.Base	Property	Sets and gets the position and orientation offset for the base of the robot.
Robot.Custom	Property	Sets and gets elements of a parameter array whose interpretation is specific to each kinematic module.
Robot.DefLinComp	Method	Defines internal table of motor encoder "Linearity compensation" correction values that are automatically applied to encoder values.
Robot.Dest	Property	Returns a Cartesian Location whose value is equal to the originally planned final destination of the previously executed motion.
Robot.DestAngles	Property	Returns an Angles Location whose value is equal to the originally planned final destination of the previously executed motion.
Robot.Home	Method	Homes the Attached robot to establish the reference positions for each axes.

<u>Robot.HomeAll</u>	Method	Homes all robots to establish the reference positions for each of their axes.
<u>Robot.LastProfile</u>	Property	Returns a Profile Object whose properties are equal to those of the currently executing motion or the last executed motion if no motion is active.
<u>Robot.RapidDecel</u>	Property	Sets the Boolean flag that forces any in-process motion for a robot to be rapidly decelerated to a stop.
<u>Robot.RestartBase</u>	Property	Gets the position and orientation offset for the base of the robot that was set when the controller was restarted.
<u>Robot.RestartTool</u>	Property	Gets the position and orientation offset for the tool or gripper of the robot that was set when the controller was restarted.
<u>Robot.Selected</u>	Property	Sets and gets the default robot number to be used when accessing a specific robot.
<u>Robot.Source</u>	Property	Returns a Cartesian Location whose value is equal to the initial position and orientation of the previously executed motion.
<u>Robot.SourceAngles</u>	Property	Returns an Angles Location whose value is equal to the initial axes positions of the previously executed motion.
<u>Robot.Tool</u>	Property	Sets and gets the position and orientation offset for the tool or gripper of the robot.
<u>Robot.TrajState</u>	Property	Gets an Integer that indicates the current state of the Trajectory Generator for a given robot.
<u>Robot.Where</u>	Property	Gets a Cartesian Location whose value indicates the current position and orientation of a robot.
<u>Robot.WhereAngles</u>	Property	Gets an Angles Location whose value indicates the current position of each axes of a robot.

Robot.Attached Property

Sets and gets the number of the robot that is exclusively controlled by a thread.

```
Robot.Attached = <robot_number>
-or-
... Robot.Attached
```

Prerequisites

None

Parameters

None

Remarks

In order to ensure that a robot receives a consistent set of motion commands, a robot must be **Attached** before any motion commands can be issued by a thread and only a single thread can be **Attached** to a robot at any given time.

While a robot is **Attached** by a thread, other threads are still permitted to read certain properties of the robot, such as the current robot position and trajectory state. Also, other threads are able to alter the robots operation in ways that make sense. For example, any thread can disable high power, signal a Soft or Hard E-Stop, or force a robot to rapidly decelerate.

The **Attached** robot number is an **Integer** that ranges from 1 to N. If the **Attached** property is set to 0, any robot attached to the thread is released (un-**Attached**).

When a robot is **Attached**, the system forces the **Selected** property to be equal to the **Attached** value.

Typically, if a project is intended to control a robot, the GPL software development environment can be configured to automatically generate the statements to ensure the robot will be **Attached** at the start of program execution and un-**Attached** when the program is terminated or pauses execution.

Examples

```
Robot.Attached = 1      ' We now have exclusive control of robot #1
Robot.Attached = 0      ' This is how you give up control
```

See Also

[Robot Class](#) | [Robot .Selected](#)

Robot.Base Property

Sets and gets the position and orientation offset for the base of the robot.

```
Robot.Base = <Cartesian_location>
-or-
... Robot.Base ( robot )
```

Prerequisites

- For the set operation, the robot must be attached to the current thread.
- For the set operation, the **Location** must be of the Cartesian type.

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

This property permits a project to either set or retrieve the Cartesian **Location Object** that defines the position and orientation offset from the base of the robot to the origin of the World coordinate system.

The **Base** definition is beneficial if you create an application using Cartesian **Locations** and the base of the robot is subsequently shifted slightly. By adjusting the position of the **Base** definition, a project can automatically correct all of the joint angle positions that will be computed from Cartesian **Locations**.

For computational reasons, some robot kinematic modules may not support the **Base** property. Also, as a computational efficiency, the value of **Base** can only contain a positional offset in X, Y, and Z and a rotation about the Z-axis. That is, the Euler angles for the Base must always be "X,Y,Z,Yaw,0,0".

For most applications, the Base value is not used and its value is set to "0,0,0,0,0,0".

Once the **Robot.Base** has been set, these dimensions remain in effect until the **Base** property is set again or the controller is powered down and restarted. As a convenience, when the controller is restarted, a "Restart Base " definition is automatically put into effect based upon the values of "Base set at restart" (DataID 16052).

Changing the robot's **Base** instantaneously changes where the system thinks that the robot's Cartesian set point is located. So, if the robot is in motion when a thread attempts to set the **Base**, GPL automatically waits until the motion is completed before executing this instruction.

GPL Dictionary Pages

Examples

```
Dim base As New Location
Robot.Attached = 1
base.XYZ(10, 0, 0)           ' Move base by 10mm in X
Robot.Base = base
Console.WriteLine(Robot.Base().X) ' Outputs a value of 10
```

See Also

[Robot Class](#) | [Robot.RestartBase](#)

Robot.Custom Property

Sets and gets elements of a parameter array whose interpretation is specific to each kinematic module.

```
Robot.Custom ( index ) = <New_value>
-or-
... Robot.Custom ( robot, index )
```

Prerequisites

-
- For the set operation, the robot must be attached to the current thread.
- For kinematic modules that do not use the array of custom kinematic parameters, setting or reading these parameters has no effect on the operation of the associated robot.

Parameters

index

An optional numeric expression that specifies the element of the custom kinematic parameter array (1-5) that is accessed. If this value is 1 or unspecified, the first element will be accessed.

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

Selected kinematic modules have special runtime parameters that alter their behavior in a non-standard fashion. For example, the "Dual RPR" robot has two arms and two sets of grippers that can be moved. At any given time, only one of the arms and one of the grippers can be factored into the computation of the Cartesian position and orientation of the robot. The "custom kinematic parameters" are utilized by this kinematic module at runtime to specify which of the two arms is logically considered part of the robot.

In some instances, setting a parameter may cause the executing thread to pause waiting for the attached robot to complete its current motion. This side effect and other similar actions are controlled by the specific kinematic module type.

For a description of how these parameters are utilized in a specific robot and their side effects, please consult the documentation on the Kinematic Robot Modules.

Examples

GPL Dictionary Pages

```
Robot.Attached = 1  
Robot.Custom(1) = 1      ' Set custom parameter value
```

See Also

[Robot Class](#)

Robot.DefLinComp Method

Defines internal table of motor encoder "Linearity compensation" correction values that are automatically applied to encoder values.

Robot.DefLinComp (*robot*, *motor*, *enc_start*, *enc_step*, *num_cor*, *cor*)

Prerequisites

- Motor linear compensation must be permitted for the *robot*.
- Motor linear compensation must be enabled.

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

motor

A required numeric expression that specifies the motor to compensate (1-n).

enc_start

A required numeric expression that specifies the first (and lowest) encoder count to be corrected.

enc_step

A required numeric expression that specifies the step size in encoder counts between successive encoder correction values. Must be greater than 0 and can be a fractional value.

num_cor

A required numeric expression that specifies the number of encoder correction values that are defined in the *cor* array (1-n). The number of values is only limited by the available system memory. Increasing the number of correction values and decreasing the step size improves the compensation and only effects memory, not execution time.

cor

A required array of double precision values that specifies the correction in encoder counts at each sequential encoder position. The corrections can include fractional encoder counts. Positive values indicate that the

encoder should be reading a higher value and negative numbers indicate the encoder reading should be lower.

Remarks

This method creates and defines an internal table of encoder correction values for the specified *motor* of a *robot*. These corrections are automatically applied to each motor command and to each encoder reading. This technique permits repeatable position errors to be corrected to yield more linear and accurate axis positioning. In between correction values, the corrections are interpolated. Outside of the correction range, the raw encoder value is utilized.

As soon as this method creates and initializes the correction data, it is immediately put into effect.

As a convenience, this instruction can be executed even when robot power is enabled. So long as the corrections are small, this will result in a small instantaneous motion of the motor.



WARNING: When first trying a new compensation data set, motor power should be disabled to avoid any sudden, high speed motor motions.

Correction data sets can be created for any motor of the robot that you wish to compensate. It is not necessary to create a correction table for all motors. Correction tables stay in effect until they are over-written or the controller is restarted.

Please see the "Motor Linearity Compensation" section in the *Controller Software > Software Setup > Selected Setup Details and Procedures* chapter of the *Precise Documentation Library* for information on creating correction data sets and for more information on this technique.

Examples

```
Dim cor(2) As Double
cor(0) = 0
cor(1) = -18           ' First step is too short
cor(2) = 5.3          ' Second step is too long
Robot.DefLinComp(1, 1, 5000, 1000, 3, cor)
```

See Also

[Robot Class](#)

Robot.Dest Property

Returns a Cartesian **Location** whose value is equal to the originally planned final destination of the previously executed motion.

...Robot.Dest (*robot*)

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

This property can be used for retrieving the Cartesian position and orientation that was originally planned as the final destination for the previously executed motion. The previously executed motion can still be in progress or could have already stopped executing when this property is accessed.

This information is useful since it is not altered even if the previous motion was prematurely terminated due to a **RapidDecel**, E-Stop, or other condition. Consequently, this data can be utilized to complete the previous motion.

Note that performing a motion that is relative to the **Dest Location** is not the same as performing a **Move.Rel** instruction. The **Move.Rel** instruction will perform an incremental motion relative to wherever the robot's final position was at the conclusion of the previous motion. Moving relative to the **Dest Location** moves with respect to where the previous motion was planned to terminate.

The following table describes the data returned in the **Location** value.

Property	Returned Location Object value
Type	Cartesian Location
PosWrtRef	Set equal to planned Cartesian position and orientation destination of the previous motion.
RefFrame	Always Null
Config	Configuration bits for the planned destination of the previous motion.
ZClearance	1.0e32 to indicate not initialized
All other properties	Always zeroed.

GPL Dictionary Pages

Examples

```
Dim DestLoc As Location
DestLoc = Robot.Dest()      ' Reads planned motion destination
```

See Also

[Robot Class](#) | [Robot.DestAngles](#) | [Robot.LastProfile](#) | [Robot.Source](#) | [Robot.SourceAngles](#)

Robot.DestAngles Property

Returns an Angles **Location** whose value is equal to the originally planned final destination of the previously executed motion.

...Robot.DestAngles (*robot*)

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

This property can be used for retrieving the axes positions that were originally planned as the final destination for the previously executed motion. The previously executed motion can still be in progress or could have already stopped executing when this property is accessed.

This information is useful since it is not altered even if the previous motion was prematurely terminated due to a **RapidDecel**, E-Stop, or other condition. Consequently, this data can be utilized to complete the previous motion.

Note that performing a motion that is relative to the **DestAngles Location** is not the same as performing a **Move.Rel** instruction. The **Move.Rel** instruction will perform an incremental motion relative to wherever the robot's final position was at the conclusion of the previous motion. Moving relative to the **DestAngles Location** moves with respect to where the previous motion was planned to terminate.

The following table describes the data returned in the **Location** value.

Property	Returned Location Object value
Type	Angles Location
Angles	Set equal to planned axes position destinations of the previous motion.
RefFrame	Always Null
ZClearance	1.0e32 to indicate not initialized
All other properties	Always zeroed.

GPL Dictionary Pages

Examples

```
Dim DestLoc As Location  
DestLoc = Robot.DestAngles() ' Reads planned motion destination
```

See Also

[Robot Class](#) | [Robot.Dest](#) | [Robot.LastProfile](#) | [Robot.Source](#) | [Robot.SourceAngles](#)

Robot.Home Method

Homes the **Attached** robot to establish the reference positions for each axes.

Robot.Home

Prerequisites

- High power to the robot must be enabled.
- A robot must be **Attached** by the thread.

Parameters

None

Remarks

This method allows a robot to be homed via a program statement. The homing process re-establishes the reference (e.g. zero) position for each axis of the robot. This enables the robot to reliably move to the same positions after each time that the controller is restarted even when the robot is equipped with incremental, not absolute encoders.

The axes homing sequence must be executed once for each axis after the system is restarted and prior to executing any position controlled motions. Often, the homing process is manually initiated via the operator control panel.

There are many different methods that can be employed to home an axis, e.g. home to hard stop, home to limit switch, home to home switch, etc. The specific method for each axis and the parameters for each method are pre-configured by the robot manufacturer. The **Home** method simply executes the pre-configured method for the robot **Attached** to the thread.

Examples

```
Robot.Attach(1)      ' Attach a robot to the thread
Robot.Home()         ' Home the Attached robot
```

See Also

[Robot Class](#) | [Robot.HomeAll](#)

Robot.HomeAll Method

Homes all robots to establish the reference positions for each of their axes.

Robot.HomeAll

Prerequisites

- High power must be enabled.
- No robot can be **Attached** by a different thread.

Parameters

None

Remarks

This method allows all robots to be homed via a program statement. This homing process re-establishes the reference (e.g. zero) position for each axis of each robot. This enables the robots to reliably move to the same positions after each time that the controller is restarted even when the robots are equipped with incremental, not absolute encoders.

The axes homing sequence must be executed once for each axis of each robot after the system is restarted and prior to executing a robot in position controlled mode. Often, the homing process is manually initiated via the operator control panel.

There are many different methods that can be employed to home an axis, e.g. home to hard stop, home to limit switch, home to home switch, etc. The specific method for each axis and the parameters for each method are pre-configured by the robot manufacturer. The **HomeAll** method simply executes the pre-configured method for all robots.

Examples

```
Robot.HomeAll()           ' Execute home sequence for all robots
```

See Also

[Robot Class](#) | [Robot.Home](#)

Robot.LastProfile Property

Returns a **Profile Object** whose properties are equal to those of the currently executing motion or the last executed motion if no motion is active.

```
...Robot.LastProfile ( robot )
```

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

This property extracts a copy of the motion **Profile** parameters that were specified for the currently executing motion of a **Robot** or the last motion if no motion is now in progress. The extracted values are returned in a **Profile Object**.

If the previous motion was interrupted due to an error, this property, in combination with the **Dest** or **DestAngles** properties, is very useful for retrying the motion.

Examples

```
Dim Profile1 As Profile
Profile1 = Robot.LastProfile()      ' Reads last Profile utilized
```

See Also

[Robot Class](#) | [Robot.Dest](#) | [Robot.DestAngles](#)

Robot.RapidDecel Property

Sets the internal **Boolean** flag that forces any in-process motion for a robot to be rapidly decelerated to a stop.

Robot.RapidDecel (*robot*)

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

Setting the **RapidDecel** flag immediately initiates a rapid deceleration of any motion in progress for the specified robot. At the conclusion of the deceleration, no error is signaled and program execution continues un-interrupted. The motion will, however, have been stopped at a location different from the original plan. If the robot was not in motion, setting this flag is ignored. At the start of the next motion, the **RapidDecel** flag is automatically reset.

The **RapidDecel** feature can be used to stop motions prematurely due to an external signal, such as tripping a switch, touch sensor, or force sensor. Since these are expected events, program processing is not halted.

In that this flag stops any in-process motion, it is similar in effect to the Soft E-Stop, Hard E-Stop, and Disable Power functions. However, those functions are typically used to stop all robots simultaneously when an unexpected event occurs and they therefore generate error conditions.

Examples

```
Robot.RapidDecel()           ' Triggers a rapid decel of Selected robot
```

See Also

[Robot Class](#) | [Controller.PowerEnabled](#) | [Controller.SoftEstop](#)

Robot.RestartBase Property

Gets the position and orientation offset for the base of the robot that was set when the controller was restarted.

```
... Robot.RestartBase ( robot )
```

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

As a convenience, when the controller is restarted, the "base" for each robot is automatically set equal to the position and orientation offset defined by its "Base set at restart" (DataID 16052) value. Since many applications utilize the same base offset each day, this ensures that the **Base** dimensions are correctly set when the system is restarted.

This property returns a Cartesian **Location** value that is equal to the **Base** dimensions that were set the last time that the system was restarted.

Once set, these **Base** dimensions can be easily modified using the **Robot.Base** property. See that property for additional information on the use and benefits of the **Base** property.

Examples

```
Robot.Attached = 1  
Robot.Base = Robot.RestartBase() ' Set base back to default
```

See Also

[Robot Class](#) | [Robot.Base](#)

Robot.RestartTool Property

Gets the position and orientation offset for the tool or gripper of the robot that was set when the controller was restarted.

```
... Robot.RestartTool ( robot )
```

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

As a convenience, when the controller is restarted, the tool for each robot is automatically set equal to the position and orientation offset defined by its "Tool set at restart" (DataID 16051) value. Since many applications utilize the same tool or gripper each day, this ensures that the **Tool** dimensions are correctly set when the system is restarted.

This property returns a Cartesian **Location** value that is equal to the **Tool** dimensions that were set the last time that the system was restarted.

Once set, these **Tool** dimensions can be easily modified using the **Robot.Tool** property. See that property for additional information on the use and benefits of the **Tool** property.

Examples

```
Robot.Attached = 1  
Robot.Tool = Robot.RestartTool() ' Set tool back to default
```

See Also

[Robot Class](#) | [Robot.Tool](#)

Robot.Selected Property

Sets and gets the default robot number to be used when accessing a specific robot.

```
Robot.Selected = <robot_number>
-or-
... Robot.Selected
```

Prerequisites

None

Parameters

None

Remarks

This property allows a thread to set its default robot number. Most of the properties and methods that reference a robot allow the robot number to be explicitly specified or to be unspecified and utilize the **Selected** robot number by default. However, there are some methods, such as the *location_object.Here*, that always access the **Selected** robot.

The **Selected** robot number is an **Integer** that ranges from 1 to N.

When a robot is **Attached**, the system forces the **Selected** property to be equal to the **Attached** value.

Examples

```
Dim iRobot As Integer
Robot.Selected = 1      ' Robot #1 is now Selected
iRobot = Robot.Selected ' iRobot will be set to 1
```

See Also

[Robot Class](#) | [Robot.Attached](#)

Robot.Source Property

Returns a Cartesian **Location** whose value is equal to the starting position and orientation of the previously executed motion.

...Robot.Source (*robot*)

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

This property can be used for retrieving the Cartesian position and orientation for the starting position of the previously executed motion. The previously executed motion can still be in progress or could have already stopped executing when this property is accessed.

The value returned by this property does not reflect any blending that may have occurred if the motion was executed as part of a continuous path. That is, the value returned will be the same whether or not continuous path was in effect.

This information is very useful when accessed in combination with the **Dest Location** to reconstruct the previously planned motion. For example, this is beneficial for moving the robot's tool back onto the previous path if the previous motion was prematurely terminated via a **RapidDecel**.

The following table describes the data returned in the **Location** value.

Property	Returned Location Object value
Type	Cartesian Location
PosWrtRef	Set equal to starting Cartesian position and orientation of the previous motion.
RefFrame	Always Null
Config	Configuration bits for the start of the previous motion.
ZClearance	1.0e32 to indicate not initialized
All other properties	Always zeroed.

Examples

```
Dim SourceLoc As Location
SourceLoc = Robot.Source() ' Reads starting motion location
```

See Also

[Robot Class](#) | [Robot.Dest](#) | [Robot.DestAngles](#) | [Robot.LastProfile](#) | [Robot.SourceAngles](#)

Robot.SourceAngles Property

Returns an Angles **Location** whose value is equal to the starting axes positions of the previously executed motion.

...Robot.SourceAngles (*robot*)

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

This property can be used for retrieving the axes positions that represent the starting position of the previously executed motion. The previously executed motion can still be in progress or could have already stopped executing when this property is accessed.

The value returned by this property does not reflect any blending that may have occurred if the motion was executed as part of a continuous path. That is, the value returned will be the same whether or not continuous path was in effect.

This information is very useful when accessed in combination with the **DestAngles Location** to reconstruct the previously planned motion. For example, this is beneficial for moving the robot's axes back onto the previous path if the previous motion was prematurely terminated via a **RapidDecel**.

The following table describes the data returned in the **Location** value.

Property	Returned Location Object value
Type	Angles Location
Angles	Set equal to initial axes positions of the previous motion.
RefFrame	Always Null
ZClearance	1.0e32 to indicate not initialized
All other properties	Always zeroed.

Examples

```
Dim SourceLoc As Location
SourceLoc = Robot.SourceAngles() ' Reads initial motion position
```

See Also

[Robot Class](#) | [Robot.Dest](#) | [Robot.DestAngles](#) | [Robot.LastProfile](#) | [Robot.Source](#)

Robot.Tool Property

Sets and gets the position and orientation offset for the tool or gripper of the robot.

```
Robot.Tool = <Cartesian_location>
-or-
... Robot.Tool ( robot )
```

Prerequisites

- For the set operation, the robot must either be attached to the current thread or must not be attached to any thread.
- For the set operation, the **Location** must be of the Cartesian type.

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

This property permits a project to either set or retrieve the Cartesian **Location Object** that defines the position and orientation offset from the last axis of the robot to the center point of the robot's gripper (or tool).

The **Tool** definition is particularly beneficial for robots that can change the orientation of the gripper. When the tool center point is properly defined and the system is instructed to move along a straight-line path, the tool center point will move along a straight line even if the orientation of the gripper is simultaneously changed. Also, in Jog-Tool control mode, the operator can easily rotate the tool center point while maintaining the same position.

For the majority of simple grippers, the gripper dimensions consist of just an offset along the Z-axis of the robot with no change in orientation. This corresponds to an **Location XYZ** specification of "0,0,tool_length,0,0,0".

Once the **Robot.Tool** has been set, these dimensions remain in effect until the **Tool** property is set again or the controller is powered down and restarted. As a convenience, when the controller is restarted, a "Restart Tool" definition is automatically put into effect based upon the values of "Tool set at restart" (DataID 16051).

Changing the **Tool** dimensions instantaneously changes where the system thinks that the robot's Cartesian set point is located. So, if the robot is in motion when a thread attempts to set the **Tool**, GPL automatically waits until the motion is completed before executing this instruction.

Examples

```
Dim tool As New Location
Robot.Attached = 1
tool.XYZ(0, 0, 100)           ' Simple tool with 100mm length
Robot.Tool = tool
Console.WriteLine(Robot.Tool().Z) ' Outputs a value of 100
```

See Also

[Robot Class](#) | [Robot.RestartTool](#)

Robot.TrajState Property

Returns an **Integer** that indicates the current state of the Trajectory Generator for a given robot.

...Robot.TrajState (*robot*)

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

The Trajectory Generator state defines whether a trajectory is currently being evaluated for the specified robot and, if so, what portion of the trajectory is being generated. This value can be utilized to determine if a trajectory is being ramped up to its maximum speed, being ramped, waiting for final position errors to be nulled, sitting idle, performing a special control mode, etc.

The possible values returned by this property are presented in the following table:

TrajState	Description
0	Halted, Trajectory Generator not being executed and no robot attached
1	Idle, Trajectory Generator ready to service commands but no motion in progress.
2	Position controlled mode, accelerating up to maximum speed.
3	Position controlled mode, moving at constant velocity.
4	Position controlled mode, blending two motions together.
5	Position controlled mode, decelerating robot to a stop.
8	Velocity controlled mode
9	Special motor speed control mode
10	Jog (manual) control mode
11	External trajectory control, special mode
15	Motion terminated, waiting for final position to satisfy InRange criteria.

Examples

```
Dim istate As Integer
istate = Robot.TrajState() ' Reads current trajectory state
```

See Also

[Robot Class](#)

Robot.Where Property

Returns a Cartesian **Location** whose value is equal to the current position and orientation of a robot.

...Robot.Where (*robot*)

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

This property returns the current position and orientation of a robot in a Cartesian **Location**. This position and orientation automatically take into account both the robot's **Base** and **Tool** offsets.

The returned value is computed by reading the instantaneous values of each motor's encoder and converting these values into an equivalent Cartesian position and orientation. These sampled values are usually slightly different than the commanded axes set point positions due to servo tracking errors and small positional errors.

The conversion to Cartesian coordinates make use of the optional Kinematic module for the selected robot.

Note, if you wish to update the position and orientation of a **Location** variable, it is often better to utilize the *location_object.Here* method rather than simply assigning the **Where Location** to the variable. The **Here** method preserves the other properties of the **Location** variable and will automatically take into account any reference frame (**RefFrame**).

The following table describes the data returned in the **Location** value.

Property	Returned Location Object value
Type	Cartesian Location
PosWrtRef	Set equal to current Cartesian position and orientation of a robot.
RefFrame	Always Null
Config	Configuration bits for the current robot position and orientation.
ZClearance	1.0e32 to indicate not initialized
All other properties	Always zeroed.

Examples

```
Dim RobotPos As Location  
RobotPos = Robot.Where() ' Where is the robot right now?
```

See Also

[Robot Class](#) | [Robot.WhereAngles](#) | [location object.Here](#)

Robot.WhereAngles Property

Returns an Angles **Location** whose value is equal to the current axes positions of a robot.

...Robot.WhereAngles (*robot*)

Prerequisites

None

Parameters

robot

An optional numeric expression that specifies the robot to be accessed (1-n). If this value is 0 or unspecified, the **Selected** robot will be accessed.

Remarks

This property returns the current positions of the axes of a robot in a Angles **Location**.

The returned value is computed by reading the instantaneous values of each motor's encoder and converting these values into equivalent axes positions. These sampled values are usually slightly different than the commanded axes set point positions due to servo tracking errors and small positional errors.

Note, if you wish to update the position of a **Location** variable, it is often better to utilize the *location_object.Here* method rather than simply assigning the **WhereAngles Location** to the variable. The **Here** method preserves the other properties of the **Location** variable.

The following table describes the data returned in the **Location** value.

Property	Returned Location Object value
Type	Angles Location
Angles	Set equal to current position of each axes of a robot.
RefFrame	Always Null
Config	Configuration bits for the current robot position and orientation.
ZClearance	1.0e32 to indicate not initialized
All other properties	Always zeroed.

Examples

```
Dim RobotPos As Location
RobotPos = Robot.WhereAngles() ' Where is the robot right now?
```

See Also

[Robot Class](#) | [Robot.Where](#) | [location_object.Here](#)

Signal Class

Signal Class Summary

The following pages provide detailed information on the properties and methods of the global **Signal Class**. This class provides access to the simple hardware interfacing features of the Guidance controller, such as the digital and analog input and output (I/O). These common interfaces allow a GPL program to coordinate its actions with those of other devices.

Using the digital I/O, programs can employ semaphores to interlock their execution with other equipment in the work cell such as feeders or processing machines. Using the analog I/O, programs can sample the values of simple sensors such as force or temperature sensors to alter the sequence of program execution.

As is standard in GPL, conversions between different arithmetic types, e.g. **Boolean**, **Integer**, **Single**, **Double**, are automatically performed as required. So, for numeric properties and methods of the **Signal Class**, it is not necessary to have different variations of these members to deal with the different possible mixes of input parameter data types. Also, as appropriate, the properties and methods generally produce results that are formatted as **Double**'s. These results will automatically be converted to smaller data types as necessary, e.g. **Double** -> **Integer**, and will not generate an error so long as numeric overflow does not occur.

The table below briefly summarizes the properties and methods that are described in greater detail in the following sections.

Member	Type	Description
Signal.AIO	Property	Sets and gets the values of the analog input and output channels.
Signal.DIO	Property	Sets and gets the values of the digital input and output channels.

Signal.AIO Property

Sets and gets the values of the analog input and output channels.

```
Signal.AIO(channel)=<new_value>
-or-
... Signal.AIO(channel)
```

Prerequisites

None

Parameters

channel

A required numeric expression that specifies the analog channel to be accessed. The allocated ranges of channel numbers are as follows:

Channel Type	Minimum number	Max allocated number
Analog outputs	1	10000
Analog inputs	10001	20000

Please consult the hardware specification for your specific version of controller for information on the maximum number of input and output channels available on your system.

Only the value of an output channel can be written. The current values of both input and output channels can be read.

Remarks

At the hardware level, both analog input and analog output signals levels are represented by integer numbers whose ranges are a function of the specific model of your controller. To generalize accessing these devices at the GPL level, analog values are represented by floating point numbers that are scaled, offset, and thresholded relative to the raw hardware values.

In many systems, analog values are configured to range from either +-1.0 or +-100. Please consult the personnel who configured your controller for the applicable ranges of possible analog values.

Examples

```
Dim sensor_reading As Single
sensor_reading = Signal.AIO(10001)  'Sets sensor_reading equal to the
                                     'scaled value of the first analog
                                     'input channel
```

GPL Dictionary Pages

See Also

[Signal Class](#) | [Signal.DIO](#)

Signal.DIO Property

Sets and gets the values of the digital input and output channels.

```
Signal.DIO(channel, count)=<new_value>
-or-
... Signal.DIO(channel, count)
```

Prerequisites

None

Parameters

channel

A required numeric expression that specifies the first digital channel to be accessed. Signal numbers are organized into ranges based on the signal type. Within those ranges, the signals are organized into banks of 96 I/O points. The bank numbers start at 0. A signal number is formed by adding the signal base value to 100 times the bank number.

In a distributed servo network, general digital I/O signals on the slave controllers may be accessed from the master controller by adding 100000 times the slave controller node number to the signal number.

count

An optional numeric expression that specifies the number of successive digital channels to be accessed. The value may range from 1 to 32. If omitted, only a single channel is accessed and the property value is a Boolean.

If specified, the property value is a numeric bit mask. Omitting the *count* parameter is not the same as specifying a *count* of 1.

If multiple channels are specified, all channels within the range *signal* to *signal+count-1* must be valid.

Remarks

When specifying DIO signal (*channel*) numbers, a positive base signal number indicates that the signal is **True** if its logical level is high. A negative base signal number indicates that the signal is **True** if its logical level is low. For example, if the *channel* is 10001, the signal is **True** if the input is at a logic high level. If the *channel* is -10001, the signal is **True** if the input is at a logic low level.

Only an output DIO signal can be written. The current values of both input and output signals can be read.

If *count* is specified, the DIO specified by *channel* corresponds to bit 0 of the property value. *channel*+1 corresponds to bit 1, *channel*+*n* corresponds to bit *n*, where *n* < *count*.

The table below shows the possible signal numbers based on the type and the bank.

Signal Type	Signal Base	Signal Range	Banks
Test	0	0	
General outputs	1	1 + 100*bank 96 + 100*bank	0 = Local outputs, 1-15 = Remote outputs on RIO or MODBUS/TCP modules.
Dedicated outputs	8001	8001 + 100*bank 8096 + 100*bank	0 = Controller outputs, 1-15 = axis outputs.
General inputs	10001	10001 + 100*bank 10096 + 100*bank	0 = Local inputs, 1-15 = Remote inputs on RIO or MODBUS/TCP modules.
Dedicated inputs	18001	18001 + 100*bank 18096 + 100*bank	0 = Controller inputs, 1-15 = axis inputs.
Software I/O	20001	20001 - 20064	Not used
Reserved	21001	21001 - 100000	
Servo Network node n general outputs	100000*n + 13	100000*n + 20	0 = Local outputs only
Servo Network node n general inputs	100000*n + 10001	100000*n + 10012	0 = Local inputs only

The following describes the different type of digital IO signals:

DIO Type	Description
Test	Channel 0 is a special test value that always reads False no matter what value is written to it.
General	These are the “user” DIO signals that are provided in the controller or remote I/O boards. They do not have a predefined use and can be freely employed. In some cases, general DIO may be configured to serve as dedicated IO. For example, a general DIO can be configured as a joint over-travel limit.
Dedicated	The dedicated DIO are pre-defined to fixed machine control functions such as a home sensor. Some of these signals are assigned to specific pins. However, others can be mapped to General DIO pins.
Software	These “soft” IO do not drive or read actual hardware output or input signals. They can be used as semaphores between threads or in place of hardware DIO for testing control algorithms.

Please consult the hardware specification for your specific version of controller for information on the maximum number of input and output channels of each type available on your system.

Examples

`Dim semaphore As Boolean`

Signal.DIO (20001) = True	' Sets soft signal 20001 to True
semaphore = Signal.DIO (-20001)	' Will set semaphore value to False
Signal.DIO (20001) = 4	' Sets soft signal 20001 to True
	' since 4 is non-zero.
Signal.DIO (20001, 1) = 4	' Sets soft signal 20001 to False
Signal.DIO (20001, 3) = 4	' Sets soft signal 20001 to False
	' and soft signal 20002 to False
	' and soft signal 20003 to True

See Also

[Signal Class](#) | [Signal.AIO](#)

Statements

Statements Summary

The following pages provide detailed information on the basic statements that are provided as an integral portion of the Guidance Programming Language. These statements provide standard functionality found in any programming language such as control structures, variable declarations, subroutine and function calls, etc. As much as possible, these statements have been modeled after standard instructions provide by other variants of the Basic Programming Language.

The table below briefly summarizes the statements that are described in greater detail in the following sections.

Statement	Description
Call	Transfers control to a procedure and ignores its return value.
Class	Begins a Class definition.
Const	Declares a read-only variable for use in a procedure.
Dim	Declares a variable for use in a procedure.
Do...Loop	Bounds a block of instructions that are repeatedly executed so long as a specified expression evaluates to True or until the expression value becomes True .
Else, Elseif	Used within an If...Then...Else...End If series of statements to conditionally execute alternative blocks of instructions.
End	Marks the end of a control structure or major project element such as a program or function.
Exit	Terminates the execution of a block of instructions within the innermost control structure of a specified type or a procedure.
For...Next	Bounds a block of instructions that are repeatedly executed a specified number of times.
Function	Begins a user-defined function procedure.
Get	Begins a Get procedure block within a Property procedure definition.
Goto	Performs an unconditional branch and continues execution at a specified labeled instruction.
If...Then...Else...End	Conditionally executes a block of embedded statements based upon the value of an expression.
Loop	Marks the end of a Do...Loop block of instructions and in some instances also specifies the loop termination condition.
Module	Begins a user-defined module section. All variable definitions and procedures must be inside a Module or Class definition.
Next	Marks the end of a For...Next block of instructions.
Property	Begins a user-defined Property procedure.
ReDim	Increases or decreases an array size by changing the array's upper bounds.
Return	Causes a user-defined procedure to return control to the calling procedure and optionally return a value.
Set	Begins a Set procedure block within a Property procedure definition.
Sub	Begins a user-defined subroutine procedure.
While...End While	Bounds a block of instructions that are repeatedly executed so long as a specified expression evaluates to True .

Call Statement

This statement transfers control to procedure, and ignores its return value.

```
Call procedure_name([argument_list])
-or-
Call class_name.procedure_name([argument_list])
-or-
Call object_name.procedure_name([argument_list])
```

Prerequisites

None

Parameters

procedure_name

The name of procedure to be called. This procedure can be either user-defined or built-in. It can be a function (Function), a subroutine(Sub) or a method of a built-in class.

class_name

The name of a built-in class of which *procedure_name* is a member.

object_name

The name of a object that is an instance of a built-in class of which *procedure_name* is a member

argument_list

A list of argument values that are passed to the procedure. The *argument_list* may be empty, or may be a list of argument values, separated by “,”, that correspond to the arguments in the called procedure.

argument, argument, argument

The type and number of arguments must match the parameters in the declaration of the called procedure. For a ByVal parameter, the argument can be any expression of the matching type. For a ByRef parameter, the argument must be a variable of the matching type.

Remarks

When a procedure is called, the current procedure is suspended until the called procedure exits. Some procedures (e.g. Function procedures) can return a value. The Call statement does not allow the returned value to be accessed.

The Call statement is optional. It can be omitted and the procedure_name specified as the first item in the statement.

Examples

```
Call my_subroutine(10, 20, 30)
my_subroutine(10, 20, 30)      ' Same as above
Call Move.OneAxis(1, 30, 0, MyProfile)
```

See Also

[Statements](#) | [Function Statements](#) | [Sub Statements](#)

Class Statement

This statement begins a **Class** definition.

```
[ Public | Private ] Class class_name
```

Prerequisites

A **Class** may only be declared at the top level of a file, within a **Module**, or within another **Class**.

Parameters

class_name

The name of the **Class** being defined.

Remarks

A **Class** definition must always end with an **End Class** statement.

If a **Class** is declared **Public**, it can be accessed from outside the **Module** or **Class** in which it is defined. A **Private Class** can only be accessed within the **Module** or **Class** where it is defined. If the **Public** attribute is omitted, the **Class** defaults to **Private**.

Other attributes such as **Friend** or **Protected** are not supported.

Variables, constants, and procedures defined within the **Class** are members of the **Class** and can only be accessed by first specifying the **Class** or an object of the **Class**.

Examples

```
Public Class cc           ' Begin the class
    Public x As Single    ' Variable x is in cc object
    Public y As Single    ' Variable y is in cc object
End Class

Sub test
    Dim obj As New cc    ' Create object of class cc
    obj.x = 2.5           ' Set x value in new object
End Sub
```

See Also

[Statements](#) | [Module Statement](#)

Const Statement

This statement declares a read-only variable for use in a procedure. Use the Dim statement for normal read-write variables.

```
Const variable_name As type = init
-or-
Const variable_name As type = init, variable_name As type = init, ...
```

Prerequisites

A Const statement can only appear inside a procedure or a module.

Parameters

variable_name

The name of the variable to be declared as a constant.

type

The type to be assigned to this variable. The type must be a primitive type.
The primitive type keywords are:

Boolean, Byte, Double, Integer, Short, Single

init

An expression that specifies the initial value for the new variable. It must have a constant value.

Remarks

Only the Const statement can set the value of this variable. Everywhere else, an error occurs if an attempt is made to modify the vlaue.

Examples

```
Const ii As Integer = 10
Const ii As Integer = 10, x As Double = 2.5
```

See Also

[Statements](#) | [Dim statements](#) | [ReDim statements](#)

Dim Statement

This statement declares a variable for use in a class or procedure.

```
[ Public | Private | Shared ] Dim variable_name [, variable_name ...] As [New] type [= [New] init ]
-or-
[ Public | Private | Shared ] Dim variable_name [, variable_name ...] As [New] type [= [New] init ], variable_name [, variable_name ...] As [New] type [= [New] init ], ...
```

Prerequisites

- A **Dim** statement can only appear inside a class, procedure or a module.
- The **Public** and **Private** keywords cannot be used inside a procedure.
- The **Shared** keyword cannot be used at the module level.

Parameters

variable_name

The name of the variable to be declared.

In addition to the name, this field may include an array specification of the form: *variable_name*(*dim_1* [, *dim_2* ...]), where *dim_1* through *dim_4* may be blank or contain an **Integer** constant defining the maximum index of the corresponding array dimension. GPL allows up to four dimensions.

type

The type to be assigned to this variable. The type may be a primitive type, the name of a built-in class, or the name of a user-defined class. The primitive type keywords are:

Boolean, Byte, Double, Integer, Short, Single

If a class name is specified, the variable becomes an object variable.

init

An expression that specifies the initial value for the new variable. It does not need to be a constant.

Remarks

If the **Public** or **Private** keywords are present, the **Dim** keyword may be omitted.

If the **Shared** keyword is specified, only a single copy of this variable is created. It exists for all threads and persists even after the procedure in which it was defined has exited.

All variables declared at the module level are implicitly shared, even though the **Shared** keyword is not allowed.

Shared variables within a procedure can only be accessed from within that procedure, but their values persist and may be accessed by a subsequent procedure call.

If the **Shared** keyword is not specified on a **Dim** statement within a procedure, the variable exists only within that procedure, and it is initialized each time the procedure runs.

If the **Shared** keyword is not specified on a **Dim** statement within a class definition, a separate copy of the variable exists in each object of that class type.

If more than one *variable_name* field is specified, no *init* clause may be specified.

The **New** clause can only be specified for objects. If a **New** keyword is specified immediately following the **As** keyword, no initializer value may be specified.

If no *init* clause is specified, the default value for numeric variables is zero, and for object variables is **Nothing**.

If an *init* clause is specified for a **Shared** variable, the initialization takes place once when the main thread begins execution. If an *init* clause is specified for a non **Shared** variable, the initialization takes place each time the defining procedure is executed, or each time a new object of the class is created.

Examples

```
Dim ii As Integer
Dim ii As Integer = 10
Public ii As Integer = 10
Shared Dim count As Integer
Dim ii, jj As Integer, x As Double
Dim ii As Integer = 10, x As Double = 2.5
Dim start As Location
Dim start As New Location
```

See Also

[Statements](#) | [Const Statement](#) | [ReDim Statement](#)

Do...Loop Statements

These instructions bound a block of instructions that are repeatedly executed so long as a specified expression evaluates to **True** or until the expression value becomes **True**.

```

Do While condition
    [statements]
Loop

-or-

Do Until condition
    [statements]
Loop

-or-

Do
    [statements]
Loop While condition

-or-

Do
    [statements]
Loop Until condition

```

Prerequisites

None

Parameters

condition

Required expression that is interpreted as a **True** or **False** value. Any expression that yields a numeric result can be specified, not just **Boolean** expressions. Any expression that evaluates to $\neq 0$ is interpreted as a **True** condition.

statements

Optional statement or list of statements that are repeatedly executed within the control structure.

Remarks

This control structure either tests a *condition* at the start or the end of a block of *statements* and repeatedly executes the *statements* so long as the *condition* is **True** or until it becomes **True**. It can be used to implement program instruction loops.

For the **Do While** and **Do Until** forms of this control structure, the *condition* test is performed prior to the execution of the *statements*. If the *condition* permits the loop to be executed, the *statements* will be executed once. At the conclusion of the loop, the test is repeated to determine if the *statements* should be executed again. So long as the *condition* permits execution, the *statements* will be repeatedly executed. If not, execution of the *statements* is terminated. In any case, if the *condition* does not permit the execution of the loop on the first test, the *statements* are never executed.

In contrast, for the **Loop While** or **Loop Until** forms of this control structure, the *statements* will always be executed at least one time. For these forms, the test is performed at the conclusion of the execution of the statements. So long as the *condition* permits execution, the *statements* will be repeatedly executed. However, if the *condition* does not permit the execution of the loop on the first test, the *statements* will still have been executed one time.

For all forms of this control structure, when the *condition* test is not satisfied, program execution continues at the first statement following the **Loop** instruction.

If the **While** form of the *condition* test is specified, the *condition* is satisfied and execution of the *statements* is permitted so long as the value of the *condition* is **True**. For the **Until** form of the *condition* test, the *condition* is satisfied and execution is permitted until the condition becomes **True**.

For more complex logic, multiple **Do... Loop** sequences can be nested to an arbitrary depth and can be combined with other nested control structures. For example, a **Do... Loop** can contain an **If...Then...End If** sequence which can in turn contain a **While...End While** sequence.

Execution of the **Do** loop can be terminated by a number of different methods: the *condition* can be set to a value that does not satisfy the test; execution can be explicitly transferred to an instruction outside of the loop, e.g. by the execution of a **GoTo** instruction; or an **Exit Do** instruction can be executed.

When an **Exit Do** statement is encountered, execution of the innermost **Do...Loop** sequence is immediately terminated and execution continues at the instruction following the **Loop** statement. There can be none or several **Exit Do** statements within each **Do** loop.

Examples

```
Dim count As Integer
count = 10
Do                                ' Embedded statements always execute at least once
    If count = 5 Then              ' Prematurely stops Do loop
        Exit Do
    End If
    count -= 1                     ' Same as "count = count-1"
Loop Until count <= 0
```

See Also

[Statements](#) | [For...Next Statements](#) | [GoTo Statements](#) | [If...Then...Else...End If Statements](#) | [While...End While Statements](#)

Else, ElseIf Statements

These instructions are used within an **If...Then...Else...End If** series of statements to conditionally execute alternative blocks of instructions.

```
If condition Then  
    [statements]  
[ Elseif elseif_condition Then  
    [elseif_statements]]  
    :  
[ Elseif elseif_condition Then  
    [elseif_statements]]  
[ Else  
    [else_statements]]  
End If
```

Prerequisites

Can only be specified within an **If...Then...End If** series of statements.

Remarks

Please see the documentation on the **If...Then...Else...End If** Statements for an explanation on the use of the **Else** and **Elseif** instructions.

See Also

[Statements](#) | [If...Then...Else...End If Statement](#)

End Statements

These statements mark the end of control structures and major project elements such as procedures or modules.

```

End Class
-Or-
End Function
-Or-
End Get
-Or-
End If
-Or-
End Module
-Or-
End Property
-Or-
End Set
-Or-
End Sub
-Or-
End While
    
```

Prerequisites

Must always follow and match the type of control structure or procedure that is referenced.

Remarks

Each of the forms of the **End** statement are qualified by the type of control structure or procedure being terminated. Please see the documentation on the related statements and program elements for information on the **End** statements, e.g. see the **While...End While** Statements for information on the **End While** and see **Sub** for information on **End Sub**.

See Also

[Statements](#) | [Function Statement](#) | [If...Then...Else...End If Statements](#) | [Module Statement](#) | [Sub Statement](#) | [While...End While Statements](#)

Exit Statements

These statements terminate the execution of a block of instructions within the innermost control structure of a specified type or a procedure. Execution is continued after the end of the control structure or the call to the procedure.

Exit Do
-or-
Exit For
-or-
Exit Function
-or-
Exit Property
-or-
Exit Sub
-or-
Exit While

Prerequisites

Can only be specified within the control structure or procedure type that is referenced.

Remarks

Each of the forms of the **Exit** statement are qualified by the type of control structure or procedure being terminated. Please see the documentation on the specific statements and program elements for information on the **Exit** statements, e.g. see the **While...End While** Statements for information on the use of **Exit While** and **Sub** for the use of **Exit Sub**.

See Also

[Statements](#) | [Do... Loop Statements](#) | [For...Next Statements](#) | [While...End While Statements](#)

For...Next Statements

These instructions bound a block of instructions that are repeatedly executed a specified number of times.

```
For variable = initial_value To final_value Step increment
    [statements]
Next variable2
```

Prerequisites

None

Parameters

variable

Required control variable that is incremented each loop and whose value determines when looping is to be terminated. The *variable* can be any numeric type, i.e.. **Byte**, **Integer**, **Short**, **Single** or **Double**. Array variables as well as object and structure fields are also permitted. However, object and structure properties are not permitted.

initial_value

Required expression that is evaluated once when the **For** loop is first entered. The *variable* is set to this *initial_value* and has this value at the start of the first pass through the execution of the *statements*.

final_value

Required expression whose value is tested against the *variable* to determine when loop execution is to terminate. This expression is evaluated once when the **For** statement is executed and its value is saved for subsequent tests by the **Next** statement. Therefore, this value will not change once the **For** loop is entered.

increment

Optional expression that determines the amount by which the *variable* is changed each loop and also whether the *variable* is tested for being greater than or less than the *final_value* as the termination condition. This expression is evaluated once when the **For** statement is executed and its value is saved for subsequent tests by the **Next** statement. Therefore, this value will not change once the **For** loop is entered. If this expression is not specified, a step of 1 is assumed.

statements

Optional statement or list of statements that are repeatedly executed during each **For** loop.

variable2

Optional control variable, which if specified, must exactly match the control *variable* in the matching **For** statement. This is only used when the program is compiled (and not at runtime) to ensure that the **Next** and **For** statements match.

Remarks

This control structure loops and repeatedly executes the *statements* a specified number of times (iterations). It can be used to implement program instruction loops and is generally more efficient than the other means of looping.

The **For** statement begins execution by evaluating its arguments and saving their values for future potential use by the matching **Next** statement. It then sets the value of the control *variable* equal to the *initial_value*. If the *variable*'s value does not exceed the *final_value*, then the statements are executed for the first time. If the *variable*'s value does exceed the *final_value*, the *statements* are skipped and execution continues at the first statement beyond the matching **Next**.

If the *statements* are executed, execution proceeds until the **Next** instruction is encountered. When the **Next** statement is executed, the *increment* is added to the *variable* and its value is compared again to the *final_value*. So long as the *final_value* is not exceeded, the *for_loop_statements* are executed again and the process is repeated. Otherwise, execution continues at the statement following the **Next**.

If the *increment* is a positive number, looping terminates when the *variable*'s value is greater than the *final_value*. If negative, looping terminates when the *variable*'s value is less than the *final_value*.

For more complex logic, multiple **For...Next** sequences can be nested to an arbitrary depth and can be combined with other nested control structures. For example, a **For** loop can contain an **If...Then...End If** sequence which can in turn contain another **For...Next** sequence.

Execution of the **For** loop can be terminated by a number of different methods: the *variable*'s value can exceed the *final_value*; execution can be explicitly transferred to an instruction outside of the loop, e.g. by the execution of a **GoTo** instruction; or an **Exit For** instruction can be executed.

When an **Exit For** statement is encountered, execution of the innermost **For...Next** sequence is immediately terminated and execution continues at the instruction following the **Next**. There can be none or several **Exit For** statements within each **For** loop.

Examples

```
Dim count As Integer
For count = 1 To 10      ' Plan to execute 10 loops
```

<code>If count = 5 Then</code>	
<code>Exit For</code>	' Prematurely stops For on 5 th loop
<code>End If</code>	
<code>Next count</code>	' count is optional in the Next

See Also

[Statements](#) | [Do... Loop Statements](#) | [GoTo Statements](#) | [If...Then...Else...End If Statements](#) | [While...End While Statements](#)

Function Statement

This statement begins a user-defined function procedure. It specifies the function return data type and any parameters that are passed when it is called.

[Public | Private | Shared] Function *function_name*(*[parameter_list]*) **As** *type*

Prerequisites

- Procedures cannot be declared inside of other procedures.
- Procedures can only be declared within modules or classes.

Parameters

function_name

The name of function to be defined.

parameter_list

A list of parameters that are passed to the procedure when it is called. Each parameter appears as a locally defined variable and is associated with a value when the procedure is called. The caller must provide arguments that match the number and type of the parameters specified in this statement.

The list may be empty if the function has no parameters. Multiple parameter list elements are separated by ",". Each element has the form:

[ByVal | ByRef] *parameter_name* **As** *type*

parameter_name

The name of the variable associated with this parameter. This name is known only within the procedure being defined.

type

The type of this parameter. The type may be either a primitive type or the name of a built-in class. The primitive type keywords are:

Boolean, Byte, Double, Integer, Short, Single

If a class name is specified, the variable becomes an object variable.

Either **ByVal** or **ByRef** can be specified, but not both. If neither is specified, the default is **ByVal**. A **ByVal** parameter receives a copy of argument value from the caller. The local procedure can change the value without affecting the caller's value. A **ByRef** parameter references the caller's value directly. Any changes to a **ByRef** parameter in the called routine are reflected in the calling routine.

Since object variables always deal with pointers to object values, the called routine can always change an object value, even when passed using a **ByVal** parameter.

type

The type of the value returned by this function. The type may be a primitive type, the name of a built-in class, or the name of a user-defined class. The primitive type keywords are:

Boolean, Byte, Double, Integer, Short, Single

If a class name is specified, the returned type is an object.

Remarks

A **Function** procedure returns a value that can be used within an expression where a value of the proper type is allowed. A **Function** can also be used with a **Call** statement or by itself as a statement when the returned value is not needed.

A **Function** definition must always end with an **End Function** statement.

A **Function** procedure exits when it encounters the **End Function** statement, an **Exit Function** statement, or a **Return** statement.

The returned value of function is specified by assigning a value to a variable named *function_name*, or by a **Return** statement.

If **Public** is specified, this procedure can be called from other modules or classes. Otherwise it can only be called from within the module or class where it is defined.

The **Shared** keyword can only be used within a class definition. If it appears, the **Function** is associated with the entire class rather than with a particular object of that class type.

Examples

```
Function add_function (x As Integer, y As Integer) As Integer
    add_function = x+y
End Function
```

```
a = add_function(4, 5) * 2           ' Variable a gets value 18
```

See Also

[Statements](#) | [End Function Statement](#) | [Exit Function Statement](#) | [Return Statement](#) | [Sub Statement](#)

Get Statement

This statement begins a **Get** procedure block within a **Property** procedure definition.

Get

Prerequisites

-
- This statement can only appear within a **Property** definition.
- The **Property** definition that contains this statement must not specify the **WriteOnly** attribute.

Parameters

None

Remarks

The **Get** procedure block must always end with an **End Get** statement.

When a procedure gets the containing **Property**, the **Get** procedure is executed. It is up to that procedure to retrieve or compute the property value and return it.

The returned value of the **Property** is specified by assigning a value to a variable with the same name as the **Property** or by a **Return** statement.

Examples

```
Class cc
    Private size2 As Integer = 44

    Public ReadOnly Property size As Integer
        Get
            Return size2/2
        End Get
    End Property
End Class

:
Dim obj As New cc
Console.WriteLine(obj.size)      ' Displays value 22
```

See Also

[Statements](#) | [Property Statement](#) | [Set Statement](#)

GoTo Statement

This statement performs an unconditional branch and continues execution at a specified labeled instruction.

GoTo *label*

Prerequisites

None

Parameters

label

Required program instruction label. A *label* must conform to the naming conventions for either be a valid variable name (e.g. label3) or an integer literal (e.g. 1000).

Remarks

This instruction alters the sequence of program statement execution by setting the *label*'ed statement as the next instruction to be executed.

The referenced *label*'ed instruction must be in the same procedure as the **GoTo** instruction and can be on an instruction before or after the **GoTo** instruction. You should not use a **GoTo** to jump from the outside of a control structure (e.g. a **For...Next** or **If...Then...Else...End If**) to within a control structure.

To label an instruction, specify the label name followed by a colon (:) followed by any standard instruction.

In general, **GoTo** instructions can make code difficult to read and debug. So, wherever possible software should be written to make use of the other control structures, e.g. **If...Then...Else...End If**, **While...End While**.

Examples

```
Dim too_big As Boolean, angle As Single
too_big = False
angle = 175.5
If angle > 360 Or angle < -360 Then
    too_big = True
    GoTo Error_Exit
End If
my_routine(angle)

Error_Exit:
```

' An Else clause would be better,
' but this shows how to use GoTo

See Also

[Statements](#) | [Do... Loop Statements](#) | [For...Next Statements](#) | [If...Then...Else...End If Statements](#) | [While...End While Statements](#)

If..Then...Else...End If Statements

A series of statements that conditionally execute a block of embedded statements based upon the value of an expression.

```
If condition Then
  [statements]
[ Elseif elseif_condition Then
  [elseif_statements]
  :
[ Elseif elseif_condition Then
  [elseif_statements]
[ Else
  [else_statements]
End If
```

Prerequisites

None

Parameters

condition

Required expression that is interpreted as a **True** or **False** value. Any expression that yields a numeric result can be specified, not just **Boolean** expressions. Any expression that evaluates to ≤ 0 is interpreted as a **True** condition.

statements

Optional statement or list of statements that are executed if the *condition* evaluates to **True**.

elseif_condition

Expression that is required if an optional **Elseif** clause is specified. Any expression that yields a numeric result can be specified, not just **Boolean** expressions. Any expression that evaluates to ≤ 0 is interpreted as a **True** condition.

elseif_statements

Optional statement or list of statements that are executed if the associated *elseif_condition* evaluates to **True**.

else_statements

Optional statement or list of statements that are executed if the **Else** clause is present and the preceeding *condition* and *elseif_condition* values all test **False**.

Remarks

This control structure tests one or more expressions and conditionally executes at most one block of statements. It can be used to implement simple “either-or” types logic or more complex decisions based upon multiple conditions with multiple possible outcomes.

The **If...Then** statement begins by first testing the value of the *condition*. If the *condition* is **True**, the *statements* are executed, after which, all of the following program instructions are skipped until the closing **End If** is encountered. If the *condition* is **False**, the *statements* are skipped and processing continues at the first **Elseif**, **Else**, or **End If** clause that follows the *statements*. Any *condition* that evaluates to $\neq 0$ will be interpreted as a **True** value.

An arbitrary number of **Elseif** clauses can optionally follow the *statements* and precede the **Else**. If the *condition* is **False**, the first **Elseif** clause is processed by evaluating its *elseif_condition*. If its *elseif_condition* is **True**, its *elseif_statements* are executed after which all of the following program instructions are skipped until the closing **End If** is encountered. If its *elseif_condition* is **False**, its *elseif_statements* are skipped and processing continues at the next **Elseif**, **Else**, or **End If** clause that follows the *elseif_statements*.

An **If...Then** group of statements can contain a single optional **Else** statement. If the *condition* and all optional *elseif_conditions* have tested false, the optional *else_statements* will be executed.

For more complex logic, multiple **If...Then...End If** statements can be nested to an arbitrary depth and can be combined with other nested control structures. For example, a **For** loop can contain an **If...Then...End If** sequence which can in turn contain another **If...Then...End If** sequence.

As an alternative to the multi-line **If...Then...End If** format, for simple cases, this conditional statement can be written all on a single line. In this format, the closing **End If** is not specified.

Examples

```
Dim a As Boolean, b As Integer, c As Single
a = True
b = 20
If a AND (b > 10) Then           ' This condition evaluates to True
    c = 3.14159                 ' This assignment will be executed
Else
    c = 0                       ' This assignment will be skipped
End If
```

See Also

[Statements](#) | [Do... Loop Statements](#) | [For...Next Statements](#) | [GoTo Statements](#) | [While...End While Statements](#)

Loop Statements

These instructions mark the end of a **Do...Loop** block of instructions and in some instances also specify the loop termination condition.

Loop
-or-
Loop Until *condition*
-or-
Loop While *condition*

Prerequisites

Must always follow and match a **Do** statement within a procedure.

Remarks

Please see the documentation on the **Do...Loop** Statements for an explanation of the use of the **Loop** instructions.

See Also

[Statements](#) | [Do... Loop Statement](#)

Module Statement

This statement begins a user-defined module section. All variable definitions and procedures must be inside a **Module** or **Class** definition.

Module *module_name*

Prerequisites

Modules can only be declared at the top-level of a file.

Parameters

module_name

The name of module that is being started.

Remarks

A Module must always end with an **End Module** statement.

A Module contains variable, procedures or class definitions. There can be multiple modules defined in a single file.

All variables, procedures and classes defined within a module can be accessed anywhere in that module. Only **Public** variables, procedures and classes can be accessed outside the module.

Examples

```
Module main_module
    Public Dim Start As Location    ' All modules can access Start
    Private Dim x1 As Location      ' Only this module can access x1

    ' All modules can access add_function

    Public Function add_function (x As Integer,y As Integer) As Integer
        add_function = x+y
    End Function
End Module
```

See Also

[Statements](#) | [Class Statement](#) | [Dim Statement](#) | [End Module Statement](#) | [Function Statement](#) | [Sub Statement](#)

Next Statements

This instruction marks the end of a **For...Next** block of instructions.

Next *variable*

Prerequisites

Must always follow and match a **For** statement within a procedure.

Remarks

Please see the documentation on the **For...Next** Statements for an explanation of the use of the **Next** instruction.

See Also

[Statements](#) | [For...Next Statements](#)

Property Statement

This statement begins a user-defined **Property** procedure. It specifies the return data type and any parameters that are passed when it is called.

```
[ Public | Private | Shared | ReadOnly | WriteOnly ] Property property_name ([  
parameter_list ]) As type)
```

Prerequisites

-
- Properties can only be declared within class definitions.

Parameters

property_name

The name of the **Property** to be defined.

parameter_list

A list of parameters that are passed to the **Property** when it is called. Properties often have an empty parameter list.

Each parameter appears as a locally defined variable and is associated with a value when the procedure is called. The caller must provide arguments that match the number and type of the parameters specified in this statement.

The list may be empty if the **Property** has no parameters. Multiple parameter list elements are separated by ",". Each element has the form:

```
[ ByVal | ByRef ] parameter_name As type
```

parameter_name

The name of the variable associated with this parameter. This name is known only within the procedure being defined.

type

The type of this parameter. The type may be a primitive type, the name of a built-in class, or the name of a user-defined class. The primitive type keywords are:

Boolean, Byte, Double, Integer, Short, Single

If a class name is specified, the variable becomes an object variable.

Either **ByVal** or **ByRef** can be specified, but not both. If neither is specified, the default is **ByVal**. A **ByVal** parameter receives a copy of the argument value from the caller. The local procedure can change the value without affecting the caller's value. A **ByRef** parameter references the caller's value directly. Any changes to a **ByRef** parameter in the called routine are reflected in the calling routine.

Since object variables always deal with pointers to object values, the called routine can always change an object value, even when passed using a **ByVal** parameter.

type

The type of the value returned by this **Property**. The type may be either a primitive type, the name of a built-in class, or the name of a user-defined class. The primitive type keywords are:

Boolean, Byte, Double, Integer, Short, Single

If a class name is specified, the returned type is an object.

Remarks

Property procedures may set a value or get (return) a value.

Property procedures that set a value must include a set procedure block that begins with a **Set** statement and ends with an **End Set** statement. The *property_name* and *parameter_list* may be used on the left-hand side of an assignment statement.

A **Property** procedure that gets a value must include a get procedure block that begins with a **Get** statement and ends with an **End Get** statement. A **Get Property** may be used just like a **Function** within an expression or on the right-hand side of an assignment statement, where a value of the proper type is allowed.

A **Property** definition must always end with an **End Property** statement.

If the **Property** contains only a get procedure, the **ReadOnly** keyword must be specified. If the **Property** contains only a set procedure, the **WriteOnly** keyword must be specified.

A property procedure exits when it encounters the **End Property** statement, an **Exit Property** statement or a **Return** statement.

If **Public** is specified, this procedure can be called from other modules or classes. Otherwise it can only be called from within the class where it is defined.

If the **Shared** keyword appears, the property is associated with the entire class rather than with a particular object of that class type.

Examples

```
Class cc
    Private size_value As Integer

    Public Property size As Integer ' Set size, clip value at 10
        Set (value As Integer)
            If value > 10 Then
                value = 10
            End If
            size_value = value
        End Set
        Get
            Return size_value
        End Get
    End Property

End Class
:
Dim obj As New cc
obj.size = 20 ' Sets size_value
Console.WriteLine(obj.size) ' Displays 10
```

See Also

[Statements](#) | [Get Statement](#) | [Set Statement](#)

ReDim Statement

This statement increases or decreases an array size by changing the array's upper bounds.

```
ReDim variable_name (dim_1 [, dim_2 ...])
```

Prerequisites

The *variable_name* parameter must already be declared to be an array, with the same number of dimensions, in a **Dim**, **Public**, or **Private** statement.

Parameters

variable_name

The name of the array variable that is to have its size changed.

dim_1, *dim_2*, ...

The new upper bounds for each dimension of the array. **ReDim** cannot change the number of dimensions, so the number of dimensions must match the original array declaration.

Remarks

The previous contents of an array are lost when a ReDim statement is executed. The **Preserve** keyword is not supported in GPL.

Examples

```
Dim array(3,4) As Integer
Dim array2() As String

ReDim array(4,6)
ReDim array2(10)
ReDim array2(2,3)           ' Invalid, cannot change # of dimensions
```

See Also

[Statements](#) | [Dim Statements](#)

Return Statement

This statement causes a user-defined procedure to return control to the calling procedure and optionally return a value.

Return [*value*]

Prerequisites

Return can only appear within a procedure.

Parameters

value

The value to be returned to the calling procedure if the current procedure is a Function. The *value* field must be specified in a Function procedure. It must not be specified in a Sub procedure.

Remarks

The current procedure exits when it encounters a Return statement and execution continues with the calling procedure. If there is no calling procedure, the current thread is terminated with success.

In a function procedure, a Return is equivalent to assigning a value to the function-name variable followed by an Exit Function statement.

Examples

```
Function add_function (x As Integer, y As Integer) As Integer
    Return x+y
End Function
```

```
Sub add_sub (x As Integer, y As Integer, ByRef result As Integer)
    result = x+y
    Return
End Sub
```

See Also

[Statements](#) | [Exit Function statement](#) | [Exit Sub statement](#)

Set Statement

This statement begins a **Set** procedure block within a **Property** procedure definition.

Set (*parameter_name As type*)

Prerequisites

-
- This statement can only appear within a **Property** definition.
- The **Property** definition that contains this statement must not specify the **ReadOnly** attribute.

Parameters

parameter_name

The name of the parameter that contains the new value to which the property is being set.

type

The type of the *parameter_name* parameter. This type must be identical to the type of the **Property** that contains the **Set** statement.

Remarks

The **Set** procedure block must always end with an **End Set** statement.

Unlike VB.NET, the clause (*parameter_name As type*) must always be specified.

When a procedure sets the containing **Property**, the new value for the property is copied to the *parameter_name* variable, and the **Set** procedure is executed. It is up to that procedure to use or save the new value as desired.

Examples

```
Class cc
    Private WriteOnly size_value As Integer
    Public Property size As Integer      ' Set size, clip value at 10
        Set (value As Integer)
            If value > 10 Then
                value = 10
            EndIf
            size_value = value
        End Set
    End Property
End Class
:
Dim obj As New cc
obj.size = 20                        ' Sets size_value
```

GPL Dictionary Pages

See Also

[Statements](#) | [Property Statement](#) | [Get Statement](#)

Sub Statement

This statement begins a user-defined subroutine procedure. It specifies any parameters that are passed when it is called.

```
[ Public | Private | Shared ] Sub subroutine_name([parameter_list])
```

Prerequisites

- Procedures cannot be declared inside of other procedures.
- Procedures must be declared within modules or classes.

Parameters

subroutine_name

The name of the subroutine to be defined.

parameter_list

A list of parameters that are passed to the procedure when it is called. Each parameter appears as a locally defined variable and is associated with a value when the procedure is called. The caller must provide arguments that match the number and type of the parameters specified in this statement.

The list may be empty if the subroutine has no parameters. Multiple parameter list elements are separated by “,”. Each element has the form:

```
[ ByVal | ByRef ] parameter_name As type
```

parameter_name

The name of the variable associated with this parameter. This name is known only within the procedure being defined.

type

The type of this parameter. The type may be either a primitive type or the name of a built-in class. The primitive type keywords are:

Boolean, Byte, Double, Integer, Short, Single

If a class name is specified, the variable becomes an object variable.

Either **ByVal** or **ByRef** can be specified, but not both. If neither is specified, the default is **ByVal**. A **ByVal** parameter receives a copy of argument value from the caller. The local procedure can change the value without affecting the caller's value. A **ByRef** parameter references the caller's value directly. Any changes to a **ByRef** parameter in the called routine are reflected in the calling routine.

Since object variables always deal with pointers to object values, the called routine can always change an object value, even when passed using a **ByVal** parameter.

Remarks

A **Sub** procedure does not return a value and cannot be used within an expression. A **Sub** procedure can be used with a **Call** statement or by itself as a statement.

A **Sub** definition must always end with an **End Sub** statement.

A subroutine procedure exits when it encounters the **End Sub** statement, an **Exit Sub** statement, or a **Return** statement.

If **Public** is specified, this procedure can be called from other modules or classes. Otherwise it can only be called from the module or class where it is defined.

The **Shared** keyword can only be used within a class definition. If it appears, the subroutine is associated with the entire class rather than with a particular object of that class type.

Examples

```
Sub add_sub (x As Integer, y As Integer, ByRef result As Integer)
    result = x+y
End Sub
```

```
add_sub(4, 5, a)           ' Variable a gets value 9
```

See Also

[Statements](#) | [End Sub Statement](#) | [Exit Sub Statement](#) | [Return Statement](#) | [Sub Statement](#)

While...End While Statements

These instructions bound a block of instructions that are repeatedly executed so long as a specified expression evaluates to **True**.

```
While condition
  [statements]
End While
```

Prerequisites

None

Parameters

condition

Required expression that is interpreted as a **True** or **False** value. Any expression that yields a numeric result can be specified, not just **Boolean** expressions. Any expression that evaluates to $\neq 0$ is interpreted as a **True** condition.

statements

Optional statement or list of statements that are repeatedly executed so long as the *condition* evaluates to **True**.

Remarks

This control structure tests an expression and repeatedly executes a block of statements. It can be used to implement program instruction loops.

The **While** statement begins execution by testing the value of the *condition*. If the *condition* is **True**, the *statements* are executed. When the **End While** instruction is encountered, the *condition* is tested again. If the *condition* is still **True**, the *statements* are executed once again. This process is repeated until the *condition* tests **False** or the *statements* explicitly execute an instruction that continues execution outside of the loop. If the *condition* ever tests **False**, execution continues at the instruction following the **End While**.

If the condition is **False** when the **While** first begins execution, the *statements* are skipped, in which case, the *statements* are not executed even once.

For more complex logic, multiple **While...End While** sequences can be nested to an arbitrary depth and can be combined with other nested control structures. For example, a **While** loop can contain an **If...Then...End If** sequence which can in turn contain another **While...End While** sequence.

Execution of the **While** loop can be terminated by a number of different methods: the *condition* can be set **False** prior to the execution of the **End While** statement; execution can be explicitly transferred to an instruction outside of the loop, e.g. by the execution of a **GoTo** instruction; or an **Exit While** instruction can be executed.

When an **Exit While** statement is encountered, execution of the innermost **While...End While** sequence is immediately terminated and execution continues at the instruction following the **End While**. There can be none or several **Exit While** statements within each **While** loop.

Examples

```
Dim count As Integer
count = 10
While count > 0           ' This condition initially evaluates to True
    If count = 5 Then
        Exit While        ' Prematurely stops While loop
    End If
    count -= 1             ' Same as "count = count-1"
End While
```

See Also

[Statements](#) | [Do... Loop Statements](#) | [For...Next Statements](#) | [GoTo Statements](#) | [If...Then...Else...End If Statements](#)

Strings

String Summary

The following pages provide detailed information on the properties, methods and functions that are available to assist in manipulating **String** variables. Internally, **Strings** are implemented using much of the same structure and procedures as other built-in Classes. Therefore, in addition to providing classic Basic functions for operating on **Strings**, e.g. **Len**, **String** variable properties and methods are also available for performing many of the same operations.

A number of easy-to-use functions are provided for converting between **String** values and numerical values, e.g. **CStr**, **Cdbl**, **Cint**, **Hex** . Each of these built-in operations is described in the section on Functions.

The table below briefly summarizes the properties and methods of **String** variables that are described in greater detail in the following section.

Member	Type	Description
String.Compare	Method	Compares the values of two Strings in either a case sensitive or case insensitive manner.
string.IndexOf	Method	Searches for an exact match of a substring within the <i>string</i> variable and returns the starting position if found (0-n).
string.Length	Property	Returns the number of characters stored in a String variable.
string.Split	Method	Divides the <i>string</i> variable value into a series of substrings based upon a specified separator character and returns the array of substrings.
string.Substring	Method	Returns a substring of the <i>string</i> variable starting at a specific character position and with the specified length.
string.ToLower	Method	Returns a copy of the <i>string</i> with all lower case characters.
string.ToUpper	Method	Returns a copy of the <i>string</i> with all upper case characters.
string.Trim	Method	Trims off characters or white space from the start and end of a String variable value.
string.TrimEnd	Method	Trims off characters or white space from the end of a String variable value.
string.TrimStart	Method	Trims off characters or white space from the start of a String variable value.

The following table summarizes the **String** functions that are also described in greater detail in the subsequent section.

Function	Description
Asc (string)	Converts the first character of a String to its equivalent ASCII numerical code.

<u>Chr (<i>expression</i>)</u>	Given a numerical ASCII code, a String that consists of the equivalent ASCII character is returned.
<u>Format (<i>expression</i>, <i>format</i> <i>s</i>)</u>	Converts a numerical value to a String value based upon a specified output format specification.
<u>Instr (<i>start</i>, <i>string</i> <i>t</i>, <i>string</i> <i>s</i>)</u>	Searches for an exact match of a substring within a String expression and returns the starting position if found (1-n).
<u>LCase (<i>string</i>)</u>	Returns a String value that has been converted to lower case.
<u>Len (<i>string</i>)</u>	Returns the number of characters in a String .
<u>Mid(<i>string</i>, <i>first</i>, <i>length</i>)</u>	Returns a substring of the <i>string</i> starting at the <i>first</i> character position and consisting of <i>length</i> number of characters.
<u>UCase (<i>string</i>)</u>	Returns a String value that has been converted to upper case.

String.Compare Method

Compares two **String** expressions either taking into consideration or ignoring the case of the characters and returns an indication of the results.

```
...String.Compare( string_a, string_b, ignore_case )
```

Prerequisites

None

Parameters

string_a

A required **String** expression. The **String** expression can be a **String** variable, constant, function or method, or a concatenation of these **String** elements.

string_b

A required **String** expression. The **String** expression can be a **String** variable, constant, function or method, or a concatenation of these **String** elements.

ignore_case

An optional numeric expression. If the value of this expression is **True**, the comparison is performed ignoring the case of the characters, i.e. "A" will be equal to "a". If this value is **False** or not specified, the comparison is performed in a case-sensitive manner.

Remarks

This shared method compares the values of two **String** expressions and returns an indication of the results of the comparison. Depending upon the value of *ignore_case*, the comparison is either performed taking into account the case of characters or ignoring the case of characters. The returned value is interpreted as follows:

String Relationship	Returned result
<i>string_a</i> > <i>string_b</i>	> 0
<i>string_a</i> = <i>string_b</i>	= 0
<i>string_a</i> < <i>string_b</i>	< 0

String comparisons can also be performed using the standard comparison operators, i.e. =, <>, <, >, <=, >=. When two **Strings** are compared using the comparison operators, the comparison is always performed taking into consideration the case of the characters.

GPL Dictionary Pages

Examples

```
Dim stg As String           ' Create a new string variable
Dim ii As Integer
stg = "aBcdef"
ii = String.Compare(stg, "abcdef")  ' ii will be set <0
```

See Also

[Strings](#)

string.IndexOf Method

Searches for an exact match of a substring within a *string* variable and returns the starting position if found (0-n).

```
...string.IndexOf( string_s, start )
```

Prerequisites

None

Parameters

string_s

A required **String** expression. The **String** expression can be a **String** variable, constant, function or method, or a concatenation of these **String** elements. This specifies the substring value that must be found within the *string* value.

start

An optional numeric expression. This value specifies the first character position that is tested in the *string*. If undefined, match testing begins with the first character in *string*. Unlike the **Instr** function, a 0 specifies the first character position in the *string*.

Remarks

This method searches the value of the *string* variable for an exact, case sensitive match to the specified *string_s* value. The search begins at the character specified by *start* and continues with successive characters until either the first match is found or the end of the *string* is encountered.

Depending upon the outcome of the search, the following values are returned by this method.

String Values	Returned Value
<i>string_s</i> is found in <i>string</i>	Character position where the match begins. 0 indicates matched started at the first character of <i>string</i> .
<i>string</i> has a zero length	-1
<i>string_s</i> has a zero length	<i>start</i> value
<i>string_s</i> not found in <i>string</i>	-1

Examples

```
Dim stg_a As String           ' Create string variable
Dim pos As Integer
stg_a = "aBcDeFgHiJkLmNoPqRsTuVwXyZaBcDeFgHiJk"
pos = stg_a.IndexOf("Fg")      ' pos will be set to 5
pos = stg_a.IndexOf("FG")     ' pos will be set to -1
pos = stg_a.IndexOf("Fg", 10) ' pos will be set to 31
```

See Also

[Strings](#) | [Instr Function](#)

string.Length Property

Returns the count of the number of characters stored in a **String** variable.

```
...string.Length
```

Prerequisites

None

Parameters

None

Remarks

Returns the **Integer** count of the number of characters that are stored in a **String** variable. If the value of the **String** is empty, a count of 0 is returned.

Examples

```
Dim stg As String          ' Create a new string variable
Dim ii As Integer
stg = "123456"
ii = stg.Length            ' ii will be set to 6
```

See Also

[Strings](#) | [Len Function](#)

string.Split Method

Divides a **String** variable value into a series of substrings based upon a specified separator character and returns the array of substrings.

```
...string.Split( separator_string )
```

Prerequisites

None

Parameters

separator_string

A required **String** expression. The **String** expression can be a **String** variable, constant, function or method, or a concatenation of these **String** elements. The first character of this expression defines the separator character. For example, to split a line containing substrings separated by commas, this **String** should be set to ",".

Remarks

This method scans the value of the *string* variable searching for the specified separator character. Each time the separator is found, the text after the previous separator (or from the start of the *string* if this is the first separator) and up to the new separator is taken as a substring and stored in a **String** array that is returned by this method. If the *string* variable does not contain a separator character, the entire contents of the *string* are copied to first element of the output array.

Examples

```
Dim stg_arr() As String      ' Create array string variable
Dim stg As String
stg = "1,2 ,this is the 3rd string"
stg_arr = stg.Split(",")    ' stg_arr(0) = "1"
                             ' stg_arr(1) = "2 "
                             ' stg_arr(2) = "this is the 3rd string"
```

See Also

[Strings](#)

string.Substring Method

Extracts and returns a substring of the *string* variable starting at a specific character position and with a specified length.

```
...string.Substring( first_pos, length )
```

Prerequisites

None

Parameters

first_pos

A required numeric expression. This specifies the position of the first character to be extracted and returned. Note, unlike the **Mid** function, the first character position is 0 rather than 1.

length

An optional numeric expression. This value specifies the number of characters to be copied into the returned value. If *length* is 0, the returned substring will be empty. If *length* is not specified, all of the remaining characters in the *string* starting at the *first_pos* will be copied.

Remarks

This method extracts a substring from the value of a **String** variable and returns the results. The substring is specified by its starting character position in the *string* and the number of characters to be extracted.

Examples

```
Dim stg_a, stg_result As String      ' Create two string variables
stg_a = "aBcdef"
stg_result = stg_a.Substring(3, 2)    ' stg_result will be set to "de"
```

See Also

[Strings](#) | [Mid Function](#)

string.ToLower Method

Returns a copy of a **String** value where all of the alphabetic characters have been changed to lower case.

...string.ToLower

Prerequisites

None

Parameters

None

Remarks

This method copies the value of a **String** variable and converts all of the alphabetic characters to lower case while leaving all of the non-alphabetic characters unchanged.

Examples

```
Dim stg_a, stg_b As String      ' Create two string variables
stg_a = "aBcDeF"
stg_b = stg_a.ToLower          ' stg_b set to "abcdef"
```

See Also

[Strings](#) | [LCase Function](#) | [string.ToUpper](#) | [UCase Function](#)

string.ToUpper Method

Returns a copy of a **String** value where all of the alphabetic characters have been changed to upper case.

```
...string.ToUpper
```

Prerequisites

None

Parameters

None

Remarks

This method copies the value of a **String** variable and converts all of the alphabetic characters to upper case while leaving all of the non-alphabetic characters unchanged.

Examples

```
Dim stg_a, stg_b As String      ' Create two string variables
stg_a = "aBcDeF"
stg_b = stg_a.ToUpper          ' stg_b set to "ABDCEF"
```

See Also

[Strings](#) | [LCase Function](#) | [string.ToLower](#) | [UCase Function](#)

string.Trim Method

Trims off characters or white space from the start and end of a **String** variable value.

```
...string.Trim( trim_chars )
```

Prerequisites

None

Parameters

trim_chars

An optional **String** expression. The characters of this expression define the individual characters that are to be trimmed from the start and the end of the *string*. If a trimming character **String** is not specified, any white space (e.g. space and/or horizontal tab characters) is trimmed off.

Remarks

This method trims off any occurrence of the characters specified in the *trim_chars* expression from the associated *string* variable and returns the resulting **String** value. If multiple trim characters are present in the *string*, trimming continues until a non-trim character is encountered. Trimming is performed at both the start and at the end of the *string* variable.

Examples

```
Dim stg_a, stg_t As String      ' Create string variables
stg_a = "112211this is a test221122"
stg_t = stg_a.Trim("12")       ' stg_t set to "this is a test"
stg_t = stg_a.TrimStart("21")  ' stg_t set to "this is a test221122"
stg_t = stg_a.TrimEnd("123")   ' stg_t set to "112211this is a test"
stg_a = "  another test  "
stg_t = stg_a.Trim()           ' stg_t set to "another test"
```

See Also

[Strings](#) | [string.TrimEnd](#) | [string.TrimStart](#)

string.TrimEnd Method

Trims off characters or white space from the end of a **String** variable value.

```
...string.TrimEnd( trim_chars )
```

Prerequisites

None

Parameters

trim_chars

An optional **String** expression. The characters of this expression define the individual characters that are to be trimmed from the end of the *string*. If a trimming character **String** is not specified, any white space (e.g. space and/or horizontal tab characters) is trimmed off.

Remarks

This method trims off any occurrence of the characters specified in the *trim_chars* expression from the associated *string* variable and returns the resulting **String** value. If multiple trim characters are present in the *string*, trimming continues until a non-trim character is encountered. Trimming is performed at the end of the *string* variable.

Examples

```
Dim stg_a, stg_t As String      ' Create string variables
stg_a = "112211this is a test221122"
stg_t = stg_a.Trim("12")       ' stg_t set to "this is a test"
stg_t = stg_a.TrimStart("21")  ' stg_t set to "this is a test221122"
stg_t = stg_a.TrimEnd("123")   ' stg_t set to "112211this is a test"
stg_a = "    another test    "
stg_t = stg_a.Trim()           ' stg_t set to "another test"
```

See Also

[Strings](#) | [string.Trim](#) | [string.TrimStart](#)

string.TrimStart Method

Trims off characters or white space from the start of a **String** variable value.

```
...string.TrimStart( trim_chars )
```

Prerequisites

None

Parameters

trim_chars

An optional **String** expression. The characters of this expression define the individual characters that are to be trimmed from the start of the *string*. If a trimming character **String** is not specified, any white space (e.g. space and/or horizontal tab characters) is trimmed off.

Remarks

This method trims off any occurrence of the characters specified in the *trim_chars* expression from the associated *string* variable and returns the resulting **String** value. If multiple trim characters are present in the *string*, trimming continues until a non-trim character is encountered. Trimming is performed at the start of the *string* variable.

Examples

```
Dim stg_a, stg_t As String      ' Create string variables
stg_a = "112211this is a test221122"
stg_t = stg_a.Trim("12")       ' stg_t set to "this is a test"
stg_t = stg_a.TrimStart("21")  ' stg_t set to "this is a test221122"
stg_t = stg_a.TrimEnd("123")   ' stg_t set to "112211this is a test"
stg_a = "  another test  "
stg_t = stg_a.Trim()           ' stg_t set to "another test"
```

See Also

[Strings](#) | [string.Trim](#) | [string.TrimEnd](#)

Asc Function

Converts the first character in a **String** variable or expression into its equivalent ASCII numerical code and returns the **Integer** result.

```
...Asc ( string )
```

Prerequisites

None

Parameters

string

A required **String** value. The *string* can be a **String** variable, constant, method or concatenated value.

Remarks

Given a **String** variable or expression, the first character in the **String** is extracted and its equivalent numerical value is returned as an **Integer**. This routine is convenient if you have a string that contains non-printable characters and you wish to operate on their values.

Examples

```
Dim ii As Integer
Dim ss As String
ss = Chr(10)           ' Line feed character
ii = Asc(ss)           ' ii will be set to 10
```

See Also

[Strings](#) | [Chr Function](#)

Chr Function

Given a numerical ASCII code, a **String** that consists of the equivalent ASCII character is constructed and returned.

...Chr (*expression*)

Prerequisites

None

Parameters

expression

A required numerical expression. The *expression* must have an **Integer** value that ranges from 0 to 255.

Remarks

Given a numerical *expression* whose **Integer** value defines one of 256 possible ANSI ASCII character codes, a **String** is constructed and returned that contains a single character set to the ASCII code.

This routine is convenient if you wish to construct a **String** value that contains non-printable characters.

Examples

```
Dim ii As Integer
Dim ss As String
ss = Chr(10)           ' Line feed character
ss = Chr(GPL_CR)       ' Carriage return character
ii = Asc(ss)           ' ii will be set to 10
```

See Also

[Strings](#) | [Asc Function](#)

Format Function

Converts a numerical value to a **String** value based upon a specified output format specification.

```
...Format( expression, format_s )
```

Prerequisites

None

Parameters

expression

A required numeric expression. This defines the numerical value that is to be converted to a string. This value can be any numeric type, e.g. **Integer**, **Double**, **Boolean**, etc.

format_s

An optional **String** expression. This **String** expression defines the output format to generate. If *format_s* is not specified or is an empty **String** value, the default format ("G") is utilized.

Remarks

This function converts a numerical value to a **String** in a specified format. The *format_s* value specifies one of several pre-defined formats or defines a custom format. If the format specification is not recognized, the contents of *format_s* are copied to the output in place of a converted numerical value.

To specify a pre-defined formats, *format_s* must contain one of the single character specifications described in the following table.

Predefined Formats	Output Format
"G" or "g"	General purpose format. Displays a maximum of 17 characters including the sign character. Includes at least one integer digit with no leading space characters or trailing zero's in the fractional part. If the number is too large to display in 17 characters, this format automatically switches to scientific notation.
"F" or "f"	Fixed format. Always displays two fractional digits plus at least one integer digit and more as required. No leading or trailing space characters are generated.
"E" or "e"	Scientific notation. Generates a value in the form of

	"[s]n.nnnnnnesxx" where "s" is a "+" or "-" sign character and "xx" is the base 10 exponent.
--	--

The custom format definition is a character by character literal description of the output format. For example, "0.00#" specifies that the output is to contain as least one integer digit and two fractional digits with an optional third fractional digit. If the numerical value contains more integer digits than specified by the format, additional digits are added to the left to fully display the numerical value. If additional fractional digits exist, the fractional part is rounded to the specified number of fractional digits and only the specified fractional digits are displayed. Leading and trailing space characters are not included in the output.

The following table defines the character placeholders permitted in a custom format.

Custom Formats	Output Format
"0"	Displays a digit or "0" if none. If a "0" is to the left of the decimal point, sufficient leading zeros are generated to display the specified number of decimal digits. Likewise, a "0" to the right of the decimal point always results in a digit or a "0" character. For instance, when the number 23 is displayed using the format "0000.0", the output of the Format function is "0023.0".
"#"	Displays a digit or nothing. If a "#" is to the left of the decimal point, a digit is displayed if it is non-zero else nothing is added to the output stream. Likewise, if a "#" is to the right of the decimal point, only non-zero digits are displayed. For instance, when the number 23 is displayed using the format "###0.#", the output of this function is "23.".
","	Decimal point placeholder. Separates integer and fractional placeholders. Also, results in a "." being included in the output stream.
"E" or "e"	Scientific notation. Outputs a number in scientific notation. This format always generates one digit to the left of the decimal point and a sign character and two digits in the exponent, e.g. "[s]n.nnnnesxx". The significance of the custom format is to specify the number of fractional digits to be included.

Examples

```

Dim stg_a As String          ' Create string variable
stg_a = Format(2323)          ' Default ("G") format, "2323"
stg_a = Format(2323,"G")     ' General ("G") format, "2323"
stg_a = Format(2323,"F")     ' Fixed ("F") format, "2323.00"
stg_a = Format(2323,"E")     ' Exponential ("E") format, "2.323000e+03"

stg_a = Format(.2,".0#")      ' Outputs ".2"
stg_a = Format(.23,".0#")    ' Outputs ".23"
stg_a = Format(-.23,".0#")   ' Outputs "-.23"
stg_a = Format(2.1,".##")    ' Outputs "2.1"
stg_a = Format(23.23,".000") ' Outputs "23.230"
stg_a = Format(23.23,"0000") ' Outputs "0023"
stg_a = Format(23.23,"0")    ' Outputs "23"
stg_a = Format(-.23,"0.00e000") ' Outputs "-2.30e-01"

```

See Also

[Strings](#) | [CStr Function](#) | [Hex Function](#)

Instr Function

Searches for an exact match of a substring within a **String** expression and returns the starting position if found (1-n).

```
...Instr( start, string_t, string_s )
```

Prerequisites

None

Parameters

start

A required numeric expression. This value specifies the first character position that is tested in *string_t*. Unlike the **IndexOf** method, a 1 specifies the first character position in *string_t*.

string_t

A required **String** expression. The **String** expression can be a **String** variable, constant, function or method, or a concatenation of these **String** elements. This specifies the target **String** that is searched for the substring, *string_s*.

string_s

A required **String** expression. The **String** expression can be a **String** variable, constant, function or method, or a concatenation of these **String** elements. This specifies the substring value that must be found within the *string_t* value.

Remarks

This method searches the value of the *string_t* expression for an exact, case sensitive match to the specified *string_s* value. The search begins at the character specified by *start* and continues with successive characters until either the first match is found or the end of the *string_t* is encountered.

Depending upon the outcome of the search, the following values are returned by this method.

String Values	Returned Value
<i>string_s</i> is found in <i>string_t</i>	Character position where the match begins. 1 indicates matched started at the first character of <i>string</i> .
<i>string_t</i> has a zero length	0

<i>string_s</i> has a zero length	<i>start</i> value
<i>string_s</i> not found in <i>string_t</i>	0

Examples

```

Dim stg_a As String           ' Create string variable
Dim pos As Integer
stg_a = "aBcDeFgHiJkLmNoPqRsTuVwXyZaBcDeFgHiJk"
pos = Instr(1, stg_a, "Fg")    ' pos will be set to 6
pos = Instr(1, stg_a, "FG")    ' pos will be set to 0
pos = Instr(10, stg_a, "Fg")   ' pos will be set to 32

```

See Also

[Strings](#) | [string.IndexOf](#)

LCase Function

Returns a copy of a **String** expression where all of the alphabetic characters have been converted to lower case.

```
...LCase( string_exp )
```

Prerequisites

None

Parameters

string_exp

A required **String** expression. *string_exp* can be a **String** variable, constant, function, method or a concatenation of these **String** elements.

Remarks

This function evaluates a **String** expression, converts all of the alphabetic characters to lower case leaving all of the non-alphabetic characters unchanged, and returns the resulting **String** value.

Examples

```
Dim stg_result As String           ' Create a string variable
stg_result = LCase("aBcDeF")      ' stg_result set to "abcdef"
```

See Also

[Strings](#) | [string.ToLower](#) | [string.ToUpper](#) | [UCase Function](#)

Len Function

Returns the count of the number of characters contained in a **String** variable or expression.

```
...Len ( string )
```

Prerequisites

None

Parameters

string

A required **String** value. The *string* can be a **String** variable, constant, method or concatenated value.

Remarks

Returns the **Integer** count of the number of characters contained in the specified *string*. If the value of the *string* is empty, a count of 0 is returned.

Examples

```
Dim ii As Integer  
ii = Len("123456")           ' ii will be set to 6
```

See Also

[Strings](#) | [string.Length](#)

Mid Function

Returns a substring of a **String** expression starting at the specified character position and consisting of a specified number of characters.

```
...Mid( string_exp, first_pos, length )
```

Prerequisites

None

Parameters

string_exp

A required **String** expression. *string_exp* can be a **String** variable, constant, function, method or a concatenation of these **String** elements.

first_pos

A required numeric expression. This specifies the position of the first character to be extracted and returned. Note, unlike the **Substring** method, the first character position is 1 rather than 0.

length

An optional numeric expression. This value specifies the number of characters to be copied into the returned value. If *length* is 0, the returned substring will be empty. If *length* is not specified, all of the remaining characters in the *string_exp* starting at the *first_pos* will be copied.

Remarks

This function evaluates a **String** expression, extracts a substring from its value, and returns the results. The substring is specified by its starting character position in *string_exp* and the number of characters to be extracted.

Examples

```
Dim stg_result As String          ' Create a string variable
stg_result = Mid("aBcdef", 4, 2)  ' stg_result will be set to "de"
```

See Also

[Strings](#) | [string.Substring](#)

UCase Function

Returns a copy of a **String** expression where all of the alphabetic characters have been converted to upper case.

```
...UCase( string_exp )
```

Prerequisites

None

Parameters

string_exp

A required **String** expression. *string_exp* can be a **String** variable, constant, function, method or a concatenation of these **String** elements.

Remarks

This function evaluates a **String** expression, converts all of the alphabetic characters to upper case leaving all of the non-alphabetic characters unchanged, and returns the resulting **String** value.

Examples

```
Dim stg_result As String           ' Create a string variable
stg_result = UCase("aBcDeF")      ' stg_result set to "ABCDEF"
```

See Also

[Strings](#) | [LCase Function](#) | [string.ToLower](#) | [string.ToUpper](#)

Thread Class

Thread Class Summary

The following pages provide detailed information on the methods of the **Thread Class**. This class provides the means for starting, stopping, and monitoring the execution of independent threads.

The GPL system supports the simultaneous execution of up to 32 GPL program threads. Each thread has its own execution stack and runs independently of all other threads. If multiple threads are active, each thread executes for up to 1 millisecond before control passes to the next ready thread.

When a GPL project is loaded, one procedure is designated as the main procedure in the project file settings. This main procedure is started by the GDE interface, the web Operator Control Panel, the **Start** console command, or automatically when the system is restarted.

The main procedure can then start additional procedures as separate threads.

The table below briefly summarizes the methods and properties that are described in greater detail in the following sections

Member	Type	Description
New Thread	Constructor Method	Creates a thread object and associates it with a procedure.
thread_object.Abort	Method	Stops execution of a thread such that it cannot be resumed.
Thread.CurrentThread	Shared Method	Returns a thread object for the currently executing thread.
thread_object.Join	Method	Waits for a thread to complete execution, with a timeout.
thread_object.Resume	Method	Resumes execution of a thread that was suspended.
thread_object.SendEvent	Method	Sends an event to a thread to notify it that a significant transition has occurred.
Thread.Sleep	Shared Method	Causes the current thread to stop execution for a specified amount of time.
thread_object.Start	Method	Initializes and starts execution of a procedure as an independent thread.
thread_object.Suspend	Method	Suspends execution of a thread so that it can be resumed.
thread_object.ThreadState	Get Property	Returns an integer indicating the execution state of a thread.
Thread.WaitEvent	Shared Method	Causes the current thread to wait for an event.

New Thread Constructor

Constructor for creating a thread object and associating it with the procedure executed by the thread.

New Thread(*procedure_name*, *project_name*, *thread_name*, *stack_size*)

Prerequisites

None

Parameters

procedure_name

A required string expression that specifies the name of the first procedure to be executed by the thread. This procedure must be declared as **Public**. That is, the **Public** keyword must be specified in its definition.

project_name

An optional string expression that specifies the name of the project that contains *procedure_name*. If this parameter is omitted, the name of the current project is assumed. Specifying this parameter is not supported by GPL at this time.

thread_name

An optional string expression that specifies the name of the thread to be created. If this parameter is omitted, the *procedure_name* value is used as the thread name.

stack_size

An optional numeric expression that specifies the number of kilobytes of stack to allocate for this thread. If zero or omitted, the default stack size for this project is used.

Remarks

This method does not actually create the thread in the system. It simply records the names for use by the **Start** method. If the procedure or project does not exist, no errors occur until the **Start** method is called.

Examples

```
Dim thread1 As New Thread("Test") ' Create a thread object to execute the  
                                     ' Public procedure Test in the current project  
Dim thread1 As New Thread("Test", "Thread1") ' Create a thread object to execute  
                                               ' Public procedure Test with thread name Thread1
```

See Also

[Thread Class](#) | [thread_object.Start](#)

thread_object.Abort Method

Stops a thread's execution immediately and does not allow it to be resumed. The thread must be restarted from the beginning.

thread_object.**Abort()**

Prerequisites

None

Parameters

None

Remarks

This method stops the thread associated with the object and deallocates internal resources, just as if a console **Stop** command were issued. The thread cannot be resumed, but can only be restarted using the **Start** method.

If you wish to be able to resume a thread, use the **Suspend** method instead.

If a thread executes the **Abort** method for itself, the thread exits with an error, but it is not deallocated in the same way as a separate thread

Examples

```
Dim thread1 As New Thread("Test") ' Create a thread object to execute the
                                   ' procedure Test in the current project
thread1.Start()                   ' Start the thread
thread1.Abort()                   ' Stop the thread and prevent resumption.
Thread.CurrentThread.Abort()      ' Stops thread in which it is executed
```

See Also

[Thread Class](#) | [thread_object.Start](#) | [thread_object.Suspend](#)

Thread.CurrentThread Shared Method

Returns a thread object that corresponds to the currently running thread.

```
thread_object = Thread.CurrentThread()
```

Prerequisites

None

Parameters

None

Remarks

This shared method returns an object that corresponds to the currently running thread. This object may be used to abort or suspend the current thread. It does not need to be associated with a thread object, only the thread class.

Examples

```
Dim mythread As Thread.CurrentThread() ' Create a thread object for the  
                                         ' current thread.  
Thread.CurrentThread.Suspend ()        ' Suspend the current thread.
```

See Also

[Thread Class](#)

thread_object.Join Method

Waits for a thread to become idle, with a timeout. Returns -1 (True) if the thread is now idle or 0 (False) if the timeout time was exceeded.

```
status = thread_object.Join( millisecond_timeout )
```

Prerequisites

None

Parameters

millisecond_timeout

The maximum time to wait for the thread associated with *thread_object* to become idle. A value of 0 means do not wait, just test if the thread is idle. A value of -1 means do not timeout, wait forever for the thread.

Remarks

When this method is called, the calling thread waits until the thread associated with *thread_object* becomes idle, or until the specified timeout value is exceeded. The returned value of the method is -1 (True) if the thread is idle or if the thread does not exist. The returned value is 0 (False) if the thread exists and is not idle. Normally a returned value of 0 indicates that the timeout time has been exceeded. If the *calling* thread is suspended externally and then resumed during the **Join** method, the value 0 is returned even though the timeout time may not have been exceeded.

If the referenced thread is suspended or stops with an error, the **Join** method continues waiting. It only completes with True when the thread is idle or deleted.

Examples

```
Dim thread1 As New Thread("Test") ' Create a thread object to execute the
                                   ' procedure Test in the current project
Dim status As Integer
thread1.Start()                   ' Start the thread
status = thread1.Join(10000)      ' Wait for the thread to complete with a
                                   ' 10-second timeout.
If status Then
    Console.WriteLine("thread1 is complete")
End If
```

See Also

[Thread Class](#) | [thread_object.ThreadState](#)

thread_object.Resume Method

Resumes execution of a thread that was previously suspended.

```
thread_object.Resume()
```

Prerequisites

None

Parameters

None

Remarks

This method resumes the thread associated with the object, just as if a console **Continue** command were issued. The thread may have been stopped by the **Suspend** method, or by a break point, or by the console **Break** command.

If the thread is not suspended, this method does nothing.

Examples

```
Dim thread1 As New Thread("Test") ' Create a thread object to execute the
                                     ' procedure Test in the current project
thread1.Start()                    ' Start the thread
thread1.Suspend()                  ' Suspend the thread for now.
Thread.Sleep(1000)                 ' Wait for 1 second
thread1.Resume()                   ' Resume the thread
```

See Also

[Thread Class](#) | [thread_object.Suspend](#)

thread_object.SendEvent Method

Sends an event to a specific thread to notify it that a significant transition has occurred.

```
thread_object.SendEvent( event_mask )
```

Prerequisites

None

Parameters

event_mask

A required numeric expression that specifies the events to be sent. Each bit in *event_mask* corresponds to a different event. Bit 0 (mask value &H0001) corresponds to event 1. Multiple events may be specified. The maximum event is 16, so the maximum value for *event_mask* is &HFFFF.

Remarks

Events are messages that are sent to synchronize one thread that is executing a GPL project with another GPL project thread. Utilizing events has several advantages over setting and polling a global variable:

-
- The thread waiting for an event uses almost no CPU time, as opposed to polling a global variable.
- There is very little latency between when a message is sent and when the target thread wakes up and handles the event, as opposed to a polling method where the worst-case latency is the polling period.

For more details on events and event handling, see the **WaitEvent** method

Examples

```
Dim t1 As New Thread("TestThread")
t1.Start
:
t1.SendEvent(&H10)           ' Send event 5 to thread
```

See Also

[Thread Class](#) | [Thread.WaitEvent](#)

Thread.Sleep Shared Method

Makes the current thread wait until a specified number of milliseconds have passed.

Thread.Sleep(*milliseconds*)

Prerequisites

None

Parameters

milliseconds

The number of milliseconds that this thread should wait before continuing execution with the next statement. A value of 0 means allow another thread to execute, but continue execution of the current thread immediately if no other thread is ready. A value < 0 means wait forever, and is equivalent to invoking the **Suspend** method for the current task.

Remarks

This shared method is normally associated with the thread class, not an object. If it is used with an object, the current thread always waits, regardless of the thread object contents.

If a sleeping thread is suspended and resumed, the wait continues relative to the time when this method was originally invoked.

Examples

```
Thread.Sleep(5000)           ' The current thread waits for 5 seconds

Dim thread1 As New Thread("Test") ' Create an object for a different thread
thread1.Sleep(1000)          ' The current thread waits for 1 second
```

See Also

[Thread Class](#)

thread_object.Start Method

Starts the execution of an independent thread.

```
thread_object.Start()
```

Prerequisites

The procedure associated with *thread_object* must be declared **Public**.

The procedure associated with *thread_object* must be loaded into memory and compiled without errors.

Parameters

None

Remarks

This method begins a new thread that executes the procedure associated with the *thread_object*, just as if a console **Start** command were issued.

If the thread is currently active, this method does nothing and returns without error.

If the thread is currently paused, it is restarted by clearing the execution stack and executing the procedure associated with the object. If a thread is stopped by using the **Abort** method, it can only be restarted by using **Start**.

If the project or procedure associated with the object does not exist, or if there were any errors compiling the project, this method issues an error.

Examples

```
Dim thread1 As New Thread("Test") ' Create a thread object to execute the
                                     ' Public procedure Test in the current project
thread1.Start()                    ' Start the thread
```

See Also

[Thread Class](#) | [thread_object.Abort](#)

thread_object.Suspend Method

Suspends the execution of an independent thread.

thread_object.**Suspend()**

Prerequisites

None

Parameters

None

Remarks

This method suspends the thread associated with *thread_object*, just as if a console **Break** command were issued. The thread stops at the end of the current GPL instruction. The thread may be resumed where it left off by the **Resume** method or by a console **Continue** command.

If the thread does not exist, an error occurs. If the thread exists but is not currently active, no error is generated.

This method does not wait until the thread actually stops. Use the **ThreadState** property to determine when the thread is suspended.

Examples

```
Dim thread1 As New Thread("Test") ' Create a thread object to execute the
                                     ' procedure Test in the current project
thread1.Start()                    ' Start the thread
thread1.Suspend()                  ' Suspend the thread for now.
Thread.Sleep(1000)                 ' Wait for 1 second
thread1.Resume()                   ' Resume the thread
```

See Also

[Thread Class](#) | [thread_object.Resume](#)

thread_object.ThreadState Property

Gets a numeric value indicating the execution state of the thread specified by *thread_object*.

```
state_var = thread_object.ThreadState
```

Prerequisites

None

Parameters

None

Remarks

This property returns information about a thread's execution state. The numeric value returned by this property is described in the table below.

ThreadState Value	Description
-1	The thread does not exist. Either it was never started or it was stopped and deleted by an Abort method.
0	The thread has completed execution normally and is idle. It cannot be resumed, but it can be restarted with Start .
1	The thread is stopping execution. This state is transient.
2	The thread is executing normally.
3	The thread is paused without error and can be resumed.
4	The thread is paused with an error. If it is resumed, it will retry the instruction that caused the error.

Examples

```
Dim thread1 As New Thread("Test")      ' Create a thread object to execute the
thread1.Start()                        ' procedure Test in the current project
Console.WriteLine(thread1.ThreadState) ' Start the thread
                                     ' Display the state code for thread1
```

See Also

[Thread Class](#)

Thread.WaitEvent Shared Method

Wait for, test and clear events received by the current thread. Returns a mask indicating the received events.

```
received_events = Thread.WaitEvent( event_mask, time_out )
```

Prerequisites

None

Parameters

event_mask

A required numeric expression that specifies the set of events to wait for. Each bit in *event_mask* corresponds to a different event. Multiple events may be specified. The maximum event is 16, so the maximum value for *event_mask* is &HFFFF.

If *event_mask* is 0, no wait occurs, no events are cleared, and all received events are returned.

time_out

A required numeric expression that specifies the maximum time, in milliseconds, to wait if no matching events are received. The maximum wait time is 2147 seconds.

If 0, this method does not wait, but only tests pending events against the *event_mask*. If < 0, this method does not timeout and waits forever.

Remarks

The returned value is a bit mask indicating events that have been received. Bit 0 (mask value &H0001) corresponds to event 1. The mask indicates either all pending events, or only those matched by *event_mask*, as described below.

The behavior of this method depends on the combination of parameters as described in the following table.

<i>event_mask</i> Value	<i>time_out</i> Value	Description
0	N.A.	The method does not wait for or clear any events, but simply returns a bit mask indicating all received events.
<> 0	0	The method does not wait. It clears all events that match the

		bits in <i>event_mask</i> . It returns a bit mask indicating the events that were cleared. This parameter combination may be used to return and clear specific received events without waiting.
<> 0	> 0	<p>The method waits until at least one event corresponding to a bit in <i>event_mask</i> has been received. If a matching event was previously received and not cleared, the method does not wait.</p> <p>Before returning, it clears all pending events that match the bits in <i>event_mask</i>, and returns a bit mask indicating the events that were cleared.</p> <p>If no matching event is received before the timeout period, this method returns a value of 0.</p>
<> 0	< 0	This case is the same as " <i>event_mask</i> <> 0, <i>time_out</i> > 0" case except that it waits indefinitely for the events, and never times out.

Events are synchronization messages that are sent from one thread executing a GPL project to another thread that is executing a GPL project. Utilizing events has several advantages over setting and polling a global variable:

-
- The thread waiting for an event uses almost no CPU time, as opposed to polling a global variable.
- There is very little latency between when a message is sent and when the target thread wakes up and handles the event, as opposed to a polling method where the worst-case latency is the polling period.

Each thread can handle up to 16 different events. These 16 events are independent of the events for all other threads. An event is specified by the target thread and a bit within the thread's *event_mask*.

Events handled by **WaitEvent** are automatically cleared, except for the special case when *event_mask* = 0. A receiving thread can simply loop waiting for events, checking the returned bit mask, and servicing whatever events bits are set. If the **WaitEvent** *event_mask* specifies more than one event, be sure to check all possible events, since more than one event may be returned simultaneously and be cleared.

In a client-server situation, a client thread can place a command in a global variable, and then send an event to the server. When the server receives the event, it can examine the global variable to determine the detailed command.

Examples

```
Public main_thread As Thread

Public Sub Main
    Dim t1 As New Thread("Testthread")
    main_thread = Thread.CurrentThread
    t1.Start
    t1.SendEvent(&H10)           ' Send event 5 to thread
    Thread.WaitEvent(&H8, -1)    ' Wait for event 4, clear it
    Console.WriteLine ("Main thread event received")
End Sub
```

```
Public Sub Testthread
    Dim events As Integer
    events = Thread.WaitEvent(&H10,100) ' Wait with timeout
    If events = 0 Then
        Console.WriteLine ("Testthread event timeout")
    Else
        Console.WriteLine ("Testthread event received")
    End If
    main_thread.SendEvent(&H8)          ' Send event 4 back to main thread
End Sub
```

See Also

[Thread Class](#) | [thread_object.SendEvent](#)

Vision Classes

Vision Classes Summary

The following pages provide detailed information on the properties and methods for the classes that implement the interface to the PreciseVision machine vision system.

This interface includes two classes: the **Vision Class** that manages communications between GPL and PreciseVision and the **VisResult Class** that stores a single set of results from a single vision tool. As a convenience, there is no explicit method for connecting to PreciseVision. Whenever the Vision methods **Process**, **Result** or **ResultCount** are executed, GPL automatically establishes a connection to the vision system.

The tables below briefly summarize the properties and methods for each Class, which are described in greater detail in the following sections.

Vision Class Member	Type	Description
New Vision	Constructor Method	Creates an empty Vision object. Does not communicate with PreciseVision.
<u>Vison.Disconnect</u>	Shared Method	Closes any open connection to PreciseVision.
<u>vision_obj.ErrorCode</u>	Property	Returns the numeric error code for the last executed vision process. A value of 0 indicates success; a negative value indicates an error.
<u>vision_obj.Process</u>	Method	Requests that PreciseVision execute a vision process and waits for it to complete. Connects to PreciseVision if there is currently no connection.
<u>vision_obj.Result</u>	Method	Returns a VisResult object that contains a single set of results from a previously executed vision tool. Connects to PreciseVision if there is currently no connection.
<u>vision_obj.ResultCount</u>	Method	Returns the number of sets of vision results created by a vision tool the last time it was executed. Connects to PreciseVision if there is currently no connection.
<u>vision_obj.Status</u>	Property	Returns a numeric value indicating the status of a vision process: 0 = No vision process for this object, 1 = Process is running, 2 = Process complete but with error, 3 = Process complete with success.
<u>Vision.ToolProperty</u>	Shared Property	Sets or gets the value of a tool property within PreciseVision.

VisResult Class Member	Type	Description
New VisResult	Constructor Method	Creates an empty VisResult object. Not useful since VisResult objects are normally created by the <i>vision_object.Result</i> method.
<i>visresult_obj.ErrorCode</i>	Property	Returns the numeric error code for this result. A value of 0 indicates success; a negative value indicates an error. A positive value indicates success with a warning.
<i>visresult_obj.Info</i>	Property	Returns the nth numeric information field contained in this set of results.
<i>visresult_obj.InfoCount</i>	Property	Returns number of numeric information items in this set of results.
<i>visresult_obj.InspectActual</i>	Property	Returns the value of the tool property that was tested in the vision inspection process.
<i>visresult_obj.InspectPassed</i>	Property	Returns True if a property of the vision results satisfied the tool's vision inspection criteria.
<i>visresult_obj.Loc</i>	Property	Returns the position and orientation from a set of results as a Cartesian Location object.
<i>visresult_obj.Type</i>	Property	Returns the type of this set of results. Currently always zero.

Vision.Disconnect Method

Closes the network connection to the PreciseVision system..

Vision.Disconnect

Prerequisites

None

Parameters

None

Remarks

This shared method closes any TCP/IP connection to PreciseVision. No error occurs if there is currently no connection.

The Ethernet connection should always be closed when communication with PreciseVision has been completed.

Examples

```
Vision.Disconnect
```

See Also

[Vision Classes](#)

vision_object.ErrorCode Property

Gets the **Integer** error code for the last executed vision process.

```
...vision_object.ErrorCode
```

Prerequisites

A **Process** method must have been executed using the *vision_object* and the execution must be completed.

Parameters

None

Remarks

This property returns the **Integer** error code for the last vision process executed by the *vision_object*. A value of 0 indicates success; a negative value indicates an error. If no process was ever run, a value of 0 is returned. Please see the section on System Error Codes in the *Precise Documentation Library* for a list of vision error codes and their interpretation.

This property is different from the *visresults_object.ErrorCode*. The *visresults_object.ErrorCode* indicates if a specific Vision Tool encountered an error during execution, e.g. it didn't find what it was searching for. The *vision_object.ErrorCode* indicates if a vision process could not be found or if a communication error occurred between GPL and PreciseVision. This property never signals an error if an individual tool fails for whatever reason.

If the *vision_object.Status* property returns a value of 2, indicating that an error has occurred, the **ErrorCode** property contains the specific error code that describes the type of error.

Examples

```
Dim vobject As New Vision
vobject.Process("find_part") ' Execute find_part process
If vobject.ErrorCode <> 0 Then ' Handle error
End If
```

See Also

[Vision Classes](#) | [vision_object.Status](#) | [visresult_object.ErrorCode](#)

vision_object.Process Method

Issues a request to PreciseVision to execute a vision process and waits for the process to complete.

```
vision_object.Process( vision_process_name )
```

Prerequisites

The specified vision process must already be defined within the PreciseVision system.

Parameters

vision_process_name

A required **String** expression that specifies the name of the PreciseVision process that is to be executed. This corresponds to the name that is displayed in the "Process Manager" window in PreciseVision.

Remarks

This method requests PreciseVision to execute the specified vision process. It then waits until PreciseVision has completed the process. If PreciseVision does not respond within 30 seconds, an error exception is thrown.

Executing a vision process is the basic means that GPL utilizes to command PreciseVision to take a picture and analyze it. From GPL's point of view, a vision process is a single, indivisible operation. That is, after GPL starts a vision process, no results are available until after the process completes its execution. When the process is done running, GPL can then interrogate PreciseVision for information on the output of any tool. Normally, a vision process consists of a command to take a picture (i.e. an Acquisition Tool) followed by additional tools to process and analyze the picture. In the simplest case, a process can consist of a single tool that operates on an existing picture. At other times, a process can be quite complex and may consist of dozens of tools that inspect multiple features of parts to verify that the part is correct.

In order for GPL to execute a process and retrieve the results, GPL has to know the name that has been assigned to the process in PreciseVision and the names of any tools for which results are desired.

Each time that a vision process is executed, all of the previous results of its tools are lost and replaced by the newly computed results. However, if a different vision process is executed using another **Vision** object, the results of first vision process are preserved.

The **Status** property can be used to determine if the process completed successfully.

The **Process** method performs communications with PreciseVision. If an Ethernet network connection does not exist, a connection is automatically established. If a

connection cannot be setup or the communication link fails for any reason, this method will throw an exception.

Examples

```
Dim vobject As New Vision
vobject.Process("find_part")
If vobject.Status <> 3 Then          ' Deal with error
End If
```

See Also

[Vision Classes](#) | [vision_object.Status](#)

vision_object.Result Method

Returns a **VisResult Object** that contains a single set of results from a vision tool.

```
...vision_object.Result( vision_tool_name, index, location_object )
```

Prerequisites

A **Process** method must have been executed using the *vision_object* and the execution must be completed.

Parameters

vision_tool_name

An optional **String** expression that specifies the name of a specific PreciseVision tool that was executed in the vision process associated with *vision_object*. The tool name must match one of those listed in the PreciseVision "Process Manager" window for the executed process. If a tool name is specified, a single set of results generated by that tool will be returned. If omitted, a single set of results from the final tool in the vision process is returned.

index

An optional numeric expression indicating which set of results to return for the selected tool. The numeric value can range from 1 to *vision_object.ResultCount*. If omitted, the first set is returned.

location_object

(*Future enhancement*) An optional Cartesian **Location Object** whose value is sent to PreciseVision when the result is requested. Depending on where the camera is mounted and the particular vision tool, this location value may be used to determine the returned vision result. Details on what value to pass in this parameter are described in the PreciseVision documentation for specific vision tools.

Remarks

This method requests PreciseVision to return a set of results from a tool that was part of the previously executed vision process. If the vision tool generated multiple sets of results, the *index* parameter is utilized to specify the set of results to be returned. The results data can be fetched any number of times from any tool that is part of the vision process until the vision process is executed again. When a vision process is executed again, all of the old results are lost and a new set of results data will be available.

When this method is executed, it returns a **VisResult Object** whose data can be accessed by the standard properties and methods available for that object class.

For cameras mounted on a robot or for pictures of an object held by the robot, it may be necessary to pass camera or robot location information to PreciseVision so that the result location may be determined. In this case, the optional *location_object* parameter must be specified.

The **Status** property can be used to determine if the previous vision process completed successfully.

This property performs communications with PreciseVision. If an Ethernet network connection does not exist, a connection is automatically established. If a connection cannot be setup or the communication link fails for any reason, this method will throw an exception.

Examples

```
Dim vobject As New Vision
Dim result As VisResult
vobject.Process("find_part")
result = vobject.Result()           ' Get result 1 of final vision tool
result = vobject.Result("hole1")    ' Get result 1 of vision tool "hole1"
result = vobject.Result(, 2)        ' Get result 2 of final vision tool
```

See Also

[Vision Classes](#) | [vision_object.Process](#)

vision_object.ResultCount Method

Gets the number of results generated by a vision tool in the last executed vision process.

```
...vision_object.ResultCount( vision_tool_name )
```

Prerequisites

A **Process** method must have been executed using the *vision_object* and the execution must be completed.

Parameters

vision_tool_name

An optional **String** expression that specifies the name of a specific PreciseVision tool that was executed in the vision process associated with *vision_object*. The tool name must match one of those listed in the PreciseVision "Process Manager" window for the executed process. If a tool name is specified, the number of sets of results generated by that tool will be returned. If omitted, the number of sets of results for the final tool in the vision process is returned.

Remarks

This property returns the number of sets of results generated by a vision tool. This is the same value as the PreciseVision **ResultCount** tool property.

A value of 0 indicates that no results are available or that some type of error occurred when the tool was executed. Depending upon the basic type for the vision tool, zero, one, or multiple sets of results may be generated each time the tool is executed. For example, the tool that extracts the best fit line (i.e. the Line Fitter) will return at most one set of results if a line can be fit or none if it is unsuccessful. On the other hand, the general tool that locates parts (i.e. the Finder) can generate dozens of sets of results if multiple identical parts are in the camera's field of view.

If one or more sets of results can be accessed, the **Result** method should be called as many times as necessary to fetch the data for each set of results.

This property performs communications with PreciseVision. If an Ethernet network connection does not exist, a connection is automatically established. If a connection cannot be setup or the communication link fails for any reason, this method will throw an exception.

Examples

```
Dim vobject As New Vision
Dim vresults As VisResult
Dim ii As Integer
```

GPL Dictionary Pages

```
Dim results As Integer
vobject.Process("find_part")

results = vobject.ResultCount()

For ii = 1 To results
    vresults = vobject.Result(,ii) ' Process results
Next ii
```

See Also

[Vision Classes](#) | [vision object.Status](#)

vision_object.Status Property

Gets the numeric status code for a vision process.

...vision_object.**Status**

Prerequisites

None

Parameters

None

Remarks

This method returns the status code for the vision process associated with the *vision_object*. The returned status codes are as follows:

Status Code	Description
0	No vision process for this object
1	Vision process is running
2	Vision process completed but with error
3	Vision process completed with success

At this time, the value 1 is not seen because the Process method always waits until the vision process is complete. A no-wait vision process may be added as a future enhancement.

If **Status** has a value is 2, the **ErrorCode** property can be used to determine the specific type of error that has occurred. Note, this property returns an error if the process did not exist or if a communication error occurs. However, if a specific tool fails, such as when a Line Fitter cannot find enough edges to fit a line, **Status** does not indicate an error. For tool analysis errors, please see the *visresults_object*.**ErrorCode** property.

Examples

```
Dim vobject As New Vision
vobject.Process("find_part")
If vobject.Status <> 3 Then      ' Handle non-successful process
End If
```

See Also

[Vision Classes](#) | [vision_object.ErrorCode](#) | [visresults_object.ErrorCode](#)

Vision.ToolProperty Shared Property

Sets or gets a property value associated with a PreciseVision tool.

```
Vision.ToolProperty (property_name_string) = <property_value_string>
-or-
...Vision.ToolProperty (property_name_string)
```

Prerequisites

None

Parameters

property_name_string

A required **String** expression that contains the name of the tool property to get or set. This **String** is normally in the form: *tool_name.property_name*, where *tool_name* is the name of a tool defined in PreciseVision, and *property_name* is the name of a property within that tool.

Remarks

This property allows a GPL program to dynamically change the properties of a tool defined within PreciseVision. This capability allows a GPL program to use the results of a previous vision process to adjust or refine the tools used by a future vision process.

The vision tools available depend on what has been defined in your particular vision application. The properties associated with each tool, and the possible property values are described fully in the *PreciseVision* documentation.

Each time a **ToolProperty** procedure is invoked, messages are exchanged between the Precise Controller and the PreciseVision system.

Examples

```
Dim prop As String
prop = Vision.ToolProperty("hist.angle")
```

See Also

[Vision Classes](#)

visresult_object.ErrorCode Property

Gets the **Integer** error code for a vision results object.

```
...visresult_object.ErrorCode
```

Prerequisites

None

Parameters

None

Remarks

This property returns the **Integer** error code for the *visresult_object*. This is the same value as the *PreciseVision ResultErrorCode* tool property.

A value of 0 indicates that the result was computed successfully and is valid. A positive value indicates a non-critical error occurred during processing, but the result information is valid. A negative value is a standard GPL error code and indicates an error occurred when *PreciseVision* was computing the result. Please see the section on System Error Codes in the *Precise Documentation Library* for a list of vision error codes and their interpretation.

When a critical error occurs, the associated tool and all of the tools that are dependent upon that tool are not processed. The dependent tools will also return a critical error condition when they are queried. When a critical error is indicated, the other properties for the *visresult_object* may not contain valid information.

Examples

```
Dim vresult As VisResult
vresult = vobject.Result()
If vresult.ErrorCode <> 0 Then      ' Handle error
End If
```

See Also

[Vision Classes](#) | [vision_object.ErrorCode](#)

visresult_object.Info Property

Returns a **Double** value from the vision result object's numeric information array.

```
...visresult_object.Info( index )
```

Prerequisites

None

Parameters

index

A required numeric expression that specifies the array index for the information element that is to be returned. The first array element has an index of 0. This parameter must have a value greater than or equal to zero.

Remarks

The common results values returned from the Vision Tools are accessed via standard properties of the **VisResults Objects**, e.g. the position and orientation of the results are available from *visresult_object.Loc*. However, some tools return special numeric data that is specific to the tool. For example, the Finder Tool returns the X and Y scale factors for the parts that it has located. This type of tool specific information is returned in the *visresult_object.Info* array property.

For information on what data a tool returns in this property and the index of the data, please consult the *"PreciseVision Machine Vision System, Introduction and Reference Manual"*. In the detailed descriptions for each tool, properties that are returned in the **Info** array and their array index values are highlighted.

Examples

```
Dim vresult As VisResult
vresult = vobject.Result() ' Get a tool's results
If vresult.Info(2) > .5 Then
    ...
```

See Also

[Vision Classes](#) | [visresult_object.InfoCount](#) | [visresult_object.Type](#)

visresult_object.InfoCount Property

Returns, as an **Integer** value, the number of elements in the vision result object's numeric information array.

`...visresult_object.InfoCount`

Prerequisites

None

Parameters

None

Remarks

The `visresult_object.InfoCount` property returns the number of elements in the `visresult_object.Info` array for the current vision result. The index values for accessing the **Info** array range from 0 to **InfoCount** - 1.

Some tools return special numeric data, which is specific to the tool, in the `visresult_object.Info` array property. Some of these tools, for example the Edge Finder tool, can return a variable number of numeric values. The **InfoCount** property allows a program to determine how many values are actually returned.

For information on what data a tool returns in this property and the index of the data, please consult the *"PreciseVision Machine Vision System, Introduction and Reference Manual"*. In the detailed descriptions for each tool, properties that are returned in the **Info** array and their array index values are highlighted.

Examples

```
Dim vresult As VisResult
Dim ii As Integer
vresult = vobject.Result()           ' Get a tool's results
For ii = 0 To vresult.InfoCount-1
    Console.WriteLine(vresult.Info(ii))
Next ii
```

See Also

[Vision Classes](#) | [vision_object.Info](#)

visresult_object.InspectActual Property

Returns a **Double** that indicates the value of the tool property that was tested in the vision inspection process.

```
...visresult_object.InspectActual
```

Prerequisites

Only returns meaningful data for results generated by a vision tool whose output includes the *InspectActual* property in *PreciseVision*.

Parameters

None

Remarks

This property returns the value of the vision tool property that was tested for the *PreciseVision* inspection process. This is the same value as the *PreciseVision* **InspectActual** tool property.

For many *PreciseVision* tools, a range of acceptable values can be set for a single results property for the tool. For example, for the general object Finder Tool, the orientation angle of any located parts can be tested to ensure that they fall within a specified range.

When the inspection criteria is set, each time the tool is executed, it automatically tests each set of results to see if it satisfies the criteria. **InspectActual** is the property value that was tested during this process. **InspectPassed** indicates the results of the test.

Examples

```
Dim vresult As VisResult
vresult = vobject.Result()

If vresult.InspectPassed = False Then      ' Inspection failed?
    If vresult.InspectActual < 10 Then      ' By how much?
        ...
```

See Also

[Vision Classes](#) | [visresults_object.InspectPassed](#)

visresult_object.InspectPassed Property

Returns a **Boolean** that indicates if a property of the vision results satisfied the tool's vision inspection criteria.

...visresult_object.InspectPassed

Prerequisites

Only returns meaningful data for results generated by a vision tool whose output includes the *InspectPassed* property in *PreciseVision*.

Parameters

None

Remarks

This property returns a **True** or **False** indication of whether or not the set of results from a vision tool satisfied the specified inspection criteria. This is the same value as the *PreciseVision* **InspectPassed** tool property.

For many *PreciseVision* tools, a range of acceptable values can be set for a single results property for the tool. For example, for the general object Finder Tool, the orientation angle of any located parts can be tested to ensure that they fall within a specified range.

When the inspection criteria is set, each time the tool is executed, it automatically tests each set of results to see if it satisfies the criteria and sets the value of *InspectPassed* appropriately. If the inspection fails, the tool is still processed in the normal fashion as well as any tools that are dependent upon the failed result. However, both the failed tool and any dependent tools will have their *InspectPassed* set to **False**.

As a convenience, the tool property value that was tested is returned in *visresults_object.InspectActual*.

Examples

```
Dim vresult As VisResult
vresult = vobject.Result()

If vresult.InspectPassed = False Then      ' Inspection failed?
    If vresult.InspectActual < 10 Then      ' By how much?
        ...
```

See Also

[Vision Classes](#) | [visresults_object.InspectActual](#)

visresult_object.Loc Property

Returns a **Location Object** containing the position and orientation information from a vision result object.

```
...visresult_object.Loc
```

Prerequisites

Only returns meaningful data for results generated by a vision tool whose output includes the *ResultAngle*, *ResultXPos*, and *ResultYPos* properties in PreciseVision.

Parameters

None

Remarks

This property returns the position and orientation results data from a vision tool and provides the information in the form of a Cartesian **Location Object**. The position and orientation data are derived from the PreciseVision **ResultXPos**, **ResultYPos** and **ResultAngle** tool properties.

While not all vision tools generate position and orientation data, many do. For example, the general purpose object Finder tool returns the position and orientation of matched parts. Likewise, the Point-Line Frame tool returns the position and orientation of its computed reference frame.

To allow this data to be easily utilized within a GPL procedure, the **Loc** property returns a Cartesian **Location Object** that is computed from the PreciseVision tool results but has been translated into the robot's world reference frame. This translation is defined by PreciseVision's camera calibration data and the camera mounting (e.g., stationary, or mounted on the robot). This **Location** can then be used as the reference frame for gripping a part or can be combined with other data to perform further analysis.

Please see the PreciseVision manual for information on which vision tools return these properties and how to interpret this data.

Examples

```
Dim vresult As VisResult
Dim visloc As Location
Dim x, y, z As Double
vresult = vobject.Result() ' Get a tool's results
visloc = vresult.Loc       ' Get position/orientation output
x = visloc.X
y = visloc.Y
z = visloc.Z
```

See Also

[Vision Classes](#) | [visresult_object.Info](#)

visresult_object.Type Property

Returns an **Integer** type code from a vision result object.

```
...visresult_object.Type
```

Prerequisites

None

Parameters

None

Remarks

This method returns the numeric Type code for a vision result object. Currently, all vision results are of type 0, so this property always returns 0.

This property will be used in the future to enhance the **VisResult** class.

Examples

```
Dim vresult As VisResult
vresult = vobject.Result()
If vresult.Type = 0 Then
    ...
```

See Also

[Vision Classes](#)