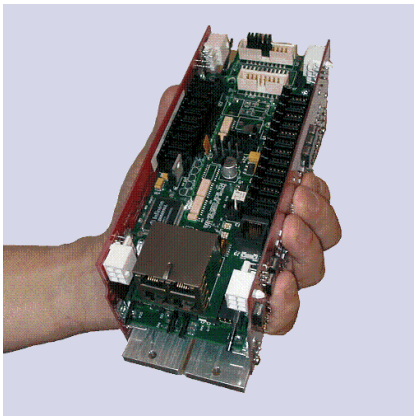
PRECISE AUTOMATION

The Guidance Programming Language



Introduction to GPL

Version 2.0.1, March 19, 2008
P/N: GPL0-DI-00010

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

Document Content

Trademarks

Disclaimer

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

The Guidance Programming Language

1. GPL Overview

This document introduces you to the Guidance Programming Language, GPL. GPL is a full-featured language designed to allow you to program and automatically operate motion controllers with machine vision and the mechanical mechanisms (“robots”) that are controlled by these devices.

GPL can be employed in a wide variety of applications including: general robotics; mechanical assembly; material handling and packaging; palletizing; carton loading or case packing; wafer handling or machine control in the semiconductor industry; life sciences equipment applications; or applications requiring conveyor tracking and/or vision guidance.

This language can be easily applied to a wide range of mechanisms ranging from simple, single axis linear and rotary devices, to complex robots that require all of their axes to be simultaneously moved in a coordinated Cartesian fashion, to systems that have multiple robots that operate either independently or cooperatively. The control hardware for such systems can reside in a single box or can be distributed in a networked control architecture. Independent of the physical control architecture, GPL makes use of its built-in networking ability and knowledge of robot geometries (kinematics) to allow mechanisms to be centrally programmed and easily controlled in Cartesian coordinates.

To support such a wide range of applications and mechanisms, GPL has extensive motion control facilities including: blending of joint and Cartesian interpolated motions (“continuous path”); s-curve profiles; base and tool offsets; built-in kinematic models for a variety of robots; mathematics for manipulating robot and part positions and orientations; and frames of reference including moving frames of references for conveyor tracking.

GPL has been targeted to execute on the Precise Automation Guidance Controller, which supports a networked control architecture. This controller includes a web based operator interface, a unified configuration and parameter database, integrated data logging capabilities, Category 3 safety circuitry, and a number of facilities that simplify both local and remote diagnostics and maintenance.

The Guidance Controller can in fact be programmed using three different methods: (1) a forms based teach-and-repeat technique that executes “MotionBlocks” in response to digital input signals; (2) GPL as described in this document; or (3) by any standard Windows PC language, which remotely controls the system via a TCP/IP connection. The MotionBlocks method is ideal for simple applications, especially those where a PLC is providing overall cell control, and is extremely easy-to-use since no programming language knowledge is required. GPL has the advantage of being embedded within the controller and allows more complex applications to be addressed while still permitting the controller to be operated in a standalone mode. The TCP/IP method allows programmers to leverage the capabilities of a PC (or other standard computing platforms) at runtime and to utilize the language of their choice.

In this document, we describe the features and syntax for the embedded system, the “Guidance Programming Language” (GPL).

GPL is a full-featured programming language. The fundamental syntax for GPL has been modeled after object-oriented forms of the Basic Language in order to provide a syntax and development environment that are familiar to many application developers. The Basic syntax has been extensively augmented with “classes” and “objects” that implement the motion control and vision capabilities. A Windows PC is required to develop and debug application programs but need not be connected when the controller is operating in automatic mode. Programmers who are familiar with Visual Basic .Net 2003 should be very comfortable with many of the computational and structural elements of GPL.

In the following sections, an introduction and overview of the GPL syntax is provided. Where it is important, we point out differences between GPL and the various variants of the Basic Language. These notes are highlighted by enclosing them within square brackets (“[]”). For more detailed information on individual instructions, objects and classes, methods, and properties, please see the GPL Dictionary document.

2. Statement structure

2.1. Program lines can begin with an optional line label. Line labels must either be a valid variable name (e.g. label1) or an integer literal (e.g. 100). Line labels must always be followed by a colon (:). The label and colon can optionally be followed by a standard statement. [In VB6 and some other version of Basic, no label separation character was required.]

2.2. The standard line is formatted as follows:

```
Label:    Statement        ' Comment
```

2.3. An apostrophe (') marks the beginning of a comment. Comments can follow a standard statement on a line. Full line comments and blank lines are permitted.

2.4. Only one statement is permitted per line but a single statement can be continued on multiple lines. To continue a line, the end of the line must contain a space character followed by an under bar (“ _”). Comment lines cannot be continued and lines cannot be broken at certain points (e.g. in the middle of a variable name).

2.5. There is no termination character at the end of a statement.

3. Data Type and Variables

3.1. Basic Data Types

3.1.1. The following table describes the basic data types that are supported in GPL:

Supported Data Types	
Boolean	**True** (<>0) or **False** (=0) values.
Byte	Unsigned 8-bit integer numbers ranging from 0 to 255 in value.
Short	Signed 16-bit integer numbers.
Integer	Signed 32-bit integer numbers.

Single	32 bit single precision floating point numbers.
Double	64 bit double precision floating point numbers.
String	**String** variables can have values that are of arbitrary length
Object	Universal data types for object oriented Basic. It's a pointer to any type of data. Can hold any data type value plus all built in system structures/classes are represented as objects. See the section later for a general description of **Objects**.

3.1.2. The following data types found in VB.Net and VB6 are not supported:

Unsupported VB6/VB.Net Data Types	
Long or **Int64**	64-bit signed integer number
Decimal	96-bit signed integer scaled by a power of 10
Int16	Synonym for Short
Int32	Synonym for Integer
Char	16-bit Unicode
Variant	Old universal data type in VB6
Date	Date and time values

3.1.3. Identifier type characters and literal type characters, which are special postfix characters used to specify the type of variables and literal constants are not supported. For example, in other systems, 725**L** identifies 725 as a **Long** constant and "**Dim** Abc**!**" Declares Abc to be of type Single.

3.1.4. In general, the system automatically converts one type of variable to another as needed. For example, all integer types (**Boolean, Byte, Short, Integer**) are automatically converted to double precision floating point values when used in floating point expressions. However, when necessary, the following explicit conversion functions can be utilized to force a specific type conversion. These functions are all described in greater detail in the Software Reference Section.

Explicit Type Conversion Functions	
CBool	Converts any numeric type or **String** to **Boolean.**
CByte	Converts any numeric type or **String** to **Byte.**
CDbl	Converts any numeric type or **String** to **Double.**
CInt	Converts any numeric type or **String** to **Integer.**
CShort	Converts any numeric type or **String** to **Short.**
CSng	Converts any numeric type or **String** to **Single.**
CStr	Converts any numeric type to **String.**
Hex	Converts an **Integer** value to **String** in Hexadecimal format.

3.1.5. All input characters are represented as 7-bit ASCII. Extended 8-bit ASCII and Unicode characters are not accepted in symbol names or in string literals.

3.1.6. Hexadecimal constant values are indicated by the prefix "**&H**". This syntax can cause confusion with the "**&**" concatenation operator. For example, if you have a variable named "HEAD" then the expression: String **&H**EAD causes a syntax error since **&H**EAD is interpreted as the hex value "EAD". To avoid this problem, insert a space after "**&**" if it is being used as a concatenation operator.

3.1.7. Octal constant values are indicated by the prefix "**&O**". This syntax can causes confusion with the "**&**" concatenation operator. For example, if you have a variable named "O2" then the expression: String **&O**2 causes a syntax error since **&O**2 is interpreted as the octal value "2". To avoid this problem, insert a space after "**&**" if it is being used as a concatenation operator.

3.1.8. As a programming convenience, there are a number of constant values that are predefined in the language. These constants all begin with "GPL_". These constants are listed in the following table and their use is described in the language dictionary pages.

GPL Constant Values	
GPL_CR	ASCII carriage return character (13).
GPL_LF	ASCII line feed character (10).
GPL_Righty	Assert right shouldered configuration (&H01).
GPL_Lefty	Assert left shouldered configuration (&H02).
GPL_Above	Assert elbow above wrist configuration (&H04).
GPL_Below	Assert elbow below wrist configuration (&H08).
GPL_Flip	Assert wrist pitched up configuration (&H10).
GPL_NoFlip	Assert wrist pitched down configuration (&H20).
GPL_Single	Assert restrict wrist position to within +/- 180 degrees (&H1000).

3.2. Variable Declarations

3.2.1. Variable names can be mixed case (upper and lower case characters), but names are not case sensitive, i.e. Abc, ABC, abc, aBC all refer to the same variable.

3.2.2. Within a given context, variable names must be unique even if they refer to variables of different data types and variable names cannot match system keywords. For example, you cannot have a string variable named “value1” and an integer variable with the same name. System keywords generally refer to words such as “**For**”, “**If**”, “**Dim**” that are expected to denote a built-in language capability.

3.2.3. Variable names must start with either a letter or an underscore “_”. This character can be followed by a sequence of up to 127 additional letters, numbers, and underscore characters for a total of 128. If a variable name starts with “_” it must be followed by at least one other character other than another underscore to distinguish it from a line-continuation.

3.2.4. **Dim** is the basic data type declaration statement within procedures for local, i.e. automatic, variables. If **Static** is used in place of **Dim** within a procedure, the value of the variable is preserved from one execution of the procedure to the next. **Dim** variables, including array variables, are initialized to 0 (numbers), **False** (**Booleans**), or **Nothing** (structures, objects, or classes), each time their enclosing procedure is executed.

```
Dim ii As Short
Static jj As Short
```

3.2.5. Variables defined within a module, outside of a procedure, are accessible by all procedures in the module and, like **Static** variables, their values are preserved independently of the execution of any procedure. If such variables are defined with **Private** or **Dim,** the variables are local to the module and cannot be accessed by procedures in other modules. If **Public** is used instead to declare a variable, the variable is accessible by all procedures within all modules loaded into the controller’s memory.

```
Module Test
    Dim Count As Integer       ' Invisible to other modules,
                               ' global in this module
    Private nBlocks As Integer ' Same as declaration above
    Public TotalArea As Single ' Visible to all procedures in
                               ' all modules within project.
End ModulePrivate
```

Variables declared within a module can also be accessed by preceding the variable name with the module name. This method of specifying a variable is required for cases when the same **Public** variable name is found in more than one module and it is unclear from the name alone which variable is being referenced.

```
Module Test1
    Public aa As Integer
    Public bb As Integer
End Module

Module Test2
    Public a As Integer
End Module

Module Test3
    Sub MyProc
        ii = bb           ' Okay since there is only bb
        ii = Test1.bb     ' Okay but not necessary
        ii = aa           ' Error since aa is duplicated
        ii = Test1.aa     ' Okay since clear which aa
    End Sub
End Module
```

3.2.6. In GPL, no matter where a variable is declared in a procedure, the scope of variables extends throughout the procedure with the restriction that variables can only be declared in the outermost level of a procedure.

```
For ii = 1 To 10
Dim jj As Integer         ' Not allowed
kk = ii                   ' Forward reference to kk is allowed.
Next ii

Dim kk As Integer
```

In the future, we may change the scoping rules to follow other variants of Basic, such as VB.NET, more closely.

3.2.7. Multiple declaration clauses may appear in a single statement.

```
Dim n As Integer, x As Double
```

3.2.8. The data type must always be specified.

```
Dim BlackObject1            ' Invalid
```

3.2.9. If multiple variables are declared within a single statement, if a variable's type is not specified, its type is defined by the next type definition in the statement [this is different from VB6 where all untyped variables became **Variants**]. Note, if a **New** clause (see below) is used, only a single variable name may appear to the left of the **As** keyword.

```
Dim ii, jj As Integer       ' Both ii and jj are of type Integer
```

3.2.10. Variable or constant values may be initialized by adding an initialization clause that beings with an "=". For example,

```
Dim Count As Integer = 1    ' Sets Count to 1
```

Each time this statement is encountered during execution, its value is initialized. If an initializer clause is used, only a single variable name may appear to the left of the **As** keyword.

An arbitrary expression may appear to the right of the "=". If the variable being initialized is an object or structure, a **New** keyword may appear to the right of the "=".

Be careful if you call a user-defined function as part of the initializer expression since some variables may not be initialized yet.

Module-level variables are initialized once when a project is started and are processed in the order in which they appear in the module. They are initialized before any user-defined procedures are executed (except in the case where you call a user-defined function from an initializer). Errors that occur while initializing variables are listed as part of a hidden procedure named "_Init".

3.2.11. The **New** keyword may appear in a clause that declares an object or structure. The **New** may appear immediately after the **As** keyword, or may appear immediately after the "=" in an initialization clause. **New** may not appear in both places within the same statement.

```
Dim Loc1 As New Location             ' Creates a location
                                     ' class instance
Dim Loc1 As Location = New Location  ' Equivalent to above
```

3.2.12. A **Const** keyword indicates that the variable is read-only and cannot be changed during normal execution. Only the initialization clause can set the value of the **Const** keyword.

```
Const MaxCount As Integer = 10
MaxCount = MaxCount+1                ' Invalid
```

3.2.13. GPL only supports strong typing, i.e. all variables must be declared in a **Dim**, **Static, Private, or Public** statement although the specific type of a variable may be excluded and will be automatically set to the default. [VB.Net allows strong typing to be disabled with the "**Option Strict Off**" statement.]

3.3. Data Type Arrays

Any of the data types described above, including objects, support array variables. The rank (dimension) of an array can be from 1 to 4. The number of array elements within a dimension is limited by available memory for **Static** arrays, and by thread stack size for normal dynamic arrays.

3.3.1. The first index in an array is always element 0. When you declare an array size, you are specifying the upper bound for a dimension. So, the number of elements for a dimension is always equal to the upper bound+1. For example:

```
Dim Count(9) As Integer    ' Allocates array of 10 elements
```

Versions of Basic such as VB6 supported means for defined ranges of indices that started with an arbitrary first index number (e.g. "10 to 20") and also statements such as "**Option Base 1**" that forced the first index to always be 1. However, VB.Net always starts arrays with index 0 and this is the convention that is supported in GPL.

3.3.2. The **Dim** statement is used to declare an array variable. The supported forms of this statement are as follows:

```
Dim MyArray(3, 4) As Integer
Dim MyArray(,) As Integer
```

The first statement specifies a 2-dimensional array with 4 elements in the first dimension and 5 in the second, for a total of 20 elements. These elements are allocated when the **Dim** statement is executed.

The second statement simply specifies a 2-dimensional array, but does not allocate any elements. Before you can use the array, you must either assign an array to it, or you must use a **ReDim** statement to allocate the elements.

When array elements are allocated, numeric arrays have the value 0 and object arrays have the value **Nothing**. Initialization of array values using an "=" clause is not supported in GPL.

3.3.3. Once an array has been declared and its dimensionality established, the **ReDim** instruction can be used to initialize or change the number of elements within any dimension. **ReDim** can be applied to any array, so no distinction is made between dynamically sizeable arrays and fixed arrays. However, **ReDim** cannot be used to change the rank of an array and **ReDim** cannot be used to initially declare an array. Some examples of **ReDim** are as follows:

```
Dim Count() As Integer
ReDim Count(9)
Dim TwoDCount(2,3) As Integer
ReDim TwoDCount(1,100)
```

3.3.4. Whole arrays may be assigned to each other with a single statement. When that occurs, the data is not actually copied, but a pointer to the data in the right-hand array is copied to the left-hand array variable so that both array variables access the same data. This behavior is similar to object variables. For example:

```
Dim CountA(9) As Integer
Dim CountB() As Integer
CountB = CountA             ' CountB now refers to the same
                            ' data as CountA
```

3.3.5. When single array elements are passed as procedure arguments, they behave the same as non-array variables of the same type. When whole array elements are passed as procedure arguments, pointers to either the array value (**ByVal**) or the array variable (**ByRef**) are passed, and the behavior is the same as when passing objects.

3.3.6. All arrays of variables are members of the built-in **Array** class. You can use properties of this class to determine the properties of any variable array.

Property	Description
array.**GetUpperBound** (*dim*)	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.
array.**Length**	The total number of elements in the entire array, in all dimensions.
array.**Rank**	Returns the rank, which is the number of dimensions, in the array.

These property methods may only be used with an entire array, not with a subset or individual array element.

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.

3.4. Scope of Names

Variables, constants, and procedures all have names. The section of a project where these names are known is called the *scope* of the name. Attempts to access a name outside its scope results in an "Undefined symbol" error because a valid name cannot be found by the compiler.

3.4.1. In general, a name is known within the block where it is declared, and within any blocks contained in the block where it is declared. For example, a variable declared in a procedure is known only in that procedure, but a variable declared in a module is known in all procedures contained in that module.

3.4.2. To access a name from outside the block where it is declared, the name must be declared as **Public**. Public names can be accessed from anywhere, provided that the path to the name is fully specified. As a special case, **Public** module-level names may be accessed without the module name being specified, provided that the name is unambiguous in all modules.

For example:

```
Module MyMod
    Public ModVar As Integer
    Public Class MyClass
        Public Shared MaxSize As Integer
        Private Shared Size2 As Integer
    End Class
End Module

Module GPL
    Public Sub Main
        MyClass.MaxSize = 100         ' Invalid, path not complete
        MyMod.MyClass.MaxSize = 100   ' Okay
        MyMod.MyClass.Size2 = 100     ' Invalid, private variable
        ModVar = 20                   ' Okay, special case
    End Sub
End Module
```

4. Objects and Classes

4. Objects and Classes

“Objects” and “classes” are the basis of object-oriented programming. A class defines a collection of related data and the procedures that operate on that data. In a sense, a class can be thought of as a template. If multiple copies (or “instances”) of a class are required to store distinct sets of data, objects of that class are created.

Objects and classes are used within GPL to provide additional functionality that is not part of the standard Basic Language and to organize functions that are related into easy-to-access groups. This functionality includes: mathematical operations, I/O operations, motion specifications, and robot control.

This section describes the general concepts associated with objects and classes. For illustration purposes, some of the objects and classes that are built into GPL are mentioned briefly in this section. The detailed description of these built-in GPL objects and classes are provided in later sections.

4.1. Objects, Fields, Properties and Methods

An object is a collection of related data and the procedures that operate on the data.

As opposed to a traditional data array, objects can and normally do contain many different types of data. For example, the GPL **Location Object** that represents robot and part positions contains an array of **Double** values to store a position and orientation, an **Integer** value for special flag bits, a **Boolean** to indicate a choice of reference frames, a pointer to another object, plus other data. The values stored within an object are called "*fields".* Generally, *fields* are accessed via "*properties*" of the object. The *properties* provide read and write access to field values and allow the values to be formatted, processed or grouped. Each *field* can have one, multiple, or no *properties* associated with it. For example, several *properties* of the **Location Object** access the same position and orientation *field* data to allow the data to be presented as individual axis positions or a set of all axes positions or a Cartesian position and orientation depending upon how the **Location** is defined.

From a data point of view, objects are similar to C structures. However, in addition to grouping data, objects also have specific procedures defined for operating on the object's *fields*. These object-specific procedures are called "*methods".* For example, the **Location Object** has *methods* for inverting its Cartesian position and for combining the positions of two **Location Objects**.

Depending upon how they are defined, some *methods* operate like subroutines while others return values like functions. If a *method* returns a value, it can be used in any expression that is appropriate for the type of its returned value. If a *method* operates like a subroutine, it must appear in a statement by itself and cannot appear within an expression. Either type of method can have a list of required arguments in the same manner as subroutines and functions.

More generally, *fields*, *properties*, and *methods* are referred to as "*members"* of an object or class. For the most part, you should only need to concern yourself with *properties* and *methods* of objects.

4.2. Classes of Objects

A class is a formal description and template for a type of object and defines its *fields*, *properties* and *methods*.

In general, there are two types of classes: *non-global* and *global*. A *non-global* class does not hold any data and relies upon its objects for data storage. Each object for a given class will have the same types of *members* but will contain an independent set of values for each *member*. For example, a typical robot application will have multiple **Location Objects**. Each **Location** will store the data that describes a specific part or robot position. However, all of the **Location Objects** will be derived from the same **Location Class** and will have the same types of *members.*

A *global* class is like a *non-global* class in that it defines all of the *fields*, *properties* and *methods* associated with this class. However, a *global* class is used when a single set or no set of data exists, so that a *global* class never has any objects. For example, many of the arithmetic functions (e.g. sine, cosine, square root) are part of the **Math Class** . This is done as a convenience to allow these functions to be grouped together and therefore easily accessed. However, the **Math Class** has no fields, no properties and no data, just methods. Consequently, the *global* **Math Class** has no objects.

4.3. The Dot "." Operator

Within GPL, a period character ".", also known as the *dot operator,* serves as a preface character to identify a member of a class or an object. To access a specific member of an object or class, you would write:

```
object.member  or  class.member
```

For *global classes*, since there are no objects, only the "*class*.**member**" form of reference can be used. For *non-global* classes, most references are to the values of objects and are written as "*object*.**member**", although the "*class*.**member**" form is permitted for certain methods.

By making use of the *dot operator*, *properties* of objects can be used in assignment statements and expressions in exactly the same manner as you would employ any other variable of the same data type. Also, the *dot operator* permits *methods* to be invoked in the same manner as you would invoke any subroutine or function.

Some examples of the *dot operator* are as follows:

```
Dim Pos_x, Value As Double
Pos_x = location_object.X+2  ' Get x-axis displacement + 2
location_object.X = 3        ' Set x-axis displacement property
Value = Math.Sqrt(3)         ' Sqrt is method of Math Class
location_object.Here         ' Invoke method to record position
```

The *dot operator* can be used multiple times in succession if a property or method returns another object. For example, the method that inverts a **Location** returns a **Location** value. So, the following could be written to first invert a **Location** and then extract the x-axis displacement of the result.

```
Pos_x = location_object.Inverse.X
```

4.4. Object Variables and the New Clause

While the members of an object can be treated like any other variable of the same data type, *object variables* are quite different from other variables. That's because an *object variable* does not contain the value of the object, it contains a reference (or "pointer") to the memory where the value is stored. For example, if we declare a **Location** variable:

```
Dim My_loc As Location
```

This statement creates a pointer, My_loc, to an object of the **Location Class**. However, at this time, the My_loc *object variable* has not allocated any storage for the value of the object and so its pointer is set to "**Nothing"**. If you attempt to access a member of My_loc at this time, an error would be generated. In general, before an object can be used, you must either allocate memory to the pointer (see below), copy a pointer to an existing value or call a method that returns a value pointer.

The standard way of creating ("allocating") an object value is by using a **New** clause. This clause may appear in a **Dim** statement or in an assignment statement and has the following syntax:

```
New class_name
```

where *class_name* is the name of the class for which you want to create an object value.

For example, the following three cases all declare a location *object variable* and allocate a **Location Object** value for it.

```
Dim My_loc As New Location            ' Create new location value
        -or-
Dim My_loc As Location = New Location ' Same as above
        -or-
Dim My_loc As Location                ' Declares variable only
My_loc = New Location                 ' Creates the location value
```

In general, if you are unsure of whether to allocate a data block or not, you should probably go ahead and allocate using the **New** clause. Using **New** unnecessarily will be somewhat less efficient, but GPL automatically takes care of managing allocated object values and so memory is never lost (i.e. you cannot create a memory "leak").

4.5. Copying Object Variables and Values

Since an *object variable* is a pointer to a value, the following simple assignment statement does not copy the value of an object, it copies an object pointer:

```
My_loc = Another_loc
```

At the conclusion of this instruction, My_loc and Another_loc both point to the same object value. Furthermore, if My_loc was the only pointer to a different object value, that object value will have been deleted ("deallocated").

This use of pointers allows some sophisticated programming techniques, but it can also be confusing. For example, after the assignment statement above, changing a *property* of either My_loc or Another_loc will alter the *property* as seen by the other object. For example:

```
Dim My_Loc1 As New Location   ' Create new location value
Dim My_Loc2 As Location       ' Does not create value
Dim tmp As Double
My_Loc1.X = 10
My_Loc2 = My_Loc1             ' Both Loc2 and Loc1 now
                              '  have the same value pointer
tmp = My_Loc2.X               ' tmp gets the value 10
My_Loc1.X = 20
tmp = My_Loc2.X               ' tmp now gets the value 20
```

4.5.1. Clone Method

Many classes include a **Clone** method to create an exact copy of an object. The value of the **Clone** method is a new object value that is the same as the referenced object. When this value is assigned to a variable, it is independent of the original object value.

For example:

```
Dim My_Loc1 As New Location   ' Create new location value
Dim My_Loc2 As Location       ' Does not create value
Dim tmp As Double

My_Loc1.X = 10
My_Loc2 = My_Loc1.Clone       ' Loc2 gets a copy of Loc1
tmp = My_Loc2.X               ' tmp gets the value 10
My_Loc1.X = 20
tmp = My_Loc2.X               ' tmp still gets the value 10
```

4.5.2. Nothing

The keyword **Nothing** is a built-in function that returns an object with no value. If you assign **Nothing** to an object variable, its previous pointer is removed and any attempt to access the variable results in an error. When an object variable is newly declared its value is **Nothing** unless a **New** clause was specified.

Assigning **Nothing** to an object variable releases the memory associated with the object value, provided it is not being used elsewhere.

4.6. Objects as Procedure Arguments

Like other variables and values, object values may be passed as procedure arguments. Object values are always passed as pointers, so the operation of **ByVal** and **ByRef** is a little different from that of other arguments.

4.6.1. ByVal

When an object value is passed **ByVal**, a pointer to the object *value* is passed to the called procedure. That means that changes made to the value via the called procedure parameter are seen by the caller. But changes made to the *variable* are not seen by the caller.

For example:

```
Sub My_Sub (ByVal Loc As Location)
    Loc.X = 20                ' Changes original value
    Loc = New Location        ' Create new value locally
    Loc.X = 30                ' Changes local value
End Sub

Sub Main()
    Dim Loc1 As New Location  ' Create new location value
    Dim Loc2 As Location      ' Does not create value
    Dim tmp As Double
    Loc2 = Loc1               ' Copy value pointer
    My_Sub ( Loc1 )           ' Pass pointer to Loc1 value
    tmp = Loc1.X              ' Gets 20 from original value
    tmp = Loc2.X              ' Gets 20 from original value
End Sub
```

4.6.2. ByRef

When an object value is passed **ByRef**, a pointer to the object *variable* is passed to the called procedure. That means that changes made to either the *value* or the *variable* via the called procedure parameter are seen by the caller.

For example:

```
Sub My_Sub (ByRef Loc As Location)
    Loc.X = 20                ' Changes original value
    Loc = New Location        ' Caller variable changed
    Loc.X = 30                ' Changes new value
End Sub
Sub Main()
    Dim Loc1 As New Location  ' Create new location value
    Dim Loc2 As Location      ' Does not create value
    Dim tmp As Double
    Loc2 = Loc1               ' Copy value pointer
    My_Sub ( Loc1 )           ' Pass pointer to Loc1 variable
```

```
    tmp = Loc1.X              ' Gets 30 from new value
    tmp = Loc2.X              ' Gets 20 from original value
End Sub
```

4.7. User-Defined Classes

In addition to using the built-in classes, users can define their own classes within GPL. User defined classes are a very powerful feature that can be of assistance is organizing a GPL project. *However, for programmers that are not comfortable with object oriented programming, user defined classes do not need to be used and this section can be skipped.* More traditional arrays of numeric and string variables are supported in GPL and can be utilized to implement a complete application.

A user class definition begins with a **Class** statement and ends with an **End Class** statement. A class may be defined at the top level of a file, within a module, or within another class. User-defined classes serve as a template for objects that contain arbitrary variable fields and are associated with procedures that create and modify the objects.

Class variables, procedures, and nested classes can be declared as either **Public** or **Private**. By default these items are all **Private**. A **Private** item may not be referenced outside of the class in which it is declared. A **Public** item may be referenced outside of a class by using the syntax: *class_name.item_name* or *object_name.item_name*.

4.7.1. Class Variables

By default, variables declared within a class are templates for fields within objects of that class. Independent copies of these variables are found in each object of the class and do not exist outside of an object. If a non-shared class variable has an initializer, that field is set to the initializer value whenever an object is created.

If a class variable is declared **Shared**, only a single copy of the variable exists and is accessed independently of any objects. A **Public Shared** variable is normally referenced by the syntax: *class_name.item_name*, to emphasize that it is associated with the class and not the object. A **Public Shared** variable may also be accessed by the syntax: *object_name.item_name* which results in the same single value being referenced. The second syntax example should be avoided to prevent confusing it with a non-shared variable. If a **Shared** class variable has an initializer clause, the initialization occurs once when the main thread starts.

An internal **Sub** procedure named *_Init* is automatically generated to perform shared variable initialization. An internal **Sub** procedure named *_New* is automatically created to perform initialization when a new object is created. Do not attempt to create procedures with these names.

4.7.2. Class Procedures

Sub, **Function**, and **Property** procedures may all be members of a class. By default, procedures declared within a class are associated with an object of that class. They are invoked by the syntax: *object_name.procedure_name*. Within such procedures, fields and other procedures in the class may be referenced without specifying *object_name* as a prefix. Instead, the object that was referenced when the procedure was initially called is assumed.

If a class procedure is declared as **Shared**, it is not associated with any object, and may be invoked simply as *class_name.procedure_name*. Since there is no object associated with this procedure, it cannot reference non-shared fields or class procedures unless it explicitly includes an *object_name* as a prefix.

In the example below, the variable *count* is a field within the class *cc*. The procedure *Main* creates a new object, *aa*, of class *cc* and sets its *count* field to 5. When the *Inc_count* procedure is called, it is passed the object *aa*. When *Inc_count* executes, its references to *count* are actually references to the field *count* within the passed *aa* object.

```
Public Class cc
    Public count As Integer     ' Count is a field in a cc-class
obj
    Public Sub Increment
        count = count+1         ' Inc count field in the current
obj
    End Sub
End Class

Sub Main()
    Dim aa As New cc            ' Creates a new object of class cc
    Dim bb As New cc            ' Creates a new object of class cc
    aa.count = 5                ' Sets count field in the object aa
    aa.Increment                ' Calls Sub Increment for object aa
    bb.count = 20               ' Sets the field count in object bb
    bb.Increment                ' Calls Sub Increment for object bb
    Console.WriteLine(aa.count) ' Writes 6
    Console.WriteLine(bb.count) ' Writes 21
End Sub
```

Property procedures improve readability by allowing assignment statements to call procedures that get and set data values. Reading a **Property** value is very similar to calling a function that returns a value. Writing a **Property** value looks like an assignment statement. Read-only properties cannot be written, and write-only properties cannot be read.

A **Property** definition must contain a *get block* (that begins with a **Get** statement and ends with an **End Get** statement) or a *set block* (that begins with a **Set** statement and ends with an **End Set** statement) or both. When a **Property** value is read, the *get block* procedure is executed. When a **Property** is written, the *set block* procedure is executed.

In the example below, the **Property** *Size* is defined to get and set the internal field value *size_in*. Additionally, the **Set** block clips the value to make sure that *size_in* is always in the range 0 to 10. Since *size_in* is declared as **Private**, it cannot be changed directly from the *Main* procedure.

```
Public Class cc
    Private size_in As Integer ' size_in is field in cc-class
    Public Property Size As Integer
        Get
            Return size_in    ' Simply return the field value
        End Get
        Set (value As Integer)
            If value > 10 Then
                value = 10
            ElseIf value < 0 Then
                value = 0
            End If
            size_in = value   ' Set clipped value in field
        End Set
    End Sub
End Class

Sub Main()
    Dim aa As New cc          ' Creates a new object of class cc
    Dim ii As Integer
    aa.Size = 20              ' Calls the Size Set Property
    ii = obj.Size             ' Calls the Size Get Property
    Console.WriteLine(ii)     ' Displays value 10
```

```
    aa.size_in = 5            ' Invalid since size_in is Private
End Sub
```

4.7.3. Me Object

When a non-shared class procedure is called, it is automatically associated with an object. This object is used implicitly whenever a non-shared procedure or field from the current class is referenced. This associated object may be accessed directly by the built-in object **Me**. This object always has the type of the current class. You can use the **Me** object when calling procedures that require an object as a parameter. If you attempt to use **Me** in a shared procedure, or one not associated with a class, an exception occurs.

4.7.4. Constructors

When an object is created with a **New** keyword, all fields in the new object are normally set to 0 (for numeric fields), empty (for string fields), and undefined (for object fields).

If a **Sub** procedure named *New* is defined for a class, it is automatically called whenever a new object is created. The *New* procedure may include an argument list. There may be multiple overloaded *New* procedures, each with a different argument list.

For example:

```
Public Class cc
    Public count As Integer       ' Count is field in cc-class
    Public Sub New
        count = 25                ' Set count to 25
    End Sub
    Public Sub New (value As Integer)
        count = value
    End Sub
End Class

Sub Main()
    Dim aa As New cc              ' Calls first New procedure
    Dim bb As New cc(15)          ' Calls second New procedure
    Console.WriteLine(aa.count)   ' Writes 25
    Console.WriteLine(bb.count)   ' Writes 15
End Sub
```

4.8. Limitations

All objects in GPL must have an explicit class specified. You cannot simply declare a variable as type **Object**. That means that late binding of objects is not supported.

5. Arithmetic Operations

5.1. Arithmetic Expressions

The following table documents the order in which elements of an arithmetic expression are evaluated (i.e. the order of precedence). The operations are presented in their order of precedence starting with the highest precedence, that is, those elements that are evaluated first. For operators that have an equal

precedence, elements are evaluated left-to-right. Parentheses can be used to change the order of evaluation. Operations within parentheses are always evaluated before operations that are outside of the parentheses.

Operation	Symbol	Notes
Exponentiation	**^**	Raises a value by a specified power. For example "x **^** 3" cubes the value of x. Powers have to be integer numbers if the number being operated on is negative. Otherwise, powers can have fractional parts.
Unary negation	**-**	This is a negative sign in front of a variable or constant that does not indicate a subtraction operation. For example, 2 * -4 is valid and produces a value of –8.
Multiplication/division	*****, **/**	This indicates the standard multiplication and division operations. For division, even if the divisor and the dividend are integer values, the result is computed as a real number with a fractional part.
Integer division	****	This indicates an integer division operation. Independent of the data type for the divisor and dividend, the result is truncated to an integer number. For example, "2.3 \ 2" yields a value of 1.
Modulus calculation	**Mod**	Computes the modulus of two numbers. For "x **Mod** y", this is equivalent to dividing x by y and returning the remainder. For example, "13.3 **Mod** 2" is equal to 1.3.
Addition/subtraction	**+**, **-**	Standard addition and subtraction operations. Automatically converts integer values to floating point and computes the result in floating point. If the value is stored into an integer variable type, the resulting answer is converted to integer before storage.
String concatenation	**+** or **&**	Either of the two symbols can be used to indicate string concatenation. However, the use of **"&"** is preferred in place of **"+"** to clearly specify a string concatenation operation instead of numerical addition.
Arithmetic bit shift	**<<**, **>>**	These are arithmetic shift operations and not bit rotations or logical shifts. For left shifts, a 0 is always shifted into the low-order bit. For right shifts, for positive numbers a 0 is shifted into the high-order bit and a 1 is shifted in for negative numbers.
Relational comparisons	**=**, **<>**, **<**, **>**, **<=**, **>=**	The six relational operator symbols represent "equal to", "not equal to", "less than", "greater than", "less than or equal to", and "greater than or equal to". The operands to the left and right of the relational operators can either both be numerical or string values.

Logical NOT	**Not**	Converts a **False** (=0) value to **True** (-1) and any **True** (<>0) value to **False** (0).
Logical or bitwise AND	**And, AndAlso**	Performs a logical AND operation unless either of the operands is not a **Boolean** value, in which case, a bitwise operation is performed. All operands of **And** are always evaluated even if an earlier operand has already decided the result. **AndAlso** prematurely stops evaluation if the result is already **False**. The following illustrates the logical AND operation: **True And True** -> **True** **True And False** -> **False** **False And True** -> **False** **False And False** -> **False**
Logical or bitwise OR	**Or, OrElse**	Performs a logical OR operation unless either of the operands is not a **Boolean** value, in which case, a bitwise operation is performed. All operands of **Or** are always evaluated even if an earlier operand has already decided the result. **OrElse** prematurely stops evaluation if the result is already **True**. The following illustrates the logical OR operation: **True Or True** -> **True** **True Or False** -> **True** **False Or True** -> **True** **False Or False** -> **False**
Logical or bitwise XOR	**Xor**	Performs a logical Exclusive Or operation unless either of the operands is not a Boolean value, in which case, a bitwise operation is performed. The following illustrates the logical XOR operation: **True Xor True** -> **False** **True Xor False** -> **True** **False Xor True** -> **True** **False Xor False** -> **False**

5.1.1. In general, most arithmetic expressions evaluation with GPL is performed in double precision floating point. For example, when two numbers are added together, they are first converted to **Double**'s if necessary and then the addition operation is performed. The results of expressions are converted to the appropriate data types when a variable is assigned a value. Because of this, GPL generally executes more quickly when variables are declared as Double's than the other types of numeric values.

5.2. Arithmetic Functions and Methods

The following tables summarize the standard arithmetic and trigonometric operations that are provided in GPL. As a convenience during editing, the operations within the first table are provided as methods of the **Math Class**. This allows programmers to display a pick list of the **Math** methods and easily see all of

operations that are available. The second table documents functions that are not part of the **Math Class.** These functions are provided in this manner for compatibility with other Basic Languages.

As is standard in GPL, conversions between different arithmetic types, e.g. **Integer**, **Single**, **Double**, are automatically performed as required. So, it is not necessary to have different variations on these methods and functions to deal with the different possible mixes of input parameter data types. Also, these methods and functions generally produce results that are formatted as **Double**'s. Results are 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.

For more information on these methods and functions, please see the Reference Documentation section.

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

Built-in Functions	Description
Fix(*number*)	Returns the integer portion of any numeric type by truncating towards zero.

Int(*number*)	Returns the integer portion of any numeric type by truncating towards negative infinity.
Rnd(*seed*)	Returns a pseudo random number.

6. Strings and String Expressions

String variables, assignment statements, and expressions provide the means for storing and manipulating text within GPL. As such, **Strings** are also the primary means for transferring data in and out of the system via the serial communications ports, the file system, and the Ethernet interface.

6.1. **String** variables store a series of ASCII characters and can be of arbitrary length. However, **String** operations have been optimized to execute most efficiently on **Strings** that are 128 characters or less in length.

6.2. **String** constants must be delimited by double quote marks, e.g. "Hello world", and can at most be 128 characters in length. To embed a double quote mark within a **String** constant, enter two double quote marks in a row, e.g. "Tom said, ""Hello world""".

6.3. As with other variables, **String** arrays are supported and the values of procedure level **String** variables can be initialized in **DIM** statements. For example,

```
Dim name As String = "Charlie"
```

6.4. A number of easy-to-use functions are provided for converting between **String** values and numerical values, e.g. **CStr, CDbl, CInt**. Each of these built-in functions was described earlier in the section on Basic Data Types.

As a convenience, GPL automatically converts a **String** value to a **Double** whenever a numerical value is expected and a **String** is encountered instead. For example, the following statements are legal:

```
Dim a As Double
a = 2.34 + "1.01"      ' Legal. a will be equal to 3.35
```

However, it is generally better practice to utilize the explicit conversion routines rather than relying upon the automatic conversions. The automatic conversions can result in some computations whose results may not be clear.

6.5. In most cases, when a **String** value is required as an input, a **String** expression can be provided. A **String** expression can consist of a **String** variable, constant, function or method or a concatenation of two or more of these **String** elements.

6.6. Two or more **String** elements can be concatenated together by utilizing the concatenation operator, "**&**". Also, for compatibility with other Basic compilers, the "**+**" can alternatively be used to indicate concatenation. However, given the automatic **String** to numeric conversion features of the language, the use of the "**+**" can make it less obvious whether a statement is intended to produce a **String** or a numeric result. Therefore, the use of the "**&**" concatenation operator is recommended over the "**+**".

The following is an example of **String** concatenation.

```
Dim s1, s2 As String
s1 = "Joe's"
s2 = s1 & " balance: " & CStr(10.2)   ' s2 = "Joe's balance: 10.2"
```

6.7. Since **String** values are often generated by appending additional text on to the end of the value of a **String** variable, for computational efficiency, the concatenation assignment operator is supported. For example,

```
s1 &= " more"    is equivalent to    s1 = s1 & " more"
```

The advantage of the concatenation assignment operator is that appended text is directed added onto the end of the variable's value. In the standard assignment statement, the value, s1, is copied to an intermediate variable where it is concatenated with the appended **String** value, " more". The resulting value then replaces the original value of the variable.

6.8. The values of two **Strings** can be compared using the **String.Compare** method. In addition, **Strings** can be compared using the standard arithmetic relational comparison operators (**=**, **<>**, **<**, **>**, **<=**, >=). Comparisons performed using the relational operators are always performed case sensitive, i.e. “A” is not equal to “a”. This is equivalent to specifying “**Option Compare Binary**” in some Basic compilers. To perform case insensitive comparisons, use the **Compare** method or force both **String** values to be upper or lower case.

6.9. Internally, **String** variables are implemented using many of the same procedures as those that apply to **Objects.** Consequently, many of the basic string manipulation operations are provided as methods and properties that can be applied to **String** variables. However, unlike other built-in **Objects**, when a **String** variable is created, it automatically has its data storage allocated. So, the use of the **New** qualifier is not needed in connections with **String** variables and is not permitted.

The following table summarizes each of the **String** methods and properties.

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 *string* variable and returns the starting position if found (0-n).
*string***.Length**	Property	Returns the number of characters in the **String.**
*string***.Split**	Method	Divides the *string* 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 *string* variable starting at a specific character position and with a specified length.
*string***.ToLower**	Method	Returns a copy of the *string* with all lower case characters.
*string***.ToUpper**	Method	Returns a copy of the *string* 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.

6.10. For compatibility with older Basic compilers, the following **String** functions are provided. In many instances, very similar functionality is provided by the **String** Members listed in the previous table.

Built-in String Functions	Description
Asc(*string*)	Converts the first character of a **String** to its equivalent ASCII numerical code.
Chr(*expression*)	Given a numerical ASCII code, a **String** that consists of the equivalent ASCII character code is returned.
Format(*expression, format_s*)	Converts a numerical value to a **String** value based upon a specified output format specification.
Instr(*start, string_t, string_s*)	Searches for an exact match of a substring within a **String** expression and returns the starting position if found (1-n).
LCase(*string*)	Returns a **String** value that has been converted to lower case.
Len(*string*)	Returns the number of characters in a **String**.
Mid(*string, first, length*)	Returns a substring of the *string* starting at the *first* character position and consisting of *length* number of characters.
UCase(*string*)	Returns a **String** value that has been converted to upper case.

7. Assignment Statements

The basic value assignment statements have the following form:

```
numeric_variable = arithmetic_expression       ' Comment
                        or
string_variable = string_expression            ' Comment
```

where the *arithmetic_expression* can be arbitrarily complex and can consist of variable values and functions inter-related by the arithmetic operations described in the previous section, and *string_expression* can be a string variable, string function, string valued property, string constant or concatenated string value.

7.1. For all arithmetic assignment statements, the result of the statement is always converted to the data type of the *variable* being assigned the new value. For example:

```
Dim a, b As Single, c As Integer
a = 2.25                        ' Assigned floating point value
b = 3.5                         ' Assigned floating point value
c = a * b                       ' Result of 7.875 rounded and stored
as 8
```

7.2. In addition to the standard assignment statements (e.g. x=2), assignment operators are provided that perform an operation on a variable value and store the result back into the variable value. For example:

```
 x *= 3              is equivalent to x = x * 3
```

The following table contains the list of assignment operators and their equivalents.

Assignment operator	Sample Use	Equivalent Code
^= Operator	x ^= y	x = x ^ y
*= Operator	x *= y	x = x * y
/= Operator	x /= y	x = x / y
\= Operator	x \= y	x = x \ y
+= Operator	x += y	x = x + y
-= Operator	x -= y	x = x – y
<<= Operator	x <<= y	x = x << y
>>= Operator	x >>= y	x = x >> y
&= Operator	x &= y	x = x & y

In addition to simplifying and clarifying complex program statements, the use of the assignment operators can be more efficient especially when the variable or property that is being assigned the new value has a complex array index or other specification or long string values are being concatenated. This is due to the fact that the full variable or property specification only has to be evaluated once to obtain the memory address of its value.

8. Control Structures

The statements described in this section alter the sequential execution of instructions within a procedure, i.e. they alter the flow of control. For example, these statements conditionally execute blocks of statements, repeatedly execute blocks of statements a fixed number of times or repeatedly execute blocks of statements until a condition is satisfied.

8.1. **GoTo** Statements

This instruction executes an unconditional branch and continues execution at a specified labeled statement.

```
GoTo label
```

A *label* must either conform to the conventions for a variable name (e.g. label3) or an integer literal (e.g. 1000). To label an instruction, the *label* is placed first on the line followed by a colon (:) followed by any standard instruction.

In general, **GoTo** instructions can make programs more difficult to understand. So, whenever possible, other control structures should be used in place of **GoTo**'s.

8.2. **If…Then…Else…End If** Statements

This control structure tests one or more expressions and conditionally executes at most one block of statements.

```
If condition Then
    if_statements
ElseIf elseif_condition Then
    elseif_statements
Else
    else_statements
End If
```

This control structure first tests the *condition* to determine if it is True (<>0) or False (=0). If True, the *if_statements* are executed and the remainder of the statements down to the End If are skipped. If False, the *if_statements* are skipped and the first ElseIf or Else, if present, is processed. If an ElseIf clause is present, its *elseif_condition* is tested and, if True, the associated *elseif_statements* are executed after which execution continues after the End If. Otherwise, the *elseif_statements* are skipped and the next ElseIf or Else is processed. If all conditional tests fail and an Else is present, the *else_statements* are executed.

8.2.1. An **If…Then** can contain several or no **ElseIf** clauses. If present, these must be specified before the optional **Else** clause.

8.2.2. An **If…Then** can only contain a single optional **Else** clause.

8.2.3. Since **True** is defined to be <>0, any arithmetic expression that evaluates to <>0 will be interpreted as a **True** condition.

8.2.4. For simple tests, this statement can be reduced to a single line format: **If…Then** *statement*.

8.3. **For…Next** Statements

This control structure executes a sequence of instructions a fixed number of times.

```
For variable = initial_value To final_value Step increment
    for_loop_statements
Next variable
```

This control structure begins by setting the *variable* to the *initial_value*. 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.

If the *initial_value* does not exceed the *final_value*, the *for_loop_statements* are executed once. However, if the *initial_value* exceeds the *final_value*, the *for_loop_statements* are skipped and execution continues at the statement following the **Next** instruction. If the *for_loop_statements* are executed, execution proceeds until the **Next** instruction is encounter. 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.

8.3.1. The *initial_value*, *final_value*, and *increment* can all be arbitrarily complex arithmetic expressions. However, these expressions are only evaluated when the **For** statement is executed and their values are saved for use by the **Next** statement. Therefore, if the values of these expressions change during the execution of the **For** loop it does not alter the saved values. Since these expression are only evaluated once, the **For** loop is generally more efficient that other looping methods.

8.3.2. The *increment* value is optional can be positive or negative. If positive, 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*. If not specified, a value of 1 is assumed.

8.3.3. The **For** loop can be prematurely terminated by executing an **Exit For** statement or a **GoTo** statement that branches outside of the **For** loop.

8.4. **While…End While** Statements

This control structure tests a condition and, if **True**, executes a block of statements repeatedly until the condition is **False**.

```
While test_expression
    while_statements
End While
```

This control structure begins by evaluating the *test_expression*. If the expression value is **True** (<>0), the block of *while_statements* is executed. When the **End While** is encounter, the *test_expression* is evaluated again. If the *test_expression* is still **True**, the *while_statements* are executed again. This sequence is repeated so long as the *test_expression* remains **True.** As soon as the *test_expression* tests **False** (=0), the *while_statements* are skipped and execution continues at the statement following the **End While**.

8.4.1. If the *test_expression* is **False** when the **While** begins execution, the *while_statements* are skipped and are not executed.

8.4.2. The **While** loop can be terminated before the conclusion of the *while_statements* by executing an **Exit While** statement or a **GoTo** statement that branches outside of the **While** loop.

8.5. Do…Loop Statements

This control structure 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**.

```
Do While | Until condition
    statements
Loop
          -or-
Do
    statements
Loop While | Until condition
```

8.6. Nested Control Structures

In general, control structures can be nested within each other to an arbitrary depth and in arbitrary combinations. For example, a **While** loop can be embedded within another **While** loop or an **If…Then** clause.

9. Procedures and Modules

9.1. Subroutines and Functions

The language includes user-defined subroutine (**Sub**) and function (**Function**) procedures. Functions are identical to subroutines except that a function returns a value and a call to a function can be included in an arithmetic expression. Except as noted, in this document, “procedure” or “routine” refer to both user defined procedures and functions.

9.1.1. Calling a Procedure

A **Function** or **Sub** may be invoked by placing its name as the first item on in a statement, or by using the **Call** keyword. If invoked in this way, the returned value of a **Function** is ignored. In addition, a **Function**, but not a **Sub**, may be embedded in an expression of the type returned by the function.

When invoking either a **Sub** or a **Function**, parentheses must always be provided around the argument list, with empty parenthesis supplied if there are no arguments. In VB6, parentheses were required if **Call** was used and forbidden if **Call** was not included. In VB.Net, parentheses are only optional for empty argument lists, although the Visual Studio.Net editor always inserts parentheses.

The following are some valid examples:

```
Call MyProcedure (1, 2, 3)  ' ()always required for non-null args
MyProcedure (1, 2, 3)       ' Call is optional
x = 2 * MyFunction (y)
MyFunction(y)               ' Do not care about the value
```

9.1.2. Returning from a Procedure

With executing a procedure, the procedure exits and returns control to the calling routine when one of the following is encountered:

1. The end of the procedure, marked by an **End Function** or **End Sub** statement.
2. An **Exit Function** or **Exit Sub** statement, depending on the procedure type
3. A **Return** statement.

If the top-level procedure exits, execution of that thread is terminated.

The returned value of a **Function** is specified by either an expression argument to the **Return** statement, or by assigning a value to the function name as if it were a variable. For example:

```
Function Test (ByVal x As Double) As Double
    If x < 10 Then
        Return x+1     ' Exits with a value of x+1
    Else
        Test = x+2     ' Sets the return value to x+2
        Exit Function  ' Exits with the current return value
    End If
End Function
```

9.1.3. Procedure Arguments

Scalar (non-array) arguments can either be passed to a procedure by value (**ByVal**) or by reference (**ByRef**).

ByVal means that a copy of the value is made for the called procedure. The called procedure may freely modify the argument variable without affecting the called program. By default, all arguments default to **ByVal**.

ByRef means that a pointer to the variable containing the value is passed to the called program. Only variables can be passed by reference. When the called procedure modifies its argument variable, it is actually modifying the value in the calling program.

Passing objects **ByVal** and **ByRef** has some subtle differences. In both cases, accessing and modifying members of the object have the same effect and change the same data. They are different for the case when you assign directly to the procedure argument. In the **ByVal** case, you only change the pointer to the value in the called procedure. In the **ByRef** case, you change the pointer to the value in the calling procedure's object variable.

9.1.4. Not Supported

The language does not support the **GoSub** statement. This statement allowed an arbitrary line within a procedure to act as the start of a procedure embedded within a procedure. Also, the language does not support declaring a procedure as **Static**. A **Static** procedure forced all of the local variables of a procedure to be statically defined such that they retained their values between calls. Variables must be individually specified as being **Static**.

The first version of GPL does not support **Optional** procedure arguments, initial argument values, or the **ParamArray** keyword. It also does not support passing a Set or Get Property as a **ByRef** argument.

9.2. Modules

A Module is a named section of code that begins with a **Module** statement an ends with an **End Module** statement. Modules may contain variable declarations, procedures, and class definitions. Modules can only appear at the top-level of a file. They cannot appear inside of other modules or classes.

9.2.1. Scope of Items within Modules

Modules provide a simple way to group variables, procedures, and classes, without concern about name conflicts.

Module variables, procedures, and classes can be declared as either **Public** or **Private**. By default these items are all **Private**. A **Private** item may not be referenced outside of the module in which it is declared. A **Public** item may be referenced outside of a module by using the syntax: *module_name.item_name*. As a special case, if *item_name* is unambiguous within all loaded modules, the *module_name.* may be omitted.

All variables declared within a module (and not within a class or procedure) are implicitly **Shared**, so they can be referenced within any procedure contained in the module. Consequently, only one copy of each implicitly **Shared** variable value can exist. All references to the variable access the same value. If a variable has any initializer clauses, the initialization occurs once when the main thread for the Module is started. **Const** symbols behave the same as variables, except their values cannot be changed once they are initialized.

9.2.2. Special Initialization Procedures

If a user **Sub** procedure named *Init* is defined within a module, it is executed as part of the module initialization, before the startup procedure begins.

An internal **Sub** procedure named *_Init* is automatically generated to perform module-level initialization. Do not attempt to create a procedure with this name.

10. Exception Handling

In automated systems, it is typically very important that the equipment be able to run unattended for long periods of time. Since errors and other unexpected events periodically occur, it is critical that the system be able to automatically field execution exceptions, attempt to correct the problem by responding in an appropriate manner, and continue execution if at all possible. In GPL, sections of procedures or entire procedures can be bounded by a **Try...Catch...Finally...End Try** structure that provides a formal means to intercept program exceptions and execute specific corrective actions. When an exception is handled in this manner, information on the type of exception is stored in an **Exception Object**.

10.1. **Try...Catch...Finally...End Try** Statements

In the group of instructions shown below, if an exception of any type occurs when the *try_statements* are executed, rather than halting execution and reporting an 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**.

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

10.1.1. 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*.

10.1.2. **Try** structures can be nested within each other. 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*. For example,

```
Public Sub MAIN
    Dim exc1 As New Exception
    Try
        test()
        Console.WriteLine("Test completed")  ' Never gets here
    Catch exc1
        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
```

In this sample code, the only output will be "Exception!". This is because the divide by 0 in *test* generates an exception, which terminates execution of *test*. If the call to *test* in the *MAIN* routine was not embedded within a **Try**, the system would normally halt the execution of the thread and report the error. Since the call is within a **Try** block that has a **Catch**, execution is instead continued at the first instruction within the

Catch block. This feature permits exceptions that occur within arbitrary depths of procedure calls to be fielded by a single **Try** structure.

10.1.3 A **Try** structure with a **Finally** instruction and no **Catch** instruction is only useful in a called procedure when a higher-level calling procedure contains a **Try** structure with a **Catch**. When an exception occurs in the *try_statements* of a called procedure with no **Catch**, the *finally_statements* are executed before the procedure exits to the higher-level procedure that contains the **Catch** statement. In the example above, if the divide by 0 statement was part of a **Try** block that was followed by a **Finally** block, the statements in the **Finally** block would have been executed prior to returning to the *MAIN* routine.

10.1.4 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*. For example,

```
    Dim exc1 As New Exception
    Try
retry:
        Move.Loc(loc1, profile1)
        Move.WaitForEOM
    Catch exc1
        If (exc1.ErrorCode = -153) Then
            profile1.Speed *= .9
            GoTo retry              ' LEGAL BRANCH
        End If
        GoTo bad_jump               ' ILLEGAL!!!
    End Try
bad_jump:
```

10.1.5 If an **Exit Try** statement 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*.

10.2. **Throw** Statement

The **Throw** statement can be used to force an exception within a program at any time. The syntax for this instruction is as follows:

```
Throw exception_object
```

In addition to forcing an exception to halt program execution, the **Throw** statement is often used within a *catch_statements* block to force an exception to be processed by a higher-level **Try** structure.

10.3. Exception Class and Objects

Whenever an exception occurs, the data that defines the specific type of exception is stored and passed in **Exception Objects**. There are two basic types of **Exceptions**: robot **Exceptions** and general **Exceptions**. Both forms have a numeric property that indicates the basic type of error. In addition, the robot **Exceptions** contain information on the robot and axis that is associated with the Exception. The general **Exceptions** contain an error code qualifier in place of the robot and axis information.

As with other types of **Objects, Exception Objects** are defined with a **Dim** statement or as an argument to a procedure. When an **Exception Object** is first created, normally the **New** token is used to allocate the data section for the **Object**.

All of the properties and methods for the **Exception Objects** are described in detail in the Reference Documentation section. The following table briefly summarizes this information.

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 *exception_obj.*
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 *exception_obj* properties.
exception_obj.**Qualifier**	Property	Sets and gets the error message qualifier for a general **Exception**.
exception_obj.**RobotError**	Property	Sets and gets the **Boolean** that indicates if an **Exception** is a robot or general type.
exception_obj.**RobotNum**	Property	Sets and gets the number of the robot associated with a robot **Exception**.

11. Motion and Controller Related Classes

11. Motion and Controller Related Classes

In the previous sections, the features of GPL that were described closely mimic those that are found in other object orientation variants of the Basic Language. Those features included arithmetic expression representations, control structures, variable types and declarations, mathematical functions, etc.

In the next sections, the features of GPL that have been added specifically to provide built-in motion control facilities are described. Consistent with the philosophy of object-oriented languages, these special features are provided as properties and methods of built-in "**Classes**". In some cases, the **Classes** are global system classes that simply serve to group features together as an aid in accessing and understanding these facilities. For global **Classes** there is a single copy of the **Class**. The **Math Class** that was described earlier is a good example of a global system class. In other cases, the classes have multiple instances (objects) that allow programs to have multiple copies of the objects, each with their own independent set of values for properties and methods. For example, in a Visual Basic program, the "Textbox" is a good example of the use of objects. An application can have multiple Textboxes each with different colors and sizes and other visual effects. In a motion application, robot locations are represented as objects to allow an application to store multiple robot and object positions, each with its own special properties.

The following table describes the motion control specific classes that are included in GPL. Each of these classes is discussed in more detail in the following sections.

Motion Control Class	Description
Signal Class (Global)	Reads and writes digital, analog and other simple means

	of input and output
Location Class and Objects	Defines positions and orientations of the robot and objects
Profile Class and Objects	Defines sets of parameters that specify the trajectory to be followed when moving between **Locations**.
Move Class (Global)	Provides the basic methods for executing a motion between **Locations** using **Profiles.**
RefFrame Class and Objects	Defines robot and part reference frames that can automatically alter the total (absolute) positions and orientations of **Locations**.
Controller Class (Global)	Provides access to general facilities provided by the motion control hardware such as power control, timers, etc.
Robot Class (Global)	Provides access to the attributes and properties of each robot such as their current position and homing methods.

For many simple pick and place operations, only the first four basic classes need be utilized, i.e. the **Signal**, **Location**, **Profile,** and **Move Classes.** The facilities provided by the more advanced Classes (**RefFrame**, **Controller**, and **Robot**) can be brought into play as an individual becomes more familiar with the system or as applications become more complex.

11.1. Signal Class

The global **Signal Class** provides access to the hardware features of the Guidance Control System that allow GPL programs to interface to other equipment in the work cell using common, simple techniques. These interfaces include “digital input and output (I/O)” signals and “analog I/O” signals. Digital and analog I/O signals permit GPL programs to coordinate the operation of the robot with other equipment using go/no-go semaphores and to interface to various simple sensors.

These hardware interfaces serve as global resources to all threads and are therefore represented by a global class. To access these interfaces, it is not necessary to create an instance of the class; you can refer to the **Signal Class** directly. For example, to read the value of the first digital I/O signal you could execute the following:

```
Dim signal_state As Boolean
signal_state = Signal.DIO(1)
```

All of the properties and methods for the **Signal Class** are described in detail in the Reference Documentation section. The following table briefly summarizes this information.

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.

11.2. Location Class and Objects

The **Location Class** and its instances (“**Location Objects”** or just **“Locations”**) are the fundamental means for specifying robot and part positions and orientations in GPL. Each **Location Object** contains data that defines: a position and orientation; special robot configuration information specific to the

geometry of the robot to be used; and clearance data that define a safe position by which the **Location** can be approached.

11.2.1. There are two basic types of **Location Objects**: **Cartesian Locations** and **Angles Locations**.

11.2.1.1. A **Cartesian Location** stores a robot or part position and orientation in Cartesian coordinates. That is, positions and orientations are represented as X, Y, and Z displacements and rotations in a Cartesian coordinate system.

This is a very intuitive representation and has the advantage of representing positions and orientations in a manner that is independent of a robot's geometry. When a **Cartesian Location** is specified as a destination for a robot motion, the system automatically utilizes its built-in knowledge of the robot's geometry (i.e. its kinematics) to convert this Cartesian position into an equivalent set of robot axes positions. Furthermore, if the kinematic model of the robot includes corrections for manufacturing tolerances (e.g. non-perpendicularity of axes, deviations in link lengths), the **Cartesian Locations** will be automatically corrected for these variances.

In addition to containing a position and orientation, a **Cartesian Location** also has an optional pointer to a reference frame object (**RefFrame**). If **RefFrame** is specified, the Cartesian position and orientation is understood to be relative to the reference frame. When such a **Location** is specified as a destination for a robot motion, GPL automatically combines the **Cartesian Location's** position and orientation with the reference frame to compute the absolute coordinates for the robot's destination.

The use of relative coordinates and reference frames is a very powerful technique since it allows related positions and orientations to be moved as a group. For example, all of the IC chips on a PC board or all of the sample tubes in a tray can be defined relative to a reference frame. If the PC board or the tray is misaligned, the position and orientation of the reference frame can be updated and the absolute values of all of the associated **Locations** will automatically be corrected as well.

For even greater flexibility, a reference frame can itself be defined relative to another reference frame.

11.2.1.2. An **Angles Location** stores a robot position as a set of axes position values. This is the traditional method of representing robot locations and was utilized extensively prior to the introduction of kinematic models. It consists of one axis position value for each degree-of-freedom of the robot.

This method has the benefit of fully and uniquely defining a position of a robot. However, there are several disadvantages of this method relative to the Cartesian representation. For one, if the robot has serial linkages or rotary axes for determining the position of the tool, it is often difficult to intuitively determine how to change the axes positions to effect a desired change in the position or orientation of the robot's tool. Secondly, the use of axes positions makes application programs non-portable between robots with different geometries or even the same geometry but different sizes. Finally, while this representation is sufficient for describing the position and orientation or a robot, it cannot be easily used to define arbitrary positions and orientations of parts and part relationships within the workspace.

The storage of axes positions has been included for completeness and does have its uses. However, it is recommended that **Cartesian Locations** be applied whenever possible.

11.2.1.3. In order to distinguish the type of data stored in a **Location**, a "Type" property is provided. This indicates if the object is an **Angles Location** or a **Cartesian Location.** If the **Location** is a Cartesian type, it can also have an optional pointer to a **RefFrame** Object.

11.2.2. For most common operations that require the position and orientation of a **Location Object**, the data of interest is referred to the "total position" or "position" of the **Location**. The "total position" or "position" is synonymous with the following:

11.2.2.1. For **Cartesian Locations** without a reference frame, the position and orientation stored in the **Location**.

11.2.2.2. For **Cartesian Locations** with a reference frame, the combination of the position and orientation stored in the **Location** with the position and orientation of its reference frame.

11.2.2.3. For **Angles Locations**, the stored axes positions.

11.2.3. For some computations, it is convenient to access the Cartesian position and orientation stored in a **Cartesian Location** while ignoring the optional reference frame. To distinguish this value from the "total position", this data is referred to as the **Location's** "position with respect to the reference frame" (**PosWrtRef**) whether or not a reference frame is specified. The **PosWrtRef** property is not meaningful for **Angles Locations.**

11.2.4. Throughout GPL, Cartesian positions and orientations are internally 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. Homogeneous transforms have several computational advantages and are used to store the "total position" of **Cartesian Locations**, **PosWrtRef** values, reference frames positions and orientations, and during **Cartesian Location** position and orientation computations. However, while this representation has computational benefits, entering the values for the elements of a 4 by 4 homogeneous transformation matrix 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. In general terms, if you are standing up straight and looking at the horizon, the Yaw angle is the amount that you rotate to look left and right along the horizon. The Pitch angle defines if you subsequently tilt your head to look up into the sky or down into the ground. The Roll angle defines a final rotation of your head about its new vertical axis. The X, Y, and Z values are in units of millimeters and the Yaw, Pitch, and Roll are in units of degrees.

11.2.5. Since flexible automation must be able to alter a robot's actions in order to accommodate process variations, one of the most important features of the GPL system is the ability to efficiently and easily mathematically manipulate position and orientation data. In the case of **Angles Locations**, this capability is limited to providing the ability to change individual axes position values. However, for **Cartesian Locations**, a much more powerful mathematics is provided.

As mentioned above, each **Cartesian Location** can have a reference frame or series of reference frames associated with it. These reference frames can not only translate but also rotate the base coordinate system in which the positions are defined. This allows arbitrary 6 degree-of-freedom adjustments to be applied to correct for part and process tolerances and variations.

More generally, GPL includes several methods that can be used to combine the positions and orientations of **Cartesian Locations** and reference frames. Reference frames are a super-set of **Cartesian Locations.** So, in the following paragraphs, the comments concerning **Locations** apply to reference frames as well.

When we combine multiple **Location** positions and orientations, it is easiest to think of **Location Objects** as representing a change in position and orientation with respect to a coordinate system, which in turn defines a new coordinate system. So, if we have a **Location** A, A can be thought of as defining a new coordinate system relative to its base coordinate system. If we combine A with a second **Location** B, the change in position and orientation of B is interpreted with respect to the new coordinate system defined by A. If a third **Location** C is added, the combination of A, B, and C can be computed by interpreting the change in position and orientation of C with respect to the coordinate system generated by combining A and B.

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

In addition to combining **Locations**, we can also eliminate the effects of **Locations** by computing the "inverse" of a **Location**. An inverse negates the change in position and orientation of a **Location**. When we combine these negative results with other computations in the proper order, we can unwind **Location** computations.

The Location Class and its Object include not only basic properties, but also extensive methods for mathematically manipulating the positions and orientations contained with these objects. All of these properties and methods are described in detail in the Reference Documentation section and are briefly summarized in the following table.

Member	Type	Description
location_obj.**Angle**	Property	Sets and gets a single axis position for an **Angles Location.**
location_obj.**Angles**	Method	Changes all of the axes positions values in an **Angles Location.**
location_obj.**Clone**	Method	Returns a copy of the *location_obj.*
location_obj.**Config**	Property	Sets and gets the bit flags that specify special robot specific location attributes.
Location.Distance	Method	Returns the distance between the XYZ positions of two **Cartesian Locations**.
location_obj.**Here**	Method	Modifies the "total position" of the *location_obj* to be equal to the current location of a robot.
location_obj.**Here3**	Method	Defines the "total position" of *location_obj* based upon the XYZ coordinates of three specified locations.
location_obj.**Inverse**	Method	Returns the inverse of the "total position" of the Cartesian *location_obj*.
location_obj.**Kinesol**	Method	Returns a **Cartesian Location** equivalent to an **Angles Location** for a specific kinematic model or vise versa.
location_obj.**Mul**	Method	Returns the result of combining the "total position" of *location_obj* with the "total position" of another **Cartesian Location**.
location_obj.**Normalize**	Method	Corrects the value of the **PosWrtRef** of a **Cartesian Location** for any mathematical inconsistencies in the value.

*location_obj***.Pitch**	Property	Sets and gets the Pitch angle of the **PosWrtRef** of a **Cartesian Location**.
*location_obj***.Pos**	Property	Sets and gets the "total position" of the *location_obj.*
*location_obj***.PosWrtRef**	Property	Sets and gets the **PosWrtRef** of a **Cartesian Location**.
*location_obj***.Roll**	Property	Sets and gets the Roll angle of the **PosWrtRef** of a **Cartesian Location**.
*location_obj***.Type**	Property	Sets and gets the Type specification.
*location_obj***.X**	Property	Sets and gets the X position value of the **PosWrtRef** of a **Cartesian Location**.
*location_obj***.XYZ**	Method	Changes the X, Y, Z, Yaw, Pitch, and Roll values of the **PosWrtRef** of a **Cartesian Location.**
*location_obj***.XYZInc**	Method	Increments the X, Y, and Z values of the **PosWrtRef** of a **Cartesian Location.**
Location.XYZValue	Method	Returns a **Cartesian Location** with a "total position" equal to specified X, Y, Z, Yaw, Pitch, and Roll coordinates.
*location_obj***.Y**	Property	Sets and gets the Y position value of the **PosWrtRef** of a **Cartesian Location**.
*location_obj***.Yaw**	Property	Sets and gets the Yaw angle of the **PosWrtRef** of a **Cartesian Location**.
*location_obj***.Z**	Property	Sets and gets the Z position value of the **PosWrtRef** of a **Cartesian Location**.
*location_obj***.ZClearance**	Property	Sets and gets the distance along the Z-axis that defines the safe approach position to the **Location**.
*location_obj***.ZWorld**	Property	Sets and gets the flag that indicates if the approach distance is measured along the Tool or World Z coordinate axis.

11.3. Profile Class and Objects

In order to move the robot in the standard position control mode, a program must specify the destination for the motion and some trajectory parameters. The trajectory parameters include values that specify how fast the robot is to move and what type of path the robot should traverse. As previously described, **Location Objects** are utilized to specify robot and part positions and orientations. In GPL, the trajectory parameters are captured in **Objects** that are instances of the **Profile Class.**

A **Profile Object** defines a motion's peak speed, peak acceleration and deceleration, s-curve profile parameters, type of path (i.e. straight line or interpolated in joint angles), and a constraint specification used to define if the robot should stop at the end of the motion and when the robot is close enough to the final destination to be considered "in position".

While a program can have a unique **Profile Object** for each motion, it is often desirable to create several, generic **Profile Objects** that can be repeatedly used throughout a project for similar types of motions. For example, you might create one **Profile** for retracting the robot, a second **Profile** for slewing the robot at high speeds between intermediate (via) points, and a third **Profile** for final positioning of parts. The repeated use of generic profiles often simplifies performance tuning an application.

All of the properties and methods for the **Profile Class** are described in detail in the Reference Documentation section. The following table briefly summarizes this information.

Member	Type	Description
profile_obj.**Speed**	Property	Sets and gets the peak motion speed specified as a percentage of the nominal speed.
profile_obj.**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_obj.**Accel**	Property	Sets and gets the peak motion acceleration specified as a percentage of the nominal acceleration.
profile_obj.**Decel**	Property	Sets and gets the peak motion deceleration specified as a percentage of the nominal deceleration.
profile_obj.**AccelRamp**	Property	Sets and gets the duration for ramping up to the peak acceleration, specified in seconds.
profile_obj.**DecelRamp**	Property	Sets and gets the duration for ramping up to the peak deceleration, specified in seconds.
profile_obj.**Straight**	Property	Sets and gets the Boolean indicating if the robot is to follow a straight-line path.
profile_obj.**InRange**	Property	Sets and gets the constraint value 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_obj.**Clone**	Method	Method that returns a copy of the *profile_obj.*

11.4. Move Class

The global **Move Class** provides the methods for commanding the robot to perform a motion. The most fundamental position-controlled motion method is:

Move.Loc (Location1, Profile1).

This executes a single motion segment and moves the robot to the absolute position and orientation specified by *Location1* using the performance parameters specified by *Profile1*. More complex, multi-segment motions can be constructed by executing several **Move** methods in rapid succession. If desired, the system will automatically blend motion segments together into a single “continuous path” that executes several segments in succession before bringing the robot to a stop. This method can significantly improve cycle times of even simple applications. Each motion segment can either move the robot’s tool tip along a Cartesian straight-line path, a circular interpolated path or a joint-interpolated path. Straight-line and circular paths are made possible by the installation of “kinematic modules” that provide GPL with a knowledge of the robot’s geometry.

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.

In addition to position-controlled motions, the system also supports velocity and torque controlled motions.

In order for a robot motion to be executed, the following conditions must be satisfied:

1. High power for the amplifiers and motors must be enabled (see **Controller.PowerEnabled**).
2. The motors must be commutated. This normally happens automatically and is performed during the **PowerEnable** or the homing sequence.
3. In the standard case where the robot is to be position controlled, the robot axes must be homed each time the controller is restarted (see **Robot.HomeAll**). Homing reestablishes the zero position for each axes so that the robot can repeat a previously taught motion.
4. The robot must be attached to the thread (see **Robot.Attached**). Attaching ensures that only a single thread can issue motion commands to a robot.

For general information on the system's motion control capabilities, please see the introductory section on "Motion Control".

All of the methods for the **Move Class** are described in detail in the Reference Documentation section. The following table briefly summarizes this information.

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 destination 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.
Move.StartJogMode	Method	Initiates execution of jog (manual) control mode.
Move.StartTorqueCntrl	Method	Initiates execution of torque control mode for one or more motors.

Move.StartVelocityCntrl	Method	Switches all axes of a robot to velocity control mode in place of position control mode.
Move.StopSpecialModes	Method	Terminates execution of any active special trajectory control modes.
Move.Trigger	Method	Primes the system to automatically assert a digital output signal at a prescribed trigger position during the next motion.
Move.WaitForEOM	Method	Pauses GPL program execution until the current motion is completed.

11.5. RefFrame Class and Objects

The **Objects** of the **RefFrame Class** define robot and part reference frames. As previously described, one or more **Cartesian Locations** can be defined relative to a **RefFrame**. If the position or orientation of the **RefFrame** is subsequently modified, the absolute (or "total) position and orientation of all associated **Cartesian Locations** are automatically adjusted and will move with the reference frame.

For example, a **RefFrame Object,** *tray_ref***,** can be created that defines the position and orientation of a tray of parts. The **Location** of each part on the tray can then be defined with respect to *tray_ref*. If the tray and its parts move in unison, the position and orientation of *tray_ref* can be updated and the total position of all of the part **Locations** will be automatically adjusted and move with the reference frame.

In addition to defining a **Location** with respect to a **RefFrame**, a **RefFrame** can be defined with respect to another **RefFrame**. In the example above, if an array of trays is organized into a two dimensional grid, a second "pallet" **RefFrame,** *pallet_ref,* can be defined to represent the grid of trays. *tray_ref* can then be defined with respect to *pallet_ref*. Each time the *pallet_ref* is advanced to the next tray, the *tray_ref* position will be modified as well as all of the part **Locations** that are defined with respect to *tray_ref*.

To define a **Location** with respect to a reference frame, you simply refer to the reference frame via the **RefFrame** property of a **Cartesian Location**. For example,

```
Dim part1 As New Location   ' part1 defaults to Cartesian Loc
Dim tray_ref As New RefFrame
part1.RefFrame = tray_ref   ' part1 defined wrt tray_ref
```

To simplify the use of reference frames, several different types of **Reframes** exist and more will be added in the future. The common members of all **RefFrame Objects** are summarized in the following table. For detailed information on these members and those of the specific types of reference frames, please consult the GPL Dictionary Pages.

Member	Type	Description
reframe_obj.**Type**	Property	Sets and gets the type of the reference frame.
reframe_obj.**Loc**	Property	Sets and gets the **Location Object** that is an integral part of the reference frame. The use of **Loc** varies for different types of reference frames although **Loc.RefFrame** always defines the next reference frame if *RefFrame_obj* is itself relative to another reference frame.
reframe_obj.**Pos**	Method	Returns the absolute ("total") position and

		orientation for any type of reference frame object.
reframe_obj.**PosWrtRef**	Method	Returns the position for any type of reference frame while ignoring any further reference frames.

11.5.1. Basic Reference Frame

The basic type of **RefFrame** simply stores the position and orientation of the reference frame in the **Loc Location**. The **Loc.Pos** property defines the position and orientation of the reference frame. The GPL project is responsible for defining and updating the **Loc.Pos** value to reflect the current reference frame value.

```
Dim loc1 As New Location          ' loc1 set to Cartesian Loc
Dim ref1 As New RefFrame
loc1.RefFrame = ref1              ' loc1 with respect to ref1
ref1.Loc.XYZ(10,20,30,0,180,20)   ' Set ref1 Pos
```

In order to define a basic reference frame with respect to another reference frame, the **Loc.RefFrame** value must reference the next reference frame.

For a basic reference frame, it is possible to use **Loc.Pos** and **Loc.PosWrtRef** to read the total position and relative position of the reference frame. However, it is generally a better practice to read the **Pos** and **PosWrtRef** of the **RefFrame** instead. *RefFrame_obj*.**Pos** and *RefFrame_obj*.**PosWrtRef** will return the current values for any type of **RefFrame**.

The **RefFrame** members that have special meaning for the basic type of reference frame are briefly described in the table below.

Member	Type	Description
reframe_obj.**Type**	Property	Set to 0 to indicate a basic reference frame.
reframe_obj.**Loc**	Property	**Loc.Pos** is set equal to the position and orientation of the reference frame by a GPL procedure.

11.5.2. Pallet Reference Frame

A pallet reference frame defines a one, two, or three-dimensional rectangular grid of positions than are sequentially indexed. For example, this type of reference frame can be utilized to represent a row of parts being feed, an array of test samples organized into a two dimensional grid or a three dimension pallet of shipping boxes. Once a pallet **RefFrame** has been defined, you can advance to the next position in the pallet by simply invoking the pallet's "**PalletNextPos**" method.

The position of the first item (i.e. index 1,1,1) is defined by the **X,Y**, and **Z** displacements of **Loc**. The directions of the X, Y, and Z axes of **Loc** define the direction for each row, column, and layer of the pallet, respectively.

The distance between each item in a row, column, or layer is defined by the "**PalletPitch**" in each dimension. The maximum number of elements in each row, column, or layer can also be specified. Setting the maximum index to 1 indicates that this corresponding dimension is not incremented.

The order in which GPL indexes along rows, columns, and layers can also be specified. For example, when **PalletNextPos** is executed, the default is to step along the row first, then along columns, and finally

to the next layer. However, you can change the order to any combination. So, you could step by layers first, rows second, and then columns if you so choose.

In addition to using **PalletNextPos** to increment to the next pallet element, the pallet element can be directly specified by the **PalletIndex** property or the **PalletRowColLay** method. When a pallet indexes beyond the final element, it automatically wraps back to the first element.

The **RefFrame** members that have special meaning for the pallet type of reference frame are briefly described in the table below.

Member	Type	Description
refframe_obj.**Type**	Property	Set to 1 to indicate a pallet reference frame.
refframe_obj.**Loc**	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.
*refframe_obj.***PalletIndex**	Property	Sets and gets the index for the next position along the pallet row, column, or layer (1 to n).
*refframe_obj.***PalletMaxIndex**	Property	Sets and gets the maximum position index along the pallet row, column, or layer (1 to n).
*refframe_obj.***PalletNextPos**	Method	Advances to the next pallet position.
*refframe_obj.***PalletOrder**	Property	Sets and gets the parameter that specifies the order in which **PlalletNextPos** indexes along the row, column, and layer indices.
*refframe_obj.***PalletPitch**	Property	Sets and gets the step size for advancing along each row, column, or layer.
*refframe_obj.***PalletRowColLay**	Method	Sets the next pallet position row, column, and layer indices in a single instruction.

11.6. Controller Class

The global **Controller Class** provides a means for GPL programs to access a number of system wide features and facilities of the Guidance Controller System, e.g. High Power control, E-Stop logic, Configuration Database values, etc. These capabilities are represented as properties and methods of the **Controller Class**. Since this class is global, it does not have any properties or fields that have values that are local to a specific routine or program scope. So, the **Controller Class** can be referenced directly without the need for creating instances of **Controller Objects**. For example, to enable high power to the amplifies for non-Category 3 safe systems, the following GPL statement could be used:

```
Controller.PowerEnabled = True
```

In this instruction, "**Controller**" refers to the global **Controller Class** and "**EnablePower**" is a property of this class. Likewise, if we wished to test if high power is currently enabled, the following instructions could be utilized:

```
If (Controller.PowerEnabled) Then
      :
End If
```

Of special interest are the **SystemMessage**, **ShowDialog** and **ShowDialogMCP** methods of this class. These methods allow GPL programs to easily output information to the operator and prompt for simple responses. For the first two methods, the output and input appear on the web page that displays the Operator Control Panel. For the third method, the output and input are performed via the Precise Hardware Manual Control Pendant. In the following example, text is output to the system message log displayed on the Operator Control Panel and then displays a pop-up to prompt for a "Yes" or "No" answer.

```
Dim button As Integer
Controller.SystemMessage("Sample output to Operator Control Panel")
Controller.ShowDialog("Yes,No","Do you like this pop-up?", button)
Controller.SystemMessage("Operator pressed button " & CStr(button))
```

All of the properties and methods for the **Controller Class** are discussed in detail in the Reference Documentation section. The following table briefly summarizes the members of the class.

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

11.7. Robot Class

The global **Robot Class** provides a means for GPL programs to access functions and properties specific to each robot configured in the system. The **Robot** is provided as a global class to simplify its access since many systems have only a single robot and many applications are written to access and control the robot from a single thread. Since this class is global, it does not have any properties or fields that have values that are local to a specific routine or program scope. So, the **Robot Class** can be referenced directly without the need for creating instances of **Robot** Objects.

The **Robot Class** provides properties and methods for reading the current position of a robot, initiating a homing sequence from a program, forcing a rapid deceleration of any in-process motion, retrieving data from the trajectory generator for the robot, 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 give 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. On the other hand, 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. 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.

All of the properties and methods for the **Robot Class** are discussed in detail in the Reference Documentation section. The following table briefly summarizes the members of the class.

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	Gets the Cartesian **Location** that is the final destination for the previously executed motion.
Robot.DestAngles	Property	Gets the Angles **Location** that is the final destination for the previously execution motion.
Robot.Home	Method	Homes the **Attached** robot to establish the reference positions for each axes.
Robot.HomeAll	Method	Homes all robots to establish the reference positions for each of their axes.
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.RapidDecel	Property	Sets the **Boolean** flag that forces any in-process motion for a robot to be rapidly decelerated to a stop.
Robot.RestartBase	Property	Gets the position and orientation offset for the

		base of the robot that was set when the controller was restarted.
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.Selected	Property	Sets and gets the number of the robot that will be accessed for read-only operations by default.
Robot.Source	Property	Returns a Cartesian **Location** whose value is equal to the initial position and orientation of the previously executed motion.
Robot.SourceAngles	Property	Returns an Angles **Location** whose value is equal to the initial axes positions of the previously executed motion.
Robot.Tool	Property	Sets and gets the position and orientation offset for the tool or gripper of the robot.
Robot.TrajState	Property	Gets the **Integer** that indicates the current state of the trajectory generator for a given robot.
Robot.Where	Property	Gets a Cartesian **Location** whose value indicates the current position and orientation of a robot.
Robot.WhereAngles	Property	Gets an Angles **Location** whose value indicates the current position of each axes of a robot.

12. Networking Communications

12. Networking Communications

The following pages explain how to communicate across the Ethernet network using GPL. They provide a summary of the classes involved and examples of how to use them. For additional details on specific methods and properties, see the GPL Dictionary.

GPL includes a number of built-in classes to allow network communications between GPL and other systems using TCP or UDP. They are similar to classes found in Visual Basic, and use concepts from Unix and Linux network stacks. These pages are not intended to be a complete tutorial on network communications, but should provide sufficient information for simple applications.

12.1. Networking Definitions and Classes

The table below summarizes the terms and abbreviations used by the network software and this documentation:

Concept	Description
Client	A TCP or UDP Endpoint. A TCP Client connects to a Server and then issues requests to that Server. Normally a TCP Client does not receive data except in response to a request. A UDP Client sends to and receives from other UDP clients.

Datagram	A unit of data that includes source and destination Endpoint information.
Endpoint	The source or destination for a datagram normally specified as an IP Address and Port.
IP	Internet Protocol - A low-level datagram protocol that is the basis for both TCP and UDP.
IP Address	A 32-bit number that identifies a particular network and computer on that network. Normally written as four decimal numbers, each of which range from 0 to 255, separated by periods. For example: 192.168.0.1
Port	A number from 0 to 65536 that identifies a process or protocol on a networked computer. Some ports are pre-assigned to particular protocols. For example, port 21 is normally used by a FTP server.
Server	A TCP Endpoint that accepts connections from a Client and services requests from a Client. Normally a Server does not initiate I/O but simply responds to requests. A UDP-based server uses the same methods as a client since there is no connection established.
Socket	An **Object** that holds connection information for network I/O. Various methods associate Endpoints with Sockets.
TCP	Transmission Control Protocol - A connection-based protocol that sends reliable Datagrams between Client and Server Endpoints. Messages are guaranteed to be delivered in order.
UDP	User Datagram Protocol – A connection-less protocol that sends Datagrams between two endpoints, without any guarantee of delivery or ordering. UDP is generally faster than TCP, but not as reliable.

GPL supports TCP Server and Client connections, as well as sending or receiving UDP datagrams. The table below summarizes the classes for network I/O.

Networking Class	Description
IPEndPoint	**Objects** of this class describe IP Endpoints.
Socket	**Objects** of this class correspond to local network Endpoints. Most network I/O operations are methods of the **Socket** class.
TcpClient	**Objects** of this class correspond to TCP Clients that can request connections to a TCP Server.
TcpListener	**Objects** of this class correspond to TCP Servers that can accept connection requests from TCP clients.
UdpClient	**Objects** of this class correspond to UDP Endpoints. They can exchange UDP Datagrams with other UDP Endpoints.

The tables below summarize the methods and properties for each of the classes. Each of these properties and methods is described in detail in the GPL Dictionary contained in the Software Reference section of the *Precise Documentation Library*.

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**.

All network-related I/O is performed using **Socket Objects.** **TcpClient**, **TcpListener**, and **UdpClient Objects** contain internal **Socket Objects** that are created by their constructors or methods. These **Socket Objects** are returned by the methods tcpclient_object.**Client**, tcplistener_object.**AcceptSocket,** and udpclient_object.**Client.** It is not useful to create a **Socket** object using **New**.

12.2. TCP Server

A TCP server is a process that listens for connection requests and sets up connections with remote TCP clients. The remote clients send requests to the server on the connection and receive responses. When the connection is no longer needed, it is closed. The steps for setting up a TCP server are:

1. Create an **IPEndPoint Object** for the local endpoint. This **Object** should leave the IP Address blank, allowing any remote node to connect, but set the port to a specific number that the remote client knows.
2. Create a **TcpListener Object** using this **IPEndPoint Object**, and start listening for a connection request by calling the *tcplistener_object*.**Start** method.
3. Optionally poll for a connection request using the *tcplistener_object.***Pending** property.
4. Accept the connection request and obtain a new **Socket Object** by calling the *tcplistener_object.***AcceptSocket** method. If no other connections are to be serviced, stop listening for connections by calling the *tcplistener_object.***Stop** method.
5. Use *socket_object*.**Receive** and *socket_object*.**Send** to perform I/O with the remote client.
6. When finished with the connection, call *socket_object*.**Close** to close it.

12.2.1. TCP Server Example

In this example, a simple TCP server is created to listen for connections on port 1234. A client may connect from anywhere. The server simply echoes back whatever the client sends. You can use a standard Telnet application to communicate with this server.

The **IPEndPoint Object** *ep* for the remote TCP client is set to IP address “”, port 1234, indicating it will connect with any IP address using that port. A **TcpListener Object**, *tl*, is created that listens for connections to that endpoint. The **Pending** method is used to poll for a connection request. When a request arrives, the **AcceptSocket** method returns a new **Socket Object** *ts* that is used for receiving messages and sending replies.

```
Public Sub Telnet
    ' Simple Telnet-like TCP server, listening on port 1234
    ' Echoes back whatever it receives
    Dim ep As New IPEndPoint("", 1234) ' Accept from any IP
    Dim tl As New TcpListener(ep)
    Dim ts As Socket
    Dim recv As String
    Dim send As String
    Dim ii As Integer

    ' Start listening and wait for a connection

    tl.Start()
    While Not tl.Pending()
        Thread.Sleep(5000)
```

```
    End While
    Console.Writeline("Connection request...")
    ts = tl.AcceptSocket()             ' Get the socket
    tl.Stop()                          ' Only service one

    ' Read from client and echo back messages

    While True
        ii = ts.Receive(recv, 1000)
        Console.Writeline("Receive count: " & CStr(ii))
        If ii = 0 Then
            Exit While
        End If
        send = "Received: " & recv
        ts.Send(send)
    End While
    Console.Writeline("Connection closed")
    ts.Close()
End Sub
```

12.3. TCP Client

A TCP client is a process that establishes a connection with a remote TCP server, sends requests to it, and receives replies. When the connection is no longer needed, it is closed. The steps for setting up a TCP client are:

1. Create an **IPEndPoint Object** for the remote server endpoint. This **Object** should specify the IP address of the remote server and the port number on which the server is listening.
2. Create a **TcpClient Object** using this *endpoint_object*. Alternately you can create a **TcpClient Object** omitting the *endpoint_object*, and later call *socket_object*.**Connect** method to establish the connection.
3. Obtain the **Socket Object** for this connection by calling the *tcpclient_object*.**Client** method.
4. Use *socket_object*.**Send** and *socket_object*.**Receive** to perform I/O with the remote client.
5. When finished with the connection, call *socket_object*.**Close** to close it.

12.3.1. TCP Client Example

This example shows how to write a TCP client that connects to a TCP server.

The **IPEndPoint Object** *ep* for the remote TCP server is set to IP address 192.168.0.2, port 1234. A **TcpClient Object**, *tc* is created that connects to that endpoint. The **Socket** *ts* is obtained from *tc* and is used for sending messages and receiving replies.

```
Public Sub Tcp_client
    ' Connect to a remote TCP server at
    ' IP address 92.168.0.2, Port 1234

    Dim ep As New IPEndPoint("192.168.0.2", 1234)
    Dim tc As New TcpClient(ep)
    Dim ts As Socket
    Dim message As String
    Dim reply As String
    Dim ii As Integer

    ts = tc.Client
    message = "Test message" & Chr(GPL_CR) & Chr(GPL_LF)
```

```
    ts.Send(message)
    ts.Receive(reply, 1000)
    Console.Writeline("Reply: " & reply)

    For ii = 1 To 100
        ts.Send(message)
        ts.Receive(reply, 1000)
    Next ii
    Console.Writeline("Test complete")

    ts.Close
End Sub
```

12.4. UDP Server and Client

A UDP Server and UDP client are very similar since there is no explicit connection between the two endpoints. The difference is in how the endpoints are determined. The remote and local endpoints are free to send or receive messages to or from any network address or port. The steps for setting up a UDP server or client are:

1. Create an **IPEndPoint Object** for the local IP address and port. Normally the IP address can be left blank. The port may be left as zero if incoming datagrams to any port should be matched, or non-zero to match only datagrams to a specific port
2. Create a **UdpClient Object** using this local **IPEndPoint Object**.
3. Obtain the **Socket Object** by calling the *udpclient_object*.**Client** method.
4. If you are initiating a request, create another **IPEndPoint Object** that contains the IP address and port of the remote destination. Use this remote **IPEndPoint Object** with the *socket_object*.**SendTo** method to send the datagram.
5. If you are expecting to receive a request, create an **IPEndPoint Object** and pass it **ByRef** when calling the *socket_object*.**ReceiveFrom** method. The IP address and port of the remote endpoint is automatically stored in this **IPEndPoint Object**. You can then use the same **IPEndPoint Object** in a *socket_object*.**SendTo** method call to respond to the endpoint that made the request.

12.4.1. UDP Client Example

In this example, a UDP client is created to read a file from a TFTP server. TFTP is a standard UDP-based file server found on many computers.

The **IPEndPoint Object** *srv_ep* for the remote UDP client is set to IP address “192.168.0.2”, and the TFTP port 69. A **UdpClient Object**, *uc*, is created and the **Socket Object** associated with *uc* is stored in *us*. The remainder of the I/O is performed with this **Socket Object**.

A TFTP “file open” message is built in string *out* and sent to the remote UDP endpoint contained in *srv_ep* using the **SendTo** method. Using the **ReceiveFrom** method, the reply is stored into the string *inp*, and the responding remote endpoint is saved in *rem_ep*. The rest of the messages are sent to *rem_ep*, and additional replies are checked to verify that they are also from r*em_ep*.

```
Public Sub TftpClient
    ' Access a TFTP server using UDP, open a file,
    ' and display it on the console.
    Dim file As String = "testfile.txt"
    Dim srv_ep As New IPEndPoint("192.168.0.2", 69)
    Dim rem_ep, ep As IPEndPoint
```

```
    Dim out, inp As String
    Dim uc As New UdpClient()
    Dim us As Socket
    Dim count, op, block As Integer

    us = uc.Client

    ' Build "open for read" command
    out = Chr(0) & Chr(1) & file & Chr(0) & "octet" & Chr(0)
    us.SendTo(out, 0, srv_ep)

    count = us.ReceiveFrom(inp, 1500, rem_ep)
    Console.Writeline("Remote ip: " & rem_ep.IPAddress & _
                      ", port: " & CStr(rem_ep.Port))

    op = Asc(inp.Substring(0,1))*256 + Asc(inp.Substring(1,1))
    block = Asc(inp.Substring(2,1))*256 + Asc(inp.Substring(3,1))
    Console.Writeline("Block: " & CStr(Block))
    If (count>4) Then
        Console.Writeline(inp.Substring(4))
    End If

    While True
        out = Chr(0) & Chr(4) & Chr(block/256) & Chr(block)
        us.SendTo(out, 0, rem_ep)

        If (count<512) Then         ' End if less than 512 bytes
            Exit While
        End If

        count = us.ReceiveFrom(inp, 1500, ep)
        If (ep.IPAddress<>rem_ep.IPAddress) Or _
            (ep.Port<>rem_ep.Port) Then
            Console.Writeline("Address mismatch")
            Exit While
        End If
        block = Asc(inp.Substring(2,1))*256 + _
                Asc(inp.Substring(3,1))
        Console.Writeline("Block: " & CStr(Block))
        If (count>4) Then
            Console.Writeline(inp.Substring(4))
        End If
    End While

    Console.Writeline("Transfer complete")
    us.Close

End Sub
```

13. MODBUS/TCP Communications

13. MODBUS/TCP Communications

The following pages explain how to communicate across the Ethernet network using the MODBUS/TCP protocol. This is an "open" de facto standard protocol that is widely employed in the industrial manufacturing environment to communicate with intelligent devices such as sensors and instruments. 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. In addition to factory applications, MODBUS/TCP is being utilized in building, infrastructure, transportation and energy applications.

MODBUS/TCP is layered on top of the Ethernet TCP protocol. The GPL **Modbus Class** is provided as a convenience to allow a GPL procedure to easily communicate with MODBUS/TCP devices without the

need to implement this protocol. This section provides a summary of the **Modbus Class** and examples of how to use it. For additional details on specific methods, see the *GPL Dictionary*.

For more information on the TCP protocol, see the *Network Communications* section. For information about the MODBUS/TCP protocol and standards, see the MODBUS-IDA website at http://www.modbus.org.

GPL operates as a Master and communicates to devices that are configured as MODBUS/TCP slaves. In this mode, GPL supports the following MODBUS/TCP functions:

Function Code	Function Name	Description
1	Read coils	Read one or more digital outputs.
2	Read discrete inputs	Read one or more digital inputs.
3	Read holding registers	Read one or more holding registers.
4	Read input registers	Read one or more input registers.
5	Write single coil	Write a single digital output.
6	Write single register	Write a single holding register.
15	Write multiple coils	Write multiple digital outputs.
16	Write multiple registers	Write multiple holding registers.
43, MEI type 13	Read Device Identification	Read string values identifying the device.

In addition, a Guidance controller can be configured to operate as a MODBUS/TCP slave and accept commands from an 3rd party MODBUS/TCP master. Please see the *Communications* section of the *Introduction to the Software* chapter of the *Precise Documentation Library* for more information on this mode of operation.

13.1. Modbus Class

The **Modbus Class** in GPL supports master access to MODBUS/TCP slave devices connected to the local Ethernet.

The tables below summarize the methods and properties of the class.

Modbus Class Member	Type	Description
New Modbus	Constructor Method	Creates an object for a MODBUS connection and specifies the IP address.
modbus_object.**Close**	Method	Closes any connections associated with this object.
modbus_object.**ReadCoils**	Method	Reads one or more outputs.
modbus_object.**ReadDeviceId**	Method	Reads the device ID strings.
modbus_object.**ReadDiscreteInputs**	Method	Reads one or more inputs.
modbus_object.**ReadHoldingRegisters**	Method	Reads one or more holding registers.
modbus_object.**ReadInputRegisters**	Method	Reads one or more input registers.
modbus_object.**Timeout**	Get/Set Property	Gets or sets the timeout, in milliseconds, that this connection will wait for a reply before throwing an

		exception.
modbus_object.**WriteMultipleCoils**	Method	Writes multiple outputs.
modbus_object.**WriteMultipleRegisters**	Method	Writes multiple holding registers.
modbus_object.**WriteSingleCoil**	Method	Writes a single output.
modbus_object.**WriteSingleRegister**	Method	Writes a single holding register.

13.2. Modbus Master Connection

When GPL operates as a MODBUS master, it sets up a TCP client connection with a remote MODBUS/TCP slave. The slave acts as a TCP server. When the connection is no longer needed, it may be closed. The steps for establishing this type of connection are as follows:

1. Create an **IPEndPoint** object for the remote MODBUS/TCP slave. This object normally specifies the IP address of the slave and omits the port number, in which case the standard MODBUS/TCP port is used.
2. Create a **Modbus** object using this *endpoint_object*. Creating a **Modbus** object does not establish a connection, but simply saves the endpoint information for later.
3. Use the *modbus_object*.**Timeout** property to set an appropriate timeout value for the connection. By default the timeout is infinite.
4. Use the various **Modbus** class methods to read or write data. The first time you issue a read or write, GPL attempts to connect with the MODBUS slave. If the slave does not respond, an exception is thrown.
5. When finished with the MODBUS slave, call *modbus_object*.**Close** to close it. Do this at the end of a session, not after each read or write request.

13.3. Modbus Master Examples

In both of these examples, the **IPEndPoint** object *ep* for the MODBUS slave is set to IP address 192.168.0.150. A **Modbus** object, *mb* is created that refers to that endpoint. The *mb* object is used for communicating with the slave.

This first example shows a procedure that reads from a MODBUS slave.

```
Public Sub Modbus_Read_Example
    Dim ep As New IPEndPoint("192.168.0.150")
    Dim mb As New Modbus(ep)
    Dim ii As Integer
    Dim bool() As Boolean
    Dim input() As Integer

    mb.ReadCoils(1, 16, bool)
    For ii = 1 To 16
        Console.Write("Coil " & CStr(ii) & ": ")
        Console.Writeline(bool(ii-1))
    Next ii

    mb.ReadDiscreteInputs(1, 16, bool)
    For ii = 1 To 16
        Console.Write("Input " & CStr(ii) & ": ")
        Console.Writeline(bool(ii-1))
    Next ii

    mb.ReadHoldingRegisters(1, 2, input)
```

```
    For ii = 1 To 2
        Console.Write("HReg " & CStr(ii) & ": ")
        Console.Writeline(Hex(input(ii-1)))
    Next ii

    mb.ReadInputRegisters(1, 2, input)
    For ii = 1 To 2
        Console.Write("IReg " & CStr(ii) & ": ")
        Console.Writeline(Hex(input(ii-1)))
    Next ii

    mb.Close()

End Sub
```

The next example shows a procedure that writes to a MODBUS slave.

```
Public Sub Modbus_Write_Example
    Dim ep As New IPEndPoint("192.168.0.150")
    Dim mb As New Modbus(ep)
    Dim ii As Integer
    Dim output() As Integer
    Dim bool() As Boolean

    For ii = 1 To 16
        mb.WriteSingleCoil(ii, ii And 1)
    Next ii

    mb.WriteSingleRegister(1, 600)

    ReDim bool(15)
    For ii = 0 To 15
        bool(ii) = ii And 2
    Next ii
    mb.WriteMultipleCoils(1, bool)

    ReDim output(15)
    For ii = 0 To 15
        output(ii) = ii*ii
    Next ii

    mb.WriteMultipleRegisters(1, output)

End Sub
```

14. File I/O, Serial I/O and Streams

14. File I/O, Serial I/O and Streams

The following pages describe how to read and write data from or to serial ports and files using GPL streams. These pages provide a summary of the classes and methods that may be used, as well as some simple examples. For additional details on individual methods, see the GPL Dictionary.

The table below summarizes many of the concepts related to file and serial I/O operations that are mentioned in this section.

Concept	Description
ASCII	American Standard Code for Information Interchange. A code that represents the

	English alphabet, numbers, symbols, and control characters as 7-bit binary numbers. Used by GPL to represent text strings.
Buffer	An internal data area that groups bytes into larger blocks so that they can be read or written more efficiently.
Byte	An 8-bit data item that can hold a number from 0 to 255 or an ASCII character. Streams are composed of bytes.
CR	Carriage Return. The ASCII control character with decimal value 13. Often used as a line terminator.
Directory	A named grouping of files. Also known as a "folder". Directory names have the same properties as normal file names. Directories may be contained inside other directories.
File	A named collection of bytes that may be stored in permanent flash memory (on device /flash) or in temporary system memory (on device /ROMDISK).
File name	The name of a file. File names may be from 1 to 43 characters and contain any printable ASCII character other than "/" or a leading ".". Upper and lower case letters are considered to be different. It is recommended, but not required, that only valid GPL symbol names are used.
Flush	For efficiency, write operations often just add data to an internal buffer and do not access the associated file or serial port. This allows small strings to be accumulated. The system then automatically writes entire buffers to the output device when the buffer is full. The downside of this process is that if the controller is turned off, the contents of the internal buffers are lost. "Flushing" buffers forces their contents to be written to the file or serial port. Closing a stream automatically flushes any associated buffer.
LF	Line Feed. The ASCII control character with decimal value 10. Often used as a line terminator.
Line terminator	A sequence of 1 or 2 ASCII characters that marks the end of a line. Normally LF, CR, or CR-LF.
Path	The file name, preceded by a list of folders that determine the location of the file. For example: A GPL program file may be found in "/flash/projects/My_project/Main.gpl".
Serial port	An I/O device that transmits and receives byte data using the RS-232 standard. The first serial port is named "/dev/com1". Depending on your controller model, you may have additional serial ports, named "/dev/com2", "/dev/com3", etc. Remote serial ports are named "/dev/comrxy" where "x" is the number of the remote device and "y" is the serial port on the remote device.

The **StreamWriter** and **StreamReader** classes treat data from serial ports or files as a continuous stream of 8-bit bytes. These bytes may be ASCII characters or they may be arbitrary binary data. Many of the methods transfer data to and from GPL string variables. Each byte of a string may be thought of as either an 8-bit binary value or an ASCII character. GPL includes methods and functions to convert between integer data and ASCII characters in strings, for example the **Chr** and **Asc** functions.

Some methods interpret the data stream as a series of lines, terminated by a special "line-terminator" character sequence. The **NewLine** property allows some flexibility in determining the line-terminator used when writing lines.

The **File** class contains methods for managing entire files or directories, such as creating directories or deleting files. All the **File** methods are shared, so there are no **File** objects.

14.1. Classes and Methods

GPL provides the **File** built-in class for managing files and directories. The table below summarizes the various methods available.

File Class Member	Type	Description
File.CreateDirectory	Shared Method	Creates a directory including any undefined directories in its path.
File.DeleteDirectory	Shared Method	Deletes a single directory, if it is empty.
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.

GPL provides two built-in classes for accessing streams: **StreamReader** and **StreamWriter**.

StreamReader Member	Type	Description
New StreamReader	Constructor Method	Opens a file or serial port device for reading.
*streamreader_obj***.Close**	Method	Closes a file or device.
*streamreader_obj***.Peek**	Method	Reads a single byte but does not remove it from the input stream.
*streamreader_obj***.Read**	Method	Reads a single byte and removes it from the input stream.
*streamreader_obj***.ReadLine**	Method	Reads a line of bytes 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	If True, automatically flushes output after every write.
*streamwriter_obj***.Close**	Method	Closes a file or device.
*streamwriter_obj***.Flush**	Method	Forces any pending output to occur immediately.
*streamwriter_obj***.NewLine**	Property	Defines the line terminator characters that are appended to output by **WriteLine**.
*streamwriter_obj***.Write**	Method	Writes a string or number to the output stream with no line terminator.
*streamwriter_obj***.WriteLine**	Method	Writes a string or number to the output stream followed by a line terminator.

The same methods are used for accessing both files and serial ports. The major differences between the two are:

1. Serial ports are normally used for communications, but files are used to save and retrieve data.

2. Data read from files is normally available immediately, but you may need to wait to receive data from a serial port.
3. Files have an "end of file", but serial port data can continue indefinitely.
4. Data written to files is normally buffered for efficiency, but serial port communications are often time-critical so the output is not buffered.

GPL also provides a built-in class for performing output to the serial console or to the GDE console window.

Console Class 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.

14.2. File I/O

Files are used to save and retrieve data to and from a disk, flash or similar device. To locate a file, you must provide a "path" to that file. The first item in the path is the device, followed by a list of folders, and ending with the file name. The device name, folder names and file name are separated by "/" characters. For example:

/ROMDISK/my_folder/my_file.dat

File names often contain an embedded "." followed by a character sequence that indicates the file type. This file type is treated as simply part of the file name and is ignored by the file system. However, the file type is used by certain system components. For example, the GPL compiler assumes that source files always have the type ".gpl", so a file name might be:

/flash/projects/Myproject/Main.gpl

Files may be either temporary or permanent. Temporary files are written to a temporary memory-based disk with device name "/ROMDISK". These files consume blocks of the CPU's main memory and are lost when the controller is restarted. Temporary files may be read or written very quickly. Permanent files are written to a disk which is part of the non-volatile flash memory, with device name "/flash". This disk is very slow to write, but may be read quickly. All file paths must begin with either "/ROMDISK" or "/flash".

14.2.1. Steps for Writing a File

1. Open the file by creating a **StreamWriter** object. The path to the file to be written is a required argument to the **StreamWriter New** method. If you want to append to an existing file, set the *append* input parameter to True. Otherwise a new file is created, overwriting any existing file that matches the path.
2. Decide if you want your data to be buffered during output. If not, change the **AutoFlush** property to **True**, from its default value of **False**. Setting **AutoFlush** to True will make the output much slower, especially for the /flash device.
3. If you are going to organize your output data into lines, decide if the default line terminator (CR-LF) is appropriate. If not, use the **NewLine** property to change it.
4. Use the **Write** or **WriteLine** methods to write the data.
5. Use the **Close** method to force any pending output to be written and to update all internal file data.

14.2.2. Steps for Reading a File

1. Open the file by creating a **StreamReader** object. The path to the file to be read is a required argument to the **StreamReader New** method.
2. Read the data by using either the **Read** or **ReadLine** methods.
3. Use the **Peek** method to check for the end of the file, or enclose the read operation in a **Try-Catch** block to capture read errors.
4. Use the **Close** method to release any system resources held by the object.

14.2.3. File I/O Example

In this example, a temporary file is created using the **StreamWriter** object *"o"* and written with lines that contain the string values "Line 1" through "Line 10". The file is closed and then opened for read using the **StreamReader** object *"i"*. If the **Peek** method does not indicate end-of-file, a line is read from the input file and written to the console. Finally the input file is closed.

```
Public Sub file_write_read
    ' Write a file, read it back, and display it on the console
    Dim o As New StreamWriter("/ROMDISK/filetest")
    Dim i As StreamReader
    Dim line As String
    Dim ii As Integer

    For ii = 1 To 10
        o.WriteLine("Line " & CStr(ii))
    Next ii
    o.Close()

    i = New StreamReader("/ROMDISK/filetest")
    While i.Peek() >=0          ' Check if end-of-file
        line = i.Readline()
        Console.WriteLine(line)
    End While
    i.Close()
End Sub
```

14.3. Serial I/O

Serial ports are normally used to communicate with a host computer or an intelligent sensor. The GPL Controller's first serial port is named "/dev/com1". If your controller contains additional serial ports, they are named "/dev/com2", "/dev/com3", etc. If your system is connected to a Remote IO (RIO) board that provides additional remote serial ports, they are named "/dev/comrxy" where "x" is the number of the RIO board and "y" is the number of the RIO's serial port.

The first serial port is also used by the GPL serial console interface, so you cannot use the serial console if you are using "/dev/com1". When you open device "/dev/com1", the console interface is immediately disabled. You can disable or re-enable the serial console, by changing Parameter Database entry "Serial console enable" (DataID 125). When the serial port is being utilized for program input and output, you can still access the system console via the Telnet interface. Note that system crash messages and certain fatal error messages may be output to /dev/com1 even when it is being used by a GPL procedure. Your remote system must be able to handle these unexpected messages.

Serial ports send and receive streams of byte data in a format and rate determined by their configuration. See instructions elsewhere for setting up the baud rate, character size, stop bits, parity, and hardware flow control settings. Unlike files, a single serial port can be opened for both input and output simultaneously. There is no way for a serial device to detect that a communications link has been closed.

Normally the remote device sends a special byte sequence or message to indicate the end of communications.

14.3.1. Steps for Communicating Using a Serial Port

1. If you are planning to use the first serial port permanently for communications, set the system parameter "Serial console enable" (DataID 125) to 0.
2. Open the port for output by creating a **StreamWriter** object. The device name is a required argument to the **StreamWriter New** method. For serial port 1, the device is "/dev/com1".
3. Open the port for input by creating a **StreamReader** object. The device name is a required argument to the **StreamReader New** method. Use the same device as specified in the previous step.
4. Decide if you want your data to be buffered during output. Generally serial communications is not buffered. If you want it buffered, change the **AutoFlush** property to **False**, from its default value of **True**. If you use buffered output you probably need to use the **Flush** method to make sure your output is transmitted when you expect.
5. If you are going to organize your output data into lines, decide if the default line terminator (CR-LF) is appropriate. If not, use the **NewLine** property to change it.
6. Use the **Write** or **WriteLine** methods on your **StreamWriter** object to write data.
7. If you do not want your input procedure to block waiting for data to be received, use the **Peek** method to check if data is present before using **Read**.
8. Use the **Read** or **ReadLine** methods on your **StreamReader** object to read data.
9. Use the **Close** method to release the serial ports for other use and free system resources.

14.3.2. Serial I/O Example

In this example, serial port 1 is used to communicate with an operator terminal connected to the port. The program prompts the operator to type a character and waits until they do. Then it outputs a message describing the character to the serial port. The device "/dev/com1" is opened for both output and input by creating both **StreamWriter** and **StreamReader** objects, *"o"* and *"i"*, respectively. The output line terminator is set to CR by using the **NewLine** property. The procedure polls the input every 500 milliseconds for input. If no input is received, a series of dots is output. When an input character is received, it is converted to a readable string and a message is written back to the serial port. When an ASCII ESC character (decimal value 27) is received, the procedure closes the streams and exits.

```
Public Sub com1
    ' Open com1, echo info about any input.
    Dim o As New StreamWriter("/dev/com1")
    Dim i As New StreamReader("/dev/com1")
    Dim c As Integer
    Dim ss As String

    o.NewLine = Chr(GPL_CR)    ' Set CR as the line terminator
    o.WriteLine("Type characters, hit ESC to quit.")

    Do
        o.Write("Waiting for input ")
        While i.Peek() < 0
            Thread.Sleep(500)
            o.Write(".")
        End While
        o.WriteLine("")

        c = i.Read()
        If c >= &H20 Then
```

```
            ss = Chr(c)
        Else
            ss = "^" & Chr(c+&H40)
        End If

        o.WriteLine("You typed " & """ & ss _
                    & """ = " & CStr(c))
    Loop While c <> 27              ' Exit if ESC typed

    i.Close()
    o.Close()
End Sub
```

14.4. Console Output

As a convenience during program development and testing, serial output may be performed 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.

For more information on how to use and configure the serial ports, see the previous Serial I/O section.

The console output methods are overloaded and allow either a numeric value or string to be output. For output that combines both string and numeric values, use the **CStr** function.

14.4.1. Example

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

15. Vision Guidance

15. Vision Guidance

The following pages describe how to access the PreciseVision machine vision system from a GPL procedure and use the vision data in a motion application.

PreciseVision is a software application that runs on a PC. That PC in-turn is connected to cameras that acquire images to be processed. The vision processing performed by PreciseVision is specified in terms of "vision tools" and "vision processes". Details about how to setup and program PreciseVision may be found in *The PreciseVision Machine Vision System, Introduction and Reference Manual*.

In order for GPL to send commands to PreciseVision, GPL must know the IP address for the PC that is executing PreciseVision. This value is specified in the Configuration and Parameter Database in the "Vision server IP address" (DataID 424).

The table below summarizes some of the concepts related to vision operations that are mentioned in this section.

Concept	Description
Client	A process that makes requests to a server and handles the responses. Normally a client initiates all communications and does not receive data except in response to a request.
Server	A process that responds to requests and sends responses. It normally does not initiate I/O.
Vision Tool	A single operation executed by PreciseVision. A typical tool might find an object (e.g. a Finder Tool), measure a dimension (e.g. an Edge Locator) or locate a key feature (e.g. a Line Fitter).
Vision Process	A series of vision tools performed on an image by PreciseVision. The tools in the process normally produce Vision Results that are used by a GPL procedure.
Vision Result	The output of a Vision Tool that is executed by PreciseVision. A set of Results may contain pass/fail information, location data, or general numeric data. Some tools only generate a single set of results (e.g. a Line Fitter) while others generate multiple sets of results (e.g. a Finder). A single set of results is normally stored in a **VisResult** object in GPL.

When active, PreciseVision acts as a server that fields requests from client GPL procedures. These client GPL procedures execute on a Precise Controller and communicate with the PC via Ethernet. By designing GPL procedures as clients of PreciseVision, GPL procedures have complete control over when pictures are taken and processed.

To take a picture and analyze its results, a GPL procedure issues a command to PreciseVision to execute a "vision process". Normally, a vision process consists of a tool that takes a picture (i.e. an Acquisition Tool) followed by additional tools to process and analyze the picture. In the simplest case, a vision process consists of a single tool that operates on an existing picture. At times, a process can be quite complex and might contain dozens of tools that inspect multiple features of parts to verify that the parts are correct. From GPL's point of view, a vision process is a single, indivisible operation. That is, after a GPL procedure 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 its results.

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. Once the vision process has completed execution, a GPL procedure can utilize the tool names to retrieve the results from any tool. These results typically indicate the locations of parts that are to be manipulated and the type of each part. In addition, vision can be used to check for key dimensions or other features of the parts and can return information to GPL about the quality of a part. As mentioned above, some tools return only a single set of results while others can return multiple sets of information.

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 second vision process is executed using another communication object, the results of first vision process are preserved.

The following pages provide a summary of the built-in GPL classes and methods that act as an interface to the PreciseVision system, as well as some simple examples.

15.1. Classes and Methods

The network communication interface between the Precise controller and the PreciseVision system is implemented by a **Vision** class and its associated objects. Its methods and properties allow a GPL procedure to establish a connection with PreciseVision, run a vision process, and obtain the results from that process.

The **VisResult** class defines objects that each store a single set of results from a vision tool. These objects may contain pass/fail information, location data, or general numeric data, depending on the vision tool.

The tables below summarize the available members for the vision classes. For additional details on individual vision methods and properties, please see the GPL Dictionary.

Vision Class Member	Type	Description
New Vision	Constructor Method	Creates an empty **Vision** object. Does not communicate with PreciseVision.
Vision.Disconnect	Shared Method	Closes any open connection to PreciseVision.
*vision_object***.ErrorCode**	Property	Returns the numeric error code for the last executed vision process. A value of 0 indicates success; a negative value indicates an error.
*vision_object***.Process**	Method	Requests that PreciseVision execute a vision process and waits for it to complete. Connects to PreciseVision if there is currently no connection.
*vision_object***.Result**	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.
*vision_object***.ResultCount**	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.
*vision_object***.Status**	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.
Vision.ToolProperty	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 *vision_object***.Result** method.
*visresult_object.***ErrorCode**	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 a non-critical error occurred.

*visresult_object.***Info**	Property	Returns the nth numeric information field contained in this set of results.
*visresult_object.***InfoCount**	Property	Returns number of numeric information items in this set of results.
*visresult_object.***InspectActual**	Property	Returns the value of the tool property that was tested in the vision inspection process.
*visresult_object.***InspectPassed**	Property	Returns **True** if a property of the vision results satisfied the tool's vision inspection criteria.
*visresult_object.***Loc**	Property	Returns the position and orientation from a set of results as a Cartesian **Location** object.
*visresult_object.***Type**	Property	Returns the type of this set of results. Currently always zero.

15.2. Vision Interface

Vision objects are used to communicate with the PreciseVision system. The communications occur across a TCP/IP Ethernet link between the Precise controller and the PC running PreciseVision. Simply creating a **Vision** object does not cause any communication to occur.

The **Vision** methods **Process**, **Result**, and **ResultCount** all send a request to PreciseVision and wait for a reply. There is no method to explicitly connect to PreciseVision. A connection is automatically established when one of these methods is called.

When making a connection, the Precise controller attempts to communicate with TCP port 1410 at the IP address specified by the parameter database entry "Vision server IP address" (DataID 424). If a connection cannot be made, an exception is thrown. Once a request is sent, PreciseVision must respond within 30 seconds or an exception is generated.

The steps for preparing PreciseVision to service requests and to execute vision processes for a Precise controller are as follows:

1. Physically connect your Precise controller with the PC running PreciseVision. Make sure the Ethernet IP addresses are setup properly and the PC can communicate with the GPL controller.
2. Using PreciseVision on the PC, create a vision process that uses vision tools to acquire an image and perform the desired vision operations.
3. Make sure that PreciseVision is active and listening for requests.

To develop a vision guidance application that will execute on a Precise controller and communicate with PreciseVision, write and execute a GPL procedure that does the following:

1. Creates a **Vision** object to serve as the interface to PreciseVision.
2. Executes a **Vision Process** method to initiate a vision process in PreciseVision. The process name specified in this method must match a vision process defined within PreciseVision.
3. Uses the **ResultCount** method to determine how many sets of results were generated by each vision tool of interest.
4. Accesses the **Result** method for each vision tool and results set of interest to obtain a **VisResult** object that stores a set of vision output information.
5. Uses the **VisResult** class properties and methods to obtain specific vision data that can be applied in your GPL procedure.
6. Executes the **Vision Disconnect** method when done with all vision processing to close the communication connection.

15.3. Vision Procedure Example

In this example, PreciseVision is used to determine the location of a part that is then acquired by the robot. The output of the vision process is used to create a reference frame, and the robot is moved to a point relative to that reference frame.

In particular, the robot moves to the location *safe* to avoid blocking the camera's field-of-view. The **Vision** object *vis* is then used to connect with PreciseVision and execute the vision process "Main". This vision process takes a picture and executes vision tools to locate the part and perform any desired visual verifications. At the end of the vision process, all that GPL requires is the results of the tool “part1”, which contains the location of the part. The GPL procedure then checks the **ResultCount** property to ensure that at least one set of results is available. The **Result** method returns the first set of results from "part1" in the **VisResult** object *vResult*. The returned vision location is used to create the object *vsRefFrame,* which is the reference frame for location *vsRelPoint*. The robot moves to *vsRelPoint* and finally moves back to its *safe* location.

```
Public Sub MAIN
    Dim vis As New Vision
    Dim vResult As New VisResult

    Robot.Attached = 1
    Move.Loc(safe, vsProfile)

    vis.Process("Main") ' Run vision process "Main"
    If vis.ResultCount("part1") = 0 Then
        Console.Writeline("Vision object not found")
        Goto done
    End If
    vResult = vis.Result("part1", 1) ' Get results

    ' Create a reference frame object and set it
    ' equal to the returned vision location
    Dim vsRefFrame As New RefFrame
    vsRefFrame.Loc.PosWrtRef = vResult.Loc

    ' Pickup point is relative to new frame
    vsRelPoint.RefFrame = vsRefFrame

    Move.Approach(vsRelPoint, vsProfile)
    Move.Loc(vsRelPoint, vsProfile)
    Move.Approach(vsRelPoint, vsProfile)

    ' Move back to safe location
    Move.Loc(safe, vsProfile)

done:
End Sub
```

16. Managing and Executing GPL Projects

16.1. Projects and Files

In GPL, rather than executing a "program", a "Project" is the basic executable entity. Console commands are provided for loading, compiling, and executing a Project. A Project consists of two or more text files that are stored within a single disk folder (directory). Each file is a standard human-readable ASCII file. The folder name and the Project name are synonymous.

The file "Project.gpr" must always be present in each project folder and is referred to as the "Project File". This file contains information on the other files within the Project including which program is invoked when the Project begins execution.

Each GPL source file has a "gpl" extension. These files each can contain one or more modules, which in-turn can contain multiple variable declarations and procedures.

A Project can also contain one, several or no files with a "gpo" extension. This type of file contains a global module that is used to defined global **Location** and **Profile** objects. This file is convenient for storing taught robot locations and general motion Profiles that are accessible by all procedures within the Project.

Loading a Project into memory or copying a Project from memory or between disk units is equivalent to copying a file folder and all of its contents. Multiple Projects can be present in memory although only one Project can be executed at any given time.

16.2. Modules

Only modules can be found at the outer-most level of a file. [In the future, class declarations will also be allowed]. These modules contain variable declarations such as **Public**, **Private**, and **Dim** statements, or procedure declarations such as **Sub** or **Function** statements. A procedure or module-level variable can be accessed by fully specifying its name using the syntax:

module_name.variable_name
-or-
module_name.procedure_name

Within a single module, all procedures and module-level variables can be freely accessed. However, only **Public** procedures and variables in other modules can be accessed. If **Public** variables or procedures with the same name are found in two different modules, they can only be accessed by using the fully-specified name, to disambiguate the multiple definitions.

16.3. Executing a Project

Before a Project can be executed, it must be loaded into memory and compiled. The steps are as follows:

1. Load the Project and associated files into memory.
2. Issue a compile request for the Project.
3. Issue a start request for the Project.

The Project begins execution at the "start" procedure specified in the Project File (Project.gpr). Note that the start procedure must be declared **Public**.

17. Thread Control

When a GPL Project begins execution, its main procedure starts running in a system "thread". Each thread has its own execution stack and runs independently of all other threads.

The GPL system supports the simultaneous execution of up to 32 GPL user program threads. These threads allow simultaneous execution of multiple projects. Even more importantly, a main thread can initiate and control the execution of additional procedures in their own threads. This is very convenient for the execution of communications servers, digital I/O scanners, and cell control tasks that are best executed asynchronously from the main execution thread.

When multiple GPL threads are used, it is often necessary to synchronize them. For example, a server thread may wait for a client thread to post a command, and the client may wait for the server to respond. Two or more threads can efficiently be synchronized by using the **SendEvent** and **WaitEvent** methods. Any GPL thread can send a synchronization message called an *event* to any other GPL thread. Up to 16 independent events per thread can be sent to permit the receiving thread to discriminate between types of events. The events are numbered 1 through 16. The target thread uses **WaitEvent** to efficiently wait for one or more of these events to be received. While a thread is waiting for an event, it uses almost no CPU time.

To control the starting, stopping, and monitoring of independent threads, GPL includes a **Thread Class** that includes the required methods and properties. In the following table, the members of this **Class** are briefly described. Completion information on these class members are provided in the GPL Dictionary pages.

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.

18. Misc Unsupported Features

GPL does not support conditional compilation and its associated directives, e.g. #If.