GlobalVariable.dll Version 2.2 - Documentation

Updated: 10/28/2004

Contents

What's New in Version 2.2?

< Top > (Alt ← = Back)

Below is a summary of the features that are new in GlobalVariable.dll version 2.2. See the ReadMe.txt file included in the Global Variable download package for more information.

Added a new type of global variables, Boolean global variables. Added functions to demonstrate storage and retrieval of numbered and named global variables of Boolean type.

Added functions to "reset" or "reinitialize" named global variables. GV_ResetAllNmd<Type> functions demonstrate how all named global variables of a given type name can have their names reset to NULL and their values reset to initial values (0 for numeric global variables, NULL for string global variables, and FALSE for Boolean global variables).

Added GV_Get<Type>NameByNum functions to demonstrate retrieval of the name of a named variable based on the named variable's element location number.

Added GV_GetNamed<Type>ByNum functions to demonstrate retrieval of the value stored in a named variable based on the named variable's element location number.

Modified return type of all functions that return strings so that they return BSTR's rather than LPSTR's to demonstrate DLL code that can be used with applications that use BSTR's rather than LPSTR's.

Expanded the interprocess data segment for string global variables from 3000 locations to 10,000 locations to better demonstrate storage of numbered string global variables.

Added internal function fnGenRunTimeError to demonstrate the ability to generate a run-time error in TradeStation under appropriate circumstances using the functionality included in the EasyLanguage Extension SDK (tskit.dll).

Added Version.rc resource file to the DLL project to allow for easy retrieval of GlobalVariable.dll version information (for example, using Windows Explorer).

Introduction

< Top > (Alt ← = Back)

A dynamic-link library (DLL) can be used to extend EasyLanguage®. DLL's can be written in any of a variety of programming languages, including C/C++, Delphi, and PowerBASIC®. User developed TradeStation(R)-compatible DLL's, while not part of the TradeStation platform itself, are used to provide user-developed function libraries that may be called from EasyLanguage analysis techniques.

Functions exported from user-developed DLL's can be called by EasyLanguage analysis techniques and by other applications. DLL functions can perform actions that cannot be done easily or at all in EasyLanguage. Additionally, they might be used to speed up processing.

The purpose of the GlobalVariable.dll example is to demonstrate how code for a TradeStation-compatible DLL can be written in C++. GlobalVariable.dll is intended to demonstrate the use of the interface between EasyLanguage and external user DLL's (the ability of EasyLanguage to call external user DLL's) and to demonstrate the run-time error generating capability of the *EasyLanguage Extension Software Development Kit* (SDK). The SDK, provided in DLL form in the file *tskit.dll*, allows user-developed DLL's to access a chart's price and volume data, EasyLanguage analysis technique variables, etc. The SDK is documented in the TradeStation User Guide (TradeStation Help Menu → TradeStation User Guide → Contents Tab → EasyLanguage Reference → Books → EasyLanguage Extension SDK). The GlobalVariable.dll example code uses the SDK to generate run-time errors in TradeStation, when appropriate. See the *TradeStation-compatible DLL Support Resources* section of this document, below, for information on example code that makes use of other features of the EasyLanguage Extension SDK.

What Are Global Variables?

< Top > (Alt ← = Back)

This Global Variable DLL (GlobalVariable.dll) is provided as an example of a TradeStation-compatible DLL developed in C++. The example application is the storage of values to and retrieval of values from "global memory" (memory shared by multiple processes). This demonstration C++ code is intended to allow EasyLanguage analysis techniques that are applied in Charting, RadarScreen or OptionStation to pass values between one another and to pass to, or retrieve values from, other applications (such as Microsoft Excel®).

Values stored in global memory by EasyLanguage analysis techniques can be used in the same analysis technique that stores them (for example, the value of a variable could be stored on a tick-by-tick basis for retrieval by the same analysis technique on the next tick) or can retrieved by, and used in, analysis techniques other than the analysis technique that stores them.

Analysis techniques that use the stored values may be techniques running in applications other than the application that places the value into global memory. For example, a study applied to a RadarScreen might store some global variable values that could be used in Charting or OptionStation. Another way of saying this is to say that GlobalVariable.dll creates shared interprocess memory.

The example code for GlobalVariable.dll demonstrates how code can be written to create two types of global variable storage locations for each of five data types. The two types of storage locations are *numbered* storage locations and *named* storage locations. The five data types are Boolean values, integers, single-precision floating point numbers (called simply "floats"), double-precision floating-point numbers (called simply "doubles"), and strings (text values).

Numbered storage locations are specified by a "location number" or "element location". For example, one might speak of "global integer 239" – a numbered integer storage location of which 239 is the element location number. Or one might speak of "global float 898" – a numbered single-precision floating point storage location of which 898 is the element location number. In a sense, the element location number for numbered storage locations can be thought of as a sort of "address" of that global variable. So, extending the examples above, there could be some integer value at "address" 239 and some float value at "address" 898 (though we haven't yet specified what the values stored at those "addresses" are).

Named storage locations are specified by a "global variable name" – a text string that performs a function similar to that performed by the element location number of a numbered storage location. So, for example, one might speak of "named global double *dStore*" – a named double-precision floating-point storage location of which "dStore" is the global variable name. Or one might speak of "global string *Sym Description*" – a named string storage location of which "Sym Description" is the global variable name.

The two most basic operations that may be performed with respect to global variables are: *setting* (placing a value into a global storage location) and *getting* (retrieving a value from a global storage location). Demonstration C++ code is provided for setting and getting either named or numbered variables of each different data type. For example, code for a DLL function called `GV_SetInteger()` is provided to demonstrate how

C++ code could be written to set a numbered global integer location. A DLL function called `GV_SetNamedInt()` is provided to demonstrate how C++ code could be written to set a named global integer location. Code for DLL functions `GV_GetInteger()` and `GV_GetNamedInt()` is provided to demonstrate how code could be written for retrieval of integer values from numbered and named locations, respectively. Similar DLL demonstration functions are provided for the other data types – Boolean, float, double, and string.

Additionally, for named global variables, demonstration C++ code is provided to:

1.) Retrieve the value stored in a named variable based on the named variable's number. For example, the value stored in named float location 37 could be retrieved without having to know the name of this named location.

2.) Retrieve the name of a named variable based on the named variable's number. For example, the name of named float location 37 could be retrieved.

3.) Reset all named global variables of a user-specified data type by erasing their names and setting their values to their original (their initialization) values.

Specifications

< Top > (Alt ← = Back)

General

As mentioned above, GlobalVariable.dll code provides a demonstration of storage and retrieval of values of five data types:

1.) Boolean (true/false)

2.) Integer (whole number valued)

3.) Single-precision floating-point (also called "floats", this type is for numbers with a fractional portion)

4.) Double-precision floating-point (double the significant figures of floats)

5.) Strings (text strings of up to 250 characters each)

There are 10,000 *numbered* (as opposed to *named*) storage locations (numbered from 0 to 9999) *for each different data type*. Another way of saying this is to say that the global memory area created by GlobalVariable.dll can *simultaneously* store 10,000 Boolean values, 10,000 integers, 10,000 floats, 10,000 doubles, and 10,000 strings (of up to 250 characters each) in *numbered* variable locations.

There are 3000 *named* (as opposed to *numbered*) variable locations of each data type. Names may be assigned to each location. Names may be up to 100 characters in length. Named string global variables may contain strings of up to 250 characters. Global variable names are case-sensitive. Global variable names may contain spaces, punctuation marks, and other alphanumeric characters. Global variable names may be created from any valid expression that evaluates to a string of 100 or fewer characters.

As a side note, strings longer than 79 characters may be created in EasyLanguage by using string concatenation. Although the text between quotation marks in a string assignment statement in EasyLanguage is limited to 79 characters per set of quotation marks, longer strings may be created by concatenating two or more quoted strings, each up to 79 characters in length, together, using the "+" operator. Concatenation can be useful when it is desired to create a global variable name that is longer than 79 characters or when it is desired to create a string for storage that is longer than 79 characters.

For example, the EasyLanguage code below creates a string longer than 79 characters:

```
variables:
     MyBigString( "" ) ;

MyBigString = "Four score and seven years ago our fathers brought
forth on this " + "continent, a new nation, conceived in Liberty,
and dedicated to the " + "proposition that all men are created
equal." ;

Value1 = GVSetString( 3, MyBigString ) ;

Print( GVGetString( 3 ) ) ;
```

This code snippet demonstrates that, although the text between any pair of quotation marks must be fewer than 79 characters in length, a string larger than 79 characters in length can be formed by concatenation of multiple shorter strings. The string held by the variable MyBigString, and placed into global location 3, is longer than 79 characters.

DLL Initialization

< Top > (Alt ← = Back)

When GlobalVariable.dll is first loaded by a process, and when no other process has already loaded it, all global variable values, numbered and named, are set to initial values. Table 1, below, shows the initial values assigned to global locations of each data type. In the table, NULL represents lack of a string value.

***Table 1* – Initial Values of Global Variables and Global Variable Names**

Global Variable Type	Total Number of Locations	Initial Name of Each Location	Initial Value Stored in Location
Numbered Boolean	10,000	*Not Applicable*	FALSE
Numbered Integer	10,000	*Not Applicable*	0
Numbered Float	10,000	*Not Applicable*	0
Numbered Double	10,000	*Not Applicable*	0
Numbered String	10,000	*Not Applicable*	NULL
Named Boolean	3,000	NULL	FALSE
Named Integer	3,000	NULL	0
Named Float	3,000	NULL	0
Named Double	3,000	NULL	0
Named String	3,000	NULL	NULL

Gobal storage locations are initialized only once by the DLL. They are initialized when the DLL is first loaded by any process and are not re-initialized unless the DLL is unloaded by **all** user processes and then reloaded.

An important point is that, although a given EasyLanguage analysis technique that uses GlobalVariable.dll might be deleted from an application (deleted from a chart, for example), any global memory locations that have been set by this deleted technique will retain their values until they are changed by another technique or until **all** analysis techniques using GlobalVariables.dll are deleted or changed to an "off" status in **all** applications. Simply changing the status of a given technique that uses GlobalVariable.dll from "On" to "Off" and then back to "On" will not, in and of itself, reinitialize the global variable locations used by the technique, unless it happens to be the only technique currently using GlobalVariable.dll. The new GV_ResetAllNmd<Type> functions can be used to reset all *named* global locations *of a particular type* to their initial values, however.

It should be noted that a chart reload will have the same effect as turning an analysis technique's status from "On" to "Off" and then back to "On". That is, a chart reload will cause re-initialization of all global variables unless another chart (that does not contain the reloaded symbol) or another application is using GlobalVariable.dll.

Error Codes

< Top > (Alt ← = Back)

Any of the following actions will result in an error code being returned from the EasyLanguage wrapper functions associated with *numbered* global variables: an attempt to set or get a numbered global variable using a location number that is less than 0 or greater than 9999. In the case of Boolean global variables, a run-time error will be generated in TradeStation.

An attempt to store a character string longer than 250 characters into a *numbered* string global variable will result in storage of only the first 250 characters of the string.

Any of the following actions will result in an error code being returned from the DLL functions associated with *named* global variables: an attempt to create or retrieve a named global variable with a name shorter than 1 character in length or longer than 100 characters in length, an attempt to retrieve the value of a named global variable by name if no variable by that name exists, an attempt to retrieve the value stored in a named global variable using that variable's number if no named variable with that number exists, an attempt to set more than 3000 named locations of any data type, or an attempt to store a string value containing more than 250 characters into a named string variable. In the case of an error involving Boolean global variables, a run-time error may be generated in TradeStation in cases when no definitive error value can be returned by the function.

It should be noted that the EasyLanguage user functions that set and get numbered global variables generally contain checks to prevent global variables from being set or retrieved until at or near the end of the loaded data stream (these EasyLanguage wrapper functions generally contain a LastBarOnChart check). This is because global variables have no intrinsic date or time stamp and, therefore, there is no built-in method of synchronizing global variable values between the bars of two different charts. For this reason, most demonstration applications that call GlobalVariable.dll make their calls to the DLL only after LastBarOnChart becomes true, and when live market data is flowing, rather than on historical bars.

The global variable EasyLanguage wrapper functions associated with numbered global variables (GVSetInteger, GVGetFloat, GVGetDouble, GVSetString, etc.) will return an

error code if called on bars in a chart's history (prior to LastBarOnChart becoming true).

Having said this, it should also be noted that the global variable DLL functions do not themselves contain any *LastBarOnChart* checks. These checks exist only in the EasyLanguage wrapper functions for *numbered* global variables. (The DLL functions can be distinguished from the EasyLanguage wrapper functions by the function name. The DLL function names all contain an underscore character (_) whereas the EasyLanguage wrapper function names do not. For example, the DLL function GV_SetInteger() is wrapped (that is, it is called) by the EasyLanguage function GVSetInteger. Similarly, the DLL function GV_GetFloat() is wrapped by the EasyLanguage function GVGetFloat. There is no restriction with respect to the bar number on which the DLL functions may be called directly (without using the EasyLanguage wrapper functions). That is, the DLL functions may be directly called on any bar in history and on the currently building bar in real-time. See the section titled *Calling GV Functions Without Using the EasyLanguage Wrapper Functions*, below, for information on how to call GlobalVariable.dll functions without using the EasyLanguage wrapper functions (thereby avoiding the LastBarOnChart checks).

Global Memory Structure and Organization

< Top > (Alt ← = Back)

Each global variable location, numbered or named, is located in an area of volatile memory shared by all applications that use GlobalVariable.dll. Values stored in global variables are not retained if computer power is turned off or if all applications unload GlobalVariable.dll.

Because all applications that load GlobalVariable.dll share the GlobalVariable.dll memory space, a given global memory location is the same location, regardless which application refers to it. For example, global integer location 152 is the same location, regardless of whether that location is referred to by an analysis technique in Charting, a technique in RadarScreen, a technique in OptionStation, or referred to in another application (such as Microsoft Excel®). If any of these applications stores a value in this location, that value would overwrite any value previously stored there, even if the original value was written there by another application. Similarly, all applications that read a value from a given location at a given time will read the same value. If global integer location 152 is read by Excel, the same value will be read as will be read by an analysis technique running in TradeStation.

Because the global memory space is shared among applications in this manner, caution should be exercised when selecting the number or name of a global variable to use in each in each application. For example, if an analysis technique in Charting writes a value

to global integer location 152, it may be undesirable for a technique in RadarScreen to write to that same storage location. Or if an analysis technique in Charting writes to a global location named "MSFT", it may be undesirable for a technique in OptionStation to write to that same location name. If desired, multiple analysis techniques/applications may write to the same named or numbered location. Caution should be exercised to ensure that this is done only when intended.

It should be noted that global variable locations are distinct both by *global variable type* (numbered or named) and by *data type* (Boolean, integer, float, double, or string). Therefore, for example, global *integer* location 152 and global *float* location 152 are two different locations. Similarly, named global *float* location "MyGV" and named global *double* location "MyGV" are two different locations. Locations of different data types are different locations. Each data type has its own storage space.

Similarly, global variable locations are distinct by variable type. All *named* global variable locations are separate from *numbered* global variable locations. Named integer locations are separate from numbered integer locations; named float locations are separate from numbered float locations, etc. The named integer location that has a location number of 37, for example, and the numbered integer location that has the location number of 37 are not the same location. In *Figure 1*, below, memory locations are represented as boxes or "cells" in a table in an attempt to illustrate the separation of global memory areas by global variable type (numbered or named) and by data type.

Figure 1 – A Representation of the Global Memory Area

(not all memory locations are shown)

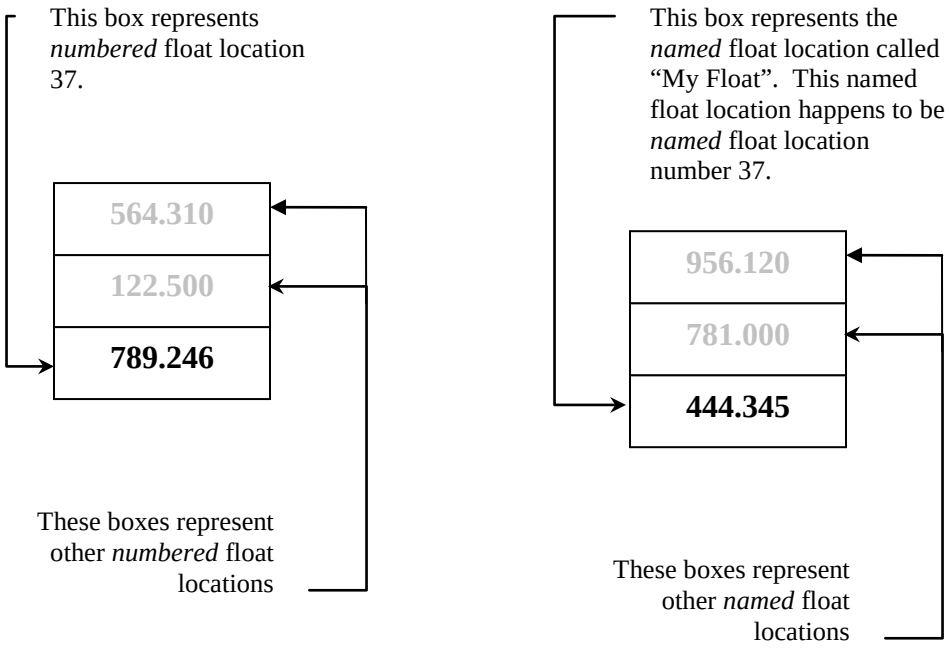
Numbered Boolean Locations ->	Location 0	Location 1	Location 2	Location 3	Location 4	Locations 5.. 6..7…	Additional Locations Up to 9999
Numbered Integer Locations ->	Location 0	Location 1	Location 2	Location 3	Location 4	Locations 5.. 6..7…	Additional Locations Up to 9999
Numbered Float Locations ->	Location 0	Location 1	Location 2	Location 3	Location 4	Locations 5.. 6..7…	Additional Locations Up to 9999
Numbered Double Locations ->	Location 0	Location 1	Location 2	Location 3	Location 4	Locations 5.. 6..7…	Additional Locations Up to 9999
Numbered String Locations ->	Location 0	Location 1	Location 2	Location 3	Location 4	Locations 5.. 6..7…	Additional Locations Up to 9999
Named Boolean Locations ->	Location 0	Location 1	Locations 2 … 3… 4…	Additional Locations Up to 2999			
Named Integer Locations ->	Location 0	Location 1	Locations 2 … 3… 4…	Additional Locations Up to 2999			
Named Float Locations ->	Location 0	Location 1	Location 2 … 3… 4…	Additional Locations Up to 2999			
Named Double Locations ->	Location 0	Location 1	Location 2 … 3… 4…	Additional Locations Up to 2999			
Named String Locations ->	Location 0	Location 1	Location 2 … 3… 4…	Additional Locations Up to 2999			

One key point about *named* global variables is that, after being created by name, they can be referred to, for the purpose of retrieving values, either by name or by number. The location number associated with a *named* global variable should not be confused with the location number associated with a *numbered* global variable.

For example, consider two float global variables, one numbered and one named. Let us presume that the numbered float global variable is number 37 (it is at float element location 37) and that the value stored at float location 37 is 789.246. Let us further presume that the named float location is named "My Float" and that the value stored at this location is 444.345. Finally, let us presume that named float location "My Float" is named float location number 37. Graphically, this situation could be represented as shown in *Figure 2* below, in which the small boxes that contain numbers represent memory locations:

Figure 2 – A Representation of the Difference between Numbered and Named Global Variable Locations

(Named locations can be referred to by both name and by location number. A named variable and a numbered variable can have the same location number without causing any conflict. See also Figure 1, above.)



From ***Figure 2*** we can see that:

1.) *Named* global locations can be referred to either by their name or by their number. The name and the number both refer to the same location in memory for named global variables. In *Figure 2* the name "My Float" and "named float at location number 37" both refer to the same space in memory.

2.) *Numbered* global locations can be referred to by their number only. Named locations have an associated number but numbered locations do not have an associated name.

3.) A named location and a numbered location that have the same location number do not refer to the same location. This is because one refers to a *named* global variable while the other refers to a *numbered* global variable. Named float location number 37 is not the same location in memory as numbered float location number 37. Just as the address "37 Main Street" is not the same location as "37 Oak Street", so *named* global variable location 37 is not the same location in memory as *numbered* global variable location 37.

EasyLanguage Wrapper Functions

< Top > (Alt ← = Back)

To make it easier to understand the EasyLanguage to DLL interface, demonstration EasyLanguage user functions are provided. Also, a demonstration user function has been provided that retrieves the version number of the GlobalVariable.dll file. These EasyLanguage functions are demonstrated below. (Additional demonstration code is contained in the workspace included in the GlobalVariable.dll download zip package.)

Set Wrapper Functions

< Top > (Alt ← = Back)

Set wrapper functions return an integer value that indicates whether the DLL function was able to successfully set (store) a value into the global memory location specified. The return value will be either equal to the *element location number* of the location set or will be a negative number to indicate an error. This is true of both numbered global variables and named global variables (see Figure 1 and related discussion, above). The value returned by a set function should always be checked by the user to ensure that the set operation was successful and that the called function did not return a negative number, which would indicate an error.

Setting a Numbered Global Variable

< Top > (Alt ← = Back)

GVSetBoolean

```
BoolSet = GVSetBoolean( 1000, TRUE ) ;
```

{ Where 1000 is the global variable element location, and TRUE is the stored value. If the EasyLanguage variable BoolSet is set to 1000 by this assignment statement then Boolean element location number 1000 was successfully set to TRUE. If BoolSet is set to –1 by the statement then an error occurred. The value of BoolSet should be checked after this statement is executed to ensure that an error has not occurred. }

GVSetInteger

```
IntRtn = GVSetInteger( 0, 1500 ) ;
```

{ Where 0 is the global variable element location, and 1500 is the stored value. If the EasyLanguage variable IntRtn is set to 0 by this assignment statement then element location number 0 was successfully set to 1500. If IntRtn is set to –1 by the statement then an error occurred. The value of IntRtn should be checked by the user after this statement is executed to ensure that an error has not occurred. }

GVSetFloat

```
SetRet = GVSetFloat( 5, Close[1] ) ;
```

{ Where 5 is the global variable element location, and the Close one bar ago is the stored value. If the EasyLanguage variable SetRet is set to 5 by this assignment statement then element location number 5 was successfully set to the value of the Close one bar ago. If SetRet is set to –1 by the statement then an error occurred. The value of SetRet should be checked by the user after this statement is executed to ensure that an error has not occurred. }

GVSetDouble

```
RtnVal = GVSetDouble( 1500, Close[1] + Open[1] ) ;
```

{ Where 1500 is the global variable element location, and the value of the expression `Close[1] + Open[1]` is the stored value. If the EasyLanguage variable `RtnVal` is set to 1500 by this assignment statement then element location number 1500 was successfully set to the value of `Close[1] + Open[1]`. If `RtnVal` is set to –1 by the statement then an error occurred. The value of `RtnVal` should be checked by the user after this statement is executed to ensure that an error has not occurred. }

GVSetString

```
Output = GVSetString( MyStringLoc, “Text to be stored in GV” ) ;
```

{ Where the EasyLanguage variable MyStringLoc contains the global variable element location to be set, and “Text to be stored in GV” is the string to be stored. If the EasyLanguage variable Output is set to MyStringLoc by this assignment statement then element location number MyStringLoc was successfully set to “Text to be stored in GV”. If Output is set to –1 by the statement then an error occurred. The value of Output should be checked by the user after this statement is executed to ensure that an error has not occurred. }

Setting a Named Global Variable

< Top > (Alt ← = Back)

For all of the EasyLanguage functions used for setting named global variables that are detailed below, error codes are as follows: A return value of –1 indicates that the global variable name was not legal (it was too long or was a null (empty) string). A return value of –2 indicates that all global named variable locations of the specified data type have been used and none is available for assignment. A return value of –3 indicates some other type of error. A non-negative return value indicates that there was no error. In this case the non-negative value returned is the numeric “element location” of the named variable. The element locations associated with named global variables have no relationship to the element locations of numbered global variables (see Figure 1 and related discussion, above).

GVSetNamedBool

```
NmdBoolSet = GVSetNamedBool( "My Order Filled", TRUE ) ;
```

{ Where “My Order Filled” is the global integer variable name, and TRUE is the stored value. If the EasyLanguage variable BoolRtn is set to a non-negative integer value by this assignment statement then a global variable was created with the name “My Order Filled” and this variable was set to TRUE. If BoolRtn is set to a negative number by the statement then an error occurred. The value of BoolRtn should be checked after the statement is executed to ensure that an error has not occurred. }

GVSetNamedInt

```
IntVRtn = GVSetNamedInt( “My Order Number”, 12345 ) ;
```

{ Where “My Order Number” is the global integer variable name, and 12345 is the stored value. If the EasyLanguage variable `IntV` is set to a non-negative integer value by this assignment statement then a global variable was created with the name “My Order Number” and this variable was set to 12345. If `IntV` is set to a negative number by the

statement then an error occurred. The value of `IntV` should be checked after the statement is executed to ensure that an error has not occurred. }

GVSetNamedFloat

```
PrevCRtn = GVSetNamedFloat( "Previous Close", Close[1] ) ;
```

{ Where "Previous Close" is the global float variable name, and the Close one bar ago is the stored value. If the EasyLanguage variable PrevCRtn is set to a non-negative integer value by this assignment statement then a global variable was created with the name "Previous Close" and this variable was set to the value of the Close one bar ago. If PrevCRtn is set to a negative number by the statement then an error occurred. The value of PrevCRtn should be checked after the statement is executed to ensure that an error has not occurred. }

GVSetNamedDouble

```
ReturnVal = GVSetNamedDouble( GetSymbolName, Average( C, 10 ) ) ;
```

{ Where the string returned by the EasyLanguage function `GetSymbolName` is the name of the global double-precision variable, and the value of the expression `Average( C, 10 )` is the stored value. If the EasyLanguage variable ReturnVal is set to a non-negative integer value by this assignment statement then a global variable was created with the name returned by the function `GetSymbolName` and this variable was set to the value of the expression `Average( C, 10 )`. If ReturnVal is set to a negative number by the statement then an error occurred. The value of ReturnVal should be checked after the statement is executed to ensure that an error has not occurred. }

GVSetNamedString

```
ErrorTest = GVSetNamedString( "String 1", "Some text…" ) ;
```

{ Where the string "String 1" is the global double variable name, and the string "Some text…" is the stored value. If the EasyLanguage variable ErrorTest is set to a non-negative integer value by this assignment statement then a global variable was created with the name "String 1" and this variable was set to the value "Some text…". If ErrorTest is set to a negative number by the statement then an error occurred. The value of ErrorTest should be checked after the statement is executed to ensure that an error has not occurred. }

Resetting Named Global Variables

< Top > (Alt ← = Back)

DLL functions that "reset" named global variables have been provided, as a DLL coding demonstration, for each data type. Any one of these functions illustrate how code may be written that sets all of the variable names of a particular data type to NULL and all of the values of a particular type to their initial values. (See *Table 1*, above, for a list of the initial values of named variables of each data type.)

It is important to note that the process of "resetting" the named variables of a particular data type will "erase" **all** named variables of that type (Boolean, integer, float, double, or string). Resetting named variables of one data type does not affect named variables of any other data type. There are demonstration reset functions in the DLL for each data type.

The GlobalVariable.dll reset functions return the number of named variable locations reset. A return value of 0 (zero) indicates that no named locations were reset. This can occur if all named global locations of a particular data type are already set to their default values (all names are NULL).

The EasyLanguage wrapper functions that can be used to call GlobalVariable.dll reset functions are illustrated in the examples below:

GVResetAllNmdBools

```
ResetBoolRtn = GVResetAllNmdBools ;
```

{ If the EasyLanguage variable ResetBoolRtn is assigned a positive integer value by this statement then the value found in ResetBoolRtn after the execution of this statement is the number of named Boolean global variables reset (by having their name set to NULL and their value set to FALSE). If the EasyLanguage variable ResetBoolRtn is set to 0 (zero) by this statement then no named locations were reset by this statement. }

GVResetAllNmdInts

```
ResetMyInts = GVResetAllNmdInts ;
```

{ If the EasyLanguage variable ResetMyInts is assigned a positive integer value by this statement then the value found in ResetMyInts after the execution of this statement is the number of named integer global variables reset (by having their name set to NULL and their value set to 0). If the EasyLanguage variable ResetMyInts is set to 0 (zero) by this statement then no named locations were reset by this statement. }

GVResetAllNmdFlts

```
NumFltsReset = GVResetAllNmdFlts ;
```

{ If the EasyLanguage variable NumFltsReset is assigned a positive integer value by this statement then the value found in NumFltsReset after the execution of this statement is

the number of named float global variables reset (by having their name set to NULL and their value set to 0.0). If the EasyLanguage variable NumFltsReset is set to 0 (zero) by this statement then no named locations were reset by this statement. }

GVResetAllNmdDbls

```
NumDblsErsd = GVResetAllNmdDbls ;
```

{ If the EasyLanguage variable NumDblsErsd is assigned a positive integer value by this statement then the value found in NumDblsErsd after the execution of this statement is the number of named double global variables reset (by having their name set to NULL and their value set to 0.0). If the EasyLanguage variable NumDblsErsd is set to 0 (zero) by this statement then no named locations were reset by this statement. }

GVResetAllNmdStrs

```
ResStrsRtn = GVResetAllNmdStrs ;
```

{ If the EasyLanguage variable ResStrsRtn is assigned a positive integer value by this statement then the value found in ResStrsRtn after the execution of this statement is the number of named string global variables reset (by having their name set to NULL and their value set to NULL). If the EasyLanguage variable ResStrsRtn is set to 0 (zero) by this statement then no named locations were reset by this statement. }

Get Wrapper Functions

< Top > (Alt ← = Back)

Get wrapper functions that are used to get values from *numbered* global variable locations (as opposed to Get functions used to retrieve values from *named* global variable locations) return the value stored at the global variable location specified in the function call. If there is an error then either a –1 or a NULL (empty) string will be returned to indicate an error condition. In the case of Get wrapper functions used to retrieve a number, a –1 will be returned if there is an error. In the case of the Get function used to retrieve a string (GVGetString), a NULL (empty) string will be returned in the case of an error. There is no way to distinguish between a return value that is retrieved from global memory and is equal to –1 (or equal to the NULL string) and an error condition.

Get wrapper functions used to get values from *named* global variable locations return the value stored at the global variable location specified in the function call or a user-specified value to indicate an error condition. The caller passes the desired error indication value to the function as a parameter. The functions used with *numbered* global variables, unlike those used with *named* global variables, do not allow the user to set the error code.

There is no way to distinguish between a return value that is retrieved from global memory and is equal to the user-specified error indication value and an error condition (both cases return a value equal to the user-specified error condition indication value). For this reason, care should be exercised when choosing the value to be returned by the function when an error occurs. The error indication value should be selected, if possible, to be a value that is outside the range of acceptable values for the global variable. For example, if a value that is known to have a range between -100 and 100 is being retrieved from a global variable, the user may wish to specify a number outside this range (such as 999) as the error return value.

Getting the Value Contained in a Numbered Global Variable Location

< Top > (Alt ← = Back)

GVGetBoolean

```
BoolGet = GVGetBoolean( 1000 ) ;
```

{ Assign to the EasyLanguage variable BoolGet the Boolean value at element location 1000. If BoolGet is assigned the value TRUE by this statement then the value found at Boolean element location 1000 is the value TRUE. If BoolGet is assigned FALSE by this statement then an error may have occurred (either an error occurred or the value found at element location 1000 was the value FALSE). }

GVGetInteger

```
MyInt = GVGetInteger( 0 ) ;
```

{ Assign to the EasyLanguage variable MyInt the integer value at element location 0. If MyInt is assigned a value other than –1 by this statement then the value found at integer element location 0 is the value found in MyInt. If MyInt is assigned –1 by this statement then an error may have occurred (either an error occurred or the value found at element location 0 was –1). }

GVGetFloat

```
FltVal = GVGetFloat( 5 ) ;
```

{ Assign to the EasyLanguage variable FltVal the float value found at element location 5. If FltVal is assigned a value other than –1 by this statement then the value found at float element location 5 is the value found in FltVal. If FltVal is assigned –1 by this statement then an error may have occurred (either an error occurred or the value found at element location 5 was –1). }

GVGetDouble

```
ValRetn = GVGetDouble( 105 ) ;
```

{ Assign to the EasyLanguage variable ValRetn the double-precision value found at element location 105. If ValRetn is assigned a value other than –1 by this statement then the value found at float element location 105 is the value found in ValRetn. If ValRtn is assigned –1 by this statement then an error may have occurred (either an error occurred or the value found at element location 105 was –1). }

GVGetString

```
MyStr = GVGetString( 5005 ) ;
```

{ Assign to the EasyLanguage variable MyStr the string value found at element location 5005. If MyStr is assigned a value other than the NULL string (an empty string) by this statement then the value found at sting element location 5005 is the value found in MyStr. If MyStr is assigned a NULL string (an empty string) by this statement then an error may have occurred (either an error occurred or the value found at element location 5005 was the NULL string). The EasyLanguage variable MyStr must be a string type variable in order to receive the string returned by the GVGetString function. }

Getting the Value Contained in a Named Global Variable Location

< Top > (Alt ← = Back)

GVGetNamedBool

```
NmdBoolGet = GVGetNamedBool( "My Order Filled" ) ;
```

{ Assign to the Boolean EasyLanguage variable NmdBoolGet the Boolean value stored in named Boolean global variable “My Order Filled”. If NmdBoolGet is assigned a value by this statement, and a run-time error is not generated, then the value found in the global Boolean storage location named “My Order Filled” is the value found in NmdBoolGet. Error conditions will be indicated by the generation of a run-time error in TradeStation. }

GVGetNamedBoolByNum

```
BoolByNum = GVGetNamedBoolByNum( 37 ) ;
```

{ Assign to the EasyLanguage variable BoolByNum the Boolean value stored in named Boolean global variable the location number of which is 37. If BoolByNum is assigned a value by this statement, and no run-time error occurs, then the value found in global named Boolean location number 37 is the value found in BoolByNum. An error condition will be indicated by generation of a run-time error in TradeStation. Location numbers for global named Boolean variables are returned by the GVSetNamedBool function. }

GVGetNamedInt

```
MyInt = GVGetNamedInt( “An Int Var’s Name”, -999 ) ;
```

{ Assign to the EasyLanguage variable MyInt the integer value stored in named integer global variable “An Int Var’s Name”. If MyInt is assigned a value other than –999 by this statement then the value found in the global integer storage location named “An Int Var’s Name” is the value found in MyInt. If MyInt is assigned the value –999 by this statement then an error may have occurred (either an error occurred or the value found in the integer location “An Int Var’s Name” was –999). }

GVGetNamedIntByNum

```
IntVal = GVGetNamedIntByNum( 10, -999 ) ;
```

{ Assign to the EasyLanguage variable IntVal the integer value stored in named integer global variable the location number of which is 10. If IntVal is assigned a value other than –999 by this statement then the value found in global named integer location number 10 is the value found in IntVal. If IntVal is assigned the value –999 by this statement then an error may have occurred (either an error occurred or the value found in the integer location 10 was –999). Location numbers for global named integer variables are returned by the GVSetNamedInt function. }

GVGetNamedFloat

```
TheClose = GVGetNamedFloat( “Closing Value”, -999.99 ) ;
```

{ Assign to the EasyLanguage variable TheClose the value stored in named floating-point global variable “Closing Value”. If TheClose is assigned a value other than –999.99 by this statement then the value found in the global integer storage location named “Closing Value” is the value found in TheClose. If TheClose is assigned the value –999.99 by this statement then an error may have occurred (either an error occurred or the value found in the location “Closing Value” was –999.99). }

GVGetNamedFltByNum

```
Boat = GVGetNamedFltByNum( 10, -999.999 ) ;
```

{ Assign to the EasyLanguage variable Boat the float value stored in the named float global variable the location number of which is 10. If Boat is assigned a value other than –999.999 by this statement then the value found in global named float location number 10 is the value found in Boat. If Boat is assigned the value –999.999 by this statement then an error may have occurred (either an error occurred or the value found in named float location 10 was –999.999). Location numbers for global named float variables are returned by the GVSetNamedFloat function. }

GVGetNamedDouble

```
DoubleDuty = GVGetNamedDouble( “Super Algorithm”, ErrorCode ) ;
```

{ Assign to the EasyLanguage variable DoubleDuty the value stored in global named double-precision variable “Super Algorithm”. If DoubleDuty is assigned a value other

than that of the variable ErrorCode by this statement then the value found in the global double storage location named “Super Algorithm” is the value found in DoubleDuty. If DoubleDuty is assigned the value of the variable ErrorCode by this statement then an error may have occurred (either an error occurred or the value found in the location “Super Algorithm” was equal to the value of the variable ErrorCode). }

GVGetNamedDblByNum

```
Dual = GVGetNamedDblByNum( 50, -98765.9876543 ) ;
```

{ Assign to the EasyLanguage variable Dual the double value stored in the global named double variable the loacation number of which is 50. If Dual is assigned a value other than –98765.9876543 by this statement then the value found in the global named double location number 50 is the value found in Dual. If Dual is assigned the value –98765.9876543 by this statement then an error may have occurred (either an error occurred or the value found in named double location 50 was –98765.9876543). Location numbers for global named double variables are returned by the GVSetNamedDouble function. }

GVGetNamedString

```
Twine = GVGetNamedString( “Rope”, “Error Message” ) ;
```

{ Assign to the EasyLanguage string variable Twine the value stored in global named string variable “Rope”. If Twine is assigned a value other than “Error Message” by this statement then the value found in the global string storage location named “Rope” is the value found in Twine. If Twine is assigned the string “Error Message” by this statement then an error may have occurred (either an error occurred or the value found in the location “Rope” was “Error Message”). The EasyLanguage variable Twine must be a string type variable in order to receive the string returned by the GVGetNamedString function. }

GVGetNamedStrByNum

```
ValOfStr = GVGetNamedStrByNum( 10, “My error message” ) ;
```

{ Assign to the EasyLanguage string variable ValOfStr the string value stored in the global named string variable the number of which is 10. If ValOfStr is assigned a value other than “My error message” by this statement then the value found in the global named string location number 10 is the value found in ValOfStr. If ValOfStr is assigned the value “My error message” by this statement then an error may have occurred (either an error occurred or the value found in the named string at location 10 was “My error message”). Location numbers for global named string variables are returned by the GVSetNamedString function. }

*Getting the **Name** of a Named Global Variable*

< Top > (Alt ← = Back)

GVGetBoolNameByNum

```
BoolNm = GVGetBoolNameByNum( 5, "Get Bool name BY NUM Error!" ) ;
```

{ Assign to the EasyLanguage string variable BoolNm the name of the global named Boolean the location number of which is 5. If BoolNm is assigned a value other than "Get Bool name BY NUM Error!" by this statement then the name of the global named Boolean at location number 5 is the value found in BoolNm. If BoolNm is assigned the value "Get Bool name BY NUM Error!" by this statement then an error may have occurred (either an error occurred or the name of the named Boolean at location 5 was "Get Bool name BY NUM Error!"). Location numbers for global named Boolean variables are returned by the GVSetNamedBool function. }

GVGetIntNameByNum

```
MyVarName = GVGetIntNameByNum( 10, "No name retrieved!" ) ;
```

{ Assign to the EasyLanguage string variable MyVarName the name of the global named integer the location number of which is 10. If MyVarName is assigned a value other than "No name retrieved!" by this statement then the name of the global named integer at location number 10 is the value found in MyVarName. If MyVarName is assigned the value "No name retrieved!" by this statement then an error may have occurred (either an error occurred or the name of the named integer at location 10 was "No name retrieved!"). Location numbers for global named integer variables are returned by the GVSetNamedInt function. }

GVGetFltNameByNum

```
MyFloatName = GVGetFltNameByNum( 10, "Error message" ) ;
```

{ Assign to the EasyLanguage string variable MyFloatName the name of the global named float the location number of which is 10. If MyFloatName is assigned a value other than "Error message" by this statement then the name of the global named float at location number 10 is the value found in MyFloatName. If MyFloatName is assigned the value "Error message" by this statement then an error may have occurred (either an error occurred or the name of the named float at location 10 was "Error message"). Location numbers for global named float variables are returned by the GVSetNamedFloat function. }

GVGetDblNameByNum

```
MyDblNm = GVGetDblNameByNum( 103, "Problem" ) ;
```

{ Assign to the EasyLanguage string variable MyDblNm the name of the global named double the location number of which is 103. If MyDblNm is assigned a value other than "Problem" by this statement then the name of global named double location number 103

is the value found in MyDblNm. If MyDblNm is assigned the value “Problem” by this statement then an error may have occurred (either an error occurred or the name of named double location 103 was “Problem”). Location numbers for global named double variables are returned by the GVSetNamedDouble function. }

GVGetStrNameByNum

```
StringV = GVGetStrNameByNum( 375, “E” ) ;
```

{ Assign to the EasyLanguage string variable StringV the name of the global named string the location number of which is 375. If StringV is assigned a value other than “E” by this statement then the name of the global named string at location number 375 is the value found in StringV. If StringV is assigned the value “E” by this statement then an error may have occurred (either an error occurred or the name of the named string at location 375 was “E”). Location numbers for global named string variables are returned by the GVSetNamedString function. }

GVGetVersion Wrapper Function

< Top > (Alt ← = Back)

GVGetVersion

```
Version = GVGetVersion ;
```

{ Assign to the EasyLanguage string variable Version the value returned by the function GVGetVersion. For GlobalVariable.dll version 2.2, the string variable Version should be assigned the string “2.20.00.0000” by this statement. }

Calling GV Functions Without Using the EasyLanguage Wrapper Functions

< Top > (Alt ← = Back)

The following example EasyLanguage indicator code illustrates the calling of the GlobalVariable.dll functions GV_SetInteger() and GV_GetInteger() without the use of the GVSetInteger and GVGetInteger EasyLanguage wrapper functions. Calling the DLL functions directly, without using the EasyLanguage wrapper functions, avoids the LastBarOnChart checks that are built into the wrapper functions, as may be desirable when attempting to store or retrieve a global variable value on an historical bar.

```
DefineDLLFunc:  "GlobalVariable.dll", int, "GV_GetInteger", int ;
DefineDLLFunc:  "GlobalVariable.dll", int, "GV_SetInteger", int,
int ;

inputs:
      ElementLocation( 1 ),
      GVValue( CurrentBar ) ;

variables:
      SetRtnVal( -1 ),
      GetRtnVal( -1 ) ;

SetRtnVal = GV_SetInteger( ElementLocation, GVValue ) ;
GetRtnVal = GV_GetInteger( ElementLocation ) ;

Plot1( GetRtnVal, "Get" ) ;
```

Demonstration Studies and Workspaces

< Top > (Alt ← = Back)

Included with the GlobalVariable.dll C++ code are EasyLanguage studies and two workspaces.

Installation

< Top > (Alt ← = Back)

1.) To begin, extract the file "GlobalVariable.dll" from the Global Variable download zip file and place it into a subdirectory of your choice. Then move this file from the subdirectory in which you placed it to your installation's equivalent to the following subdirectory (substitute your current build number for the letters WXYZ and your current version number for 8.x):

C:\Program Files\TradeStation 8.x (Build WXYZ)\Program\

For example, your installation might be something like this:

C:\Program Files\TradeStation 8.0 (Build 1869)**Program**\

2.) Next, import the studies and functions from the ELD file that is included in the zip file. Be sure to import the studies and functions contained in the supplied ELD file before attempting to open either of the demonstration workspaces. Please note: some real-time anti-virus/firewall/internet security (AV/IS) products may interfere with the importation of ELD files. If you experience difficulty importing the studies and functions from the ELD file, follow these steps:

a.) Disable the AV/IS product.

b.) Deselect the option to load the AV/IS product at Windows start-up if this option was previously selected in the AV/IS software.

c.) Reboot the computer.

d.) Import the ELD file.

e.) Re-select the option to load the AV/IS product at Windows start-up if desired.

f.) Re-start the AV/IS product.

g.) Reboot the computer and ensure that the AV/IS product is loaded at Windows start-up (if this option was selected in Step e.) above).

3.) Extract the two demonstration workspaces, *GV Demo Workspace 1.tsw* and *GV Demo Workspace 2.tsw*, from the zip file and place them into your installation's equivalent to the following subdirectory for easy access (substitute your current build number for the letters WXYZ and your current version number for 8.x):

C:\Program Files\TradeStation 8.x (Build WXYZ)**MyWork**\

4.) View the ReadMe.txt file that is included in the downloaded zip package.

5.) The additional C++ code and project files included in the zip file can be viewed in Visual Studio® 6 in the normal manner.

GV 2.2 Demo Workspace 1

< Top > (Alt ← = Back)

Note: Before attempting to open either of the two demonstration workspaces, close any workspaces that you might already have open that use global variables (workspaces you may have developed for a previous version of Global Variables). Also, have only one of the two demonstration workspaces open at any given time. Do not attemtpt to view GV 2.2 Demo Workspace 1 and GV 2.2 Demo Workspace 2 at the same time or the demonstration analysis techniques may not work correctly. The two demonstration workspaces may interfere with one another as they can use the same areas in global memory.

Included in the first demonstration workspace, *GV Demo Workspace 1*, are two charts and two RadarScreens. The charts share some global variables with one another. Similarly, the RadarScreens share some global variables.

Let's look at the charts first. The 1-minute chart contains an indicator that sets some numbered global variables. The values of a "fast" and "slow" moving average are placed into global memory. The other chart, the 500-tick chart, retrieves these moving average values from global memory and assigns these values to local (EasyLanguage) variables. These values are then plotted. In this way, values calculated on the 1-minute chart are transferred to, and plotted on, the 500-tick chart.

At times the Global Variables that are stored or retrieved may be one tick or more behind what you might expect. This is because it takes one tick in the sending chart to run the code that stores values into global variables. Then it takes at least one additional tick for the value to be retrieved by the receiving chart.

The studies that are applied to the two charts are described below:

GV Set MA Values Indicator:

This indicator is applied to the 1-minute chart.

Inputs:

GVLocationFast(1), { global variable element location at which to store the GVFastMA value }

GVLocationSlow(2), { global variable element location at which to store the GVSlowMA value }

GVFastLen(10), { length of the "fast" moving average }

GVSlowLen(50) ; { length of the "slow" moving average }

GV Get MA Values Indicator:

This indicator retrieves the moving average values that are placed into global memory by *GV Set MA Values*, the values stored by the 1-minute chart in "real-time". It plots the values on the 500-tick chart, in "real-time" (and not on historical bars).

Note: Values are plotted on the 500-tick chart only at the end of the chart (on bars after *LastBarOnChart* becomes true, at the far right of the chart). Historical values should not be plotted. For this reason, this example is best viewed when live market data is flowing.

Let's now turn our attention to the two RadarScreens in the workspace, the RadarScreens labeled *MA Sender* and *MA Retriever*. These RadarScreens demonstrate the use of *named* global variables. They demonstrate a method of passing values generated by symbols of one bar interval, on one RadarScreen, to a second RadarScreen containing the same symbols but of a different bar interval. The value of the 200-day moving average is calculated for all of the symbols in the *MA Sender* RadarScreen. This 200-day moving average value is then passed to the *MA Retriever* RadarScreen, which contains the same symbols as the *MA Sender* RadarScreen, but which has the bar interval of all symbols set to 15 ticks.

If *GV 2.2 Demo Workspace 1* is opened before or after regular market hours, it may be necessary to reload the symbols on the *MA Retriever* RadarScreen in order to ensure that the latest values sent by the *MA Sender* RadarScreen are retrieved from global memory. If this is necessary, it should be done after the completion of calculation of values in the *MA Sender* RadarScreen. To reload the data for all symbols in the *MA Retriever* RadarScreen, select the "Symbol" column of this RadarScreen by clicking the "Symbol" column heading in the gray area then, with all symbols in this column selected, click on the *View* menu and select *Refresh*.

The indicators *MA Sender* and *MA Retriever* demonstrate two things. First, that the symbol name, retrieved automatically using the EasyLanguage reserved word *Symbol*, can be used as the name of the named global variable. This allows the passing of values from one RadarScreen to the other without the user ever having to choose or manually enter a global variable name or number. Second, these indicators demonstrate how a value of interest based on values of one bar interval can be retrieved by symbols with another bar interval for use in current calculations. This use of data streams of different bar intervals in RadarScreen is similar in principle to the use of multiple data streams in a single chart in Charting, albeit without the ability to reference historical values of the "second data stream".

Here is a description of the indicators that have been applied to the two RadarScreens:

MA Sender Indicator (applied to the daily bar interval symbols in the *MA Sender* RadarScreen):

Inputs:

MALength(200) – The number of bars to be used in the daily moving average calculation.

This indicator displays the calculated moving average and sends the calculated values to global variable storage locations. Each row of the RadarScreen stores the moving average value calculated for that row in a global variable that has the symbol for its name. For example, the AXP row stores its moving average value in a global variable named "AXP".

MA Retriever Indicator (applied to the 15-tick bar interval symbols in the *MA Retriever* RadarScreen):

This indicator retrieves the moving average values that were placed into global memory by the indicator *MA Sender*. Each row of the RadarScreen uses its symbol name to retrieve the global variable value that is intended for it. For example, the AXP row retrieves its moving average value from the global variable named "AXP".

Both of these indicators contain the following statement:

```
Value1 = Ticks ; { force RadarScreen to update every tick }
```

This statement is used, as the comment in curly braces indicates, to force the indicators to update every tick when used in RadarScreen.

GV 2.2 Demo Workspace 2

< Top > (Alt ← = Back)

After you have finished viewing *GV 2.2 Demo Workspace **1***, it should be closed, as should any other applications that use global variables, before any attempt is made to open *GV 2.2 Demo Workspace **2***. Once all workspaces that use global variables have been closed, *GV 2.2 Demo Workspace 2* may be opened. Allow a few seconds for all of the values in the OptionStation Analysis window to load. If you are viewing *GV 2.2 Demo Workspace 2* first, without previously having viewed *GV 2.2 Demo Workspace 1*, be sure that you have imported the ELD file included in the download package before attempting to open the workspace.

Once values are visible in the OptionStation analysis window, click anywhere in either of the QQQ charts that are included in the workspace. Then, press the <*Ctrl*>-<*R*> key sequence or, from the *View* menu, select *Refresh* > *Reload*. This will reload data on the charts (because the same symbol is displayed in both charts, reloading either of them will automatically reload the other). When the charts reload, some horizontal lines and text will appear, so that the chart will resemble that shown in *Figure 3*, below. (The price bars, horizontal red lines, and yellow text in the figure have been made boldface for easier viewing in this document. In the workspace, these items are not boldface. Also, these items are not the same colors in both charts.)

***Figure 3* – The Chart in *GV 2.2 Demo Workspace 2* following a Refresh**

It may be necessary to change the text font applied to the charts in order to view this demonstration clearly. To do this, right-click any piece of text and choose *Format 'Text'* from the right-click menu. To the Chart Analysis dialog box that appears asking whether you wish to format the analysis technique, choose *No*. On the *Font* tabo of the *Format Text* dialog box that then appears choose *Courier New – Regular – 9* point as the desired font. Click the check box labeled "Set as default". Please note: This change will affect all charts on which you use text objects. Then reload the chart (*<Ctrl>-<R>).*

The horizontal lines and text reflect information passed to the chart from the OptionStation Analysis window using global variables. The indicator *Option Info Sender* has been applied to both the puts and the calls in the OptionStation Analysis window (see the column headed "OptInfo" that appears on both the call (left) and put (right) sides of the options pane). This indicator places some values into global variables so that they can be retrieved by *Option Info Retrievr*, an indicator applied to each of the charts. Among the values placed in global memory by the *Option Info Sender* indicator that is applied in OptionStation are:

1.) Options symbols
2.) Options expiration month and year information
3.) Option types (put or call)
4.) Option strike prices
5.) Option bids and asks
6.) Implied volatility based on option bids and asks

Based on user inputs, the indicator *Option Info Retrievr*, which has been applied to each of the charts, retrieves information from global memory for certain options (options of user selected type, expiration month, and expiration year). It then draws red lines at the strike price levels for these options and labels the lines with information about the options. The text information included for each strike price is: the option symbol, the bid, the ask, the implied volatility of the bid, and the implied volatility of the ask.

The options information placed into global memory by the indicator *Option Info Sender* is determined by the values of the following user inputs to this indicator:

ExpMonth – Numeric value representing the expiration month of the options on which information is to be transferred to the chart. Month numbers may range from 1 to 12,

inclusive. An *ExpMonth* month number of 1 indicates that January option information is desired. An *ExpMonth* number of 6 indicates that June option information is desired, etc.

ExpYear – Numeric value representing the four-digit expiration number of the options on which information is to be transferred to the chart. For example, an *ExpYear* of 2005 indicates that options expiring in *ExpMonth* of 2005 is to be transferred to global memory.

Option_Type – This input may be set to *Put* if information on put options is to be placed in global memory, or may be set to *Call* if information on calls is desired.

To change the values of these inputs, right-click the the column heading *OptInfo* in the options pane of the OptionStation Analysis window. Note: there are column headings of *OptInfo* on both the call (left) side of the options pane and on the put (right) side of the options pane. These may be separately formatted.

The options information retrieved from global memory by the indicator *Option Info Retrievr*, which has been inserted into each of the charts, is determined by the values of the following user inputs to this indicator:

ExpMonth – Numeric value representing the expiration month of the options on which information is to be retrieved from global memory. Month numbers may range from 1 to 12, inclusive. An *ExpMonth* month number of 1 indicates that January option information is desired. An *ExpMonth* number of 6 indicates that June option information is desired, etc.

ExpYear – Numeric value representing the four-digit expiration number of the options on which information is to be retrieved from global memory. For example, an *ExpYear* of 2005 indicates that options expiring in *ExpMonth* of 2005 is to be retrieved from global memory.

Option_Type – This input may be set to *Put* if information on put options is to be placed on the chart, or may be set to *Call* if information on calls is desired. One of the charts has this input set to *Call* and the other chart has this information set to *Put*.

TL_Color – A numeric value or reserved word representing the desired color of the horizontal lines plotted by the indicator.

TL_Thickness – A numeric value representing the desired thickness of the horizontal lines plotted by the indicator. Increasing numbers will result in lines of increasing thickness.

Text_Color – A numeric value or reseved word representing the desired color of the text placed on the horizontal lines by this indicator.

TradeStation-compatible DLL Support Resources

< Top > (Alt ← = Back)

Should you have questions about the GlobalVariable.dll example, another TradeStation-compatible DLL, or about development of your own TradeStation-compatible DLL's, please post them in the EasyLanguage-DLL category of TradeStation's online support forums, here:

http://www.tradestationsupport.com/discussions/forum.aspx?Forum_ID=213&selCategory=506&selType=552&selVersion=1422&selStatus=531&selSortField=1&selSortOrder=1&selFrom=0&Page=1&

There is a short registration process required in order to access the online support forums. After you've registered and logged in, simply click on the link (or button) that says "New Topic" to go to the form on which you can enter your question. When completing the new topic form, be sure to select "EasyLanguage-DLL" as the category for your question.

The Help menu in TradeStation provides an alternative route to the online EasyLanguage-DLL Support Forum. Just click *Help* on the main TradeStation menu and select "EasyLanguage-DLL Support Forum" from the Help menu.

The latest documentation of the EasyLanguage Extension SDK is also available through the TradeStation Help menu, as mentioned above (click TradeStation Help Menu → TradeStation User Guide → Contents Tab → EasyLanguage Reference → Books → EasyLanguage Extension SDK).