EasyLanguage Extension Software Development Kit

The Standard in Rule-Based Trading™

Important Information and Disclaimer:

TradeStation Securities, Inc. seeks to serve institutional and active traders. Please be advised that active trading is generally not appropriate for someone of limited resources, limited investment or trading experience, or low risk tolerance, or who is not willing to risk at least $50,000 of capital.

This book discusses in detail how TradeStation is designed to help you develop, test and implement trading strategies. However, TradeStation Securities does not provide or suggest trading strategies. We offer you unique tools to help you design your own strategies and look at how they could have performed in the past. While we believe this is very valuable information, we caution you that simulated past performance of a trading strategy is no guarantee of its future performance or success. We also do not recommend or solicit the purchase or sale of any particular securities or securities derivative products. Any securities symbols referenced in this book are used only for the purposes of the demonstration, as an example ---- not a recommendation.

Finally, this book shall discuss automated electronic order placement and execution. Please note that even though TradeStation has been designed to automate your trading strategies and deliver timely order placement, routing and execution, these things, as well as access to the system itself, may at times be delayed or even fail due to market volatility, quote delays, system and software errors, Internet traffic, outages and other factors.

All proprietary technology in TradeStation is owned by TradeStation Technologies, Inc., an affiliate of TradeStation Securities, Inc. The order execution services accessible from within TradeStation are provided by TradeStation Securities, Inc. pursuant to a technology license from its affiliate and its authority as a registered broker-dealer and introducing broker. All other features and functions of TradeStation are provided directly by TradeStation Technologies. TradeStation® and EasyLanguage® are registered trademarks of TradeStation Technologies, Inc. "TradeStation," as used in this document, should be understood in the foregoing context.

Options trading is not suitable for all investors. Your account application to trade options will be considered and approved or disapproved based on all relevant factors, including your trading experience. Automated trading, as it relates to direct-access electronic placement and execution of equity options trades, requires manual one-click verification before order is sent. Please visit www.TradeStation.com to view the document titled Characteristics and Risks of Standardized Options.

EasyLanguage Extension Software Development Kit

Contents

Overview

The EasyLanguage Extension SDK is intended for use in conjunction with analysis techniques written in EasyLanguage. It makes it easier for you to integrate function libraries developed in another programming language with analysis techniques developed in EasyLanguage.

Used in conjunction with an EasyLanguage analysis technique, the SDK provides access to the price and volume data of the chart (or RadarScreen symbol) to which the EasyLanguage analysis technique is applied, and to the EasyLanguage variables and arrays contained in the analysis technique. Certain EasyLanguage analysis technique settings, like "maxbarsback," are accessible through the SDK, as are some values relevant to calculation, like *CurrentBar.* Additionally, the SDK provides the ability to raise a run-time error in TradeStation, in a manner similar to that of the EasyLanguage reserved word *RaiseRunTimeError.*

This version of the EasyLanguage SDK does not provide access to all TradeStation reserved words, nor can functions written in EasyLanguage be called directly. (Of course, many programming languages have run-time libraries that perform many of the functions available through EasyLanguage reserved words, especially mathematical functions. Market/Trading functions are a notable exception, because of the specific nature of their use.)

The SDK does not provide the ability to plot directly on a chart, nor to directly buy or sell. However, these functions can be performed by EasyLanguage studies or strategies. The SDK does not provide the ability to access data from TradeStation windows other than those into which an EasyLanguage analysis technique has been inserted. For example, the SDK does not provide access to values shown in the "Matrix" or "News" windows of TradeStation.

This reference provides the basic information necessary to make the SDK functions available to your DLL, and to call your DLL functions from EasyLanguage in a manner that will provide your DLL with the information it needs to use the SDK. Because use of the EasyLanguage Extension SDK is an advanced topic in EasyLanguage, this reference presumes some familiarity with EasyLanguage. Some familiarity with C++ is presumed by the C++ code examples. Some example code may be Visual C++® (Microsoft) specific.

Style Conventions Used In This Reference

The library descriptions presented in this reference are usually provided in a "pseudocode" format. That is, they do not follow any particular programming language's syntax requirements. When language-specific examples are provided, it will be clear from the context that the example is language-specific.

EasyLanguage is not a case-sensitive language. Any capitalization added to the EasyLanguage code in this document is provided for the sole purpose of enhancing code readability. (There is an exception to the rule that EasyLanguage is not case-sensitive. When a DLL function name is given in an external statement, the function name is case-sensitive. However, the function name is not case sensitive when used elsewhere in the code (outside the

external statement). More detail on the external statement is provided below, in the section on declaration of DLL functions.)

The following is a description of the format used for all code in this document:

Courier New font – Program code is presented using the Courier New font. This font is used to distinguish code segments from descriptive text. Also, this font is a "monospace" font; all letters are the same width horizontally. This aids in creating uniform spacing of code blocks.

Italic – Italics are used in property or method syntax examples to distinguish words that are incidental to the property or method being described.

Boldface Italic – Boldface italics are used in property or method syntax examples to distinguish the names of the actual properties or methods being described.

() – Parentheses are used in property and method syntax examples to indicate statement elements that are optional to the use of a property or method. Such examples usually present only one possible use of the property or method.

Italic and boldface fonts, and parentheses, have their usual meanings when used in this document outside code blocks.

Style Examples:

Pseudocode:

*InterfaceObject.**AsDouble[**nBarsBack**] (= dValue**;)*

C++

*pMyVar->***AsDouble[**nBarsBack**]** *(= 17.5475* ;)

Using the SDK

The EasyLanguage Extension SDK is implemented in the file tskit.dll. This file is installed, by default, in the *Program* subdirectory of the directory into which TradeStation is installed.

EasyLanguage exposes its run-time information through a COM interface, which is implemented in tskit.dll. This dispatch interface has the name IEasyLanguageObject.

In order for your DLL to use the SDK, and to make a DLL TradeStation-compatible, several things must be accomplished. These requirements are listed below. Some requirements are specific to DLL's that use the SDK. Whether a requirement applies to all DLL's or to only DLL's that use the SDK is noted in the detailed list of requirements below. Here is the list of requirements:

1.) *The DLL must import tskit.dll.* Every programming language has a specific means of accomplishing this. A C++ example is provided below. This requirement applies to the DLL only if the DLL uses the SDK. This is not a requirement of DLL's that do not use the SDK.

2.) *The EasyLanguage analysis technique that calls the DLL function must declare the DLL function in an external statement.* This requirement applies to the DLL only if the DLL uses the SDK. This is not a requirement of DLL's that do not use the SDK. If the DLL does not use the SDK then either the external reserved word or the legacy reserved word DefineDLLFunc may be used to declare DLL functions.

3.) *A pointer to the IEasyLanguageObject interface of the EasyLanguage analysis technique must be passed from the analysis technique to the DLL.* This is accomplished using the reserved word self in the EasyLanguage analysis technique. This requirement applies to the DLL only if the DLL uses the SDK. This is not a requirement of DLL's that do not use the SDK.

4.) *The DLL must export functions using the __stdcall calling convention.* This is a requirement of all TradeStation-compatible DLL's. It applies to both DLL's that use the SDK and to DLL's that do not use the SDK.

5.) *Optional: The #Events and #End compiler directives may be used in conjunction with the reserved words OnCreate and OnDestroy to specify DLL functions to be run when an EasyLanguage analysis technique that calls DLL functions is activated or deactivated.* This option applies only to DLL's that use the SDK.

Additional detail on these requirements is provided below:

1.) Importation of tskit.dll

In order to use the SDK, your DLL project must include a reference to tskit.dll. As previously mentioned, tskit.dll is located, by default, in the *Program* subdirectory of the directory into which TradeStation is installed. Unlike the legacy SDK, there are no header files included; they are not needed. To make use of the SDK, simply include the following line in your Visual C++® DLL Project:

```
#import "c:\Program Files\TradeStation\Program\tskit.dll" no_namespace
```

This #import directive results in the creation of two type library header files: a primary header file named "tskit.tlh", and a secondary header file named "tskit.tli". These files are created during DLL project compilation and are placed, by the compiler, in the DLL project subdirectory. Both files are read and compiled as if the primary header file had been named in an #include directive. (The secondary header file is compiled because it is named in an #include directive at the end of the primary header file.)

2.) Declaration of DLL Functions

The reserved word external is used in EasyLanguage to declare a DLL function that resides outside EasyLanguage. Such a declaration may contain path information which specifies the

location of the DLL file that contains the external function. The external statement must contain the name of the DLL file, the DLL function return type, and the types of all parameters being passed to the DLL function.

The syntax of an external statement is:

external: ["<PATH>",] [RETURN TYPE,] "<DLL FUNCTION NAME>", [ARGUGMENT 1 TYPE, ARGUMENT 2 TYPE, …] ;

Here is a detailed description of the syntax specified above. An example external statement is provided below.

PATH is the Windows directory path to the DLL file. PATH is an optional parameter. If a path is not specified, the subdirectory "C:\Program Files\TradeStation\Program\" is presumed to be the subdirectory that contains the DLL file. If the DLL is not found in the \TradeStation\Program subdirectory, then the directories in the computer's system path statement will be searched. If the DLL is not located in any of those directories, a run-time error will occur.

RETURN TYPE is the data type of the value that the DLL function will return (int, bool, double, float, etc.)

DLL FUNCTION NAME is a mandatory parameter of the external statement. It is the name of the function exported by the DLL. The function name must be placed inside quotation marks. As noted above, function names are case-sensitive when used in the external statement but are not case-sensitive when used elsewhere in EasyLanguage code.

ARGUMENT 1 TYPE, ARGUMENT 2 TYPE, … are optional parameters of the external statement. The argument types are the data types of the arguments received by the DLL function. The supported argument data types are listed in the Type Library Reference section, below.

Here's an example external statement:

external: "c:\MyDLLs\MyDLLFile.dll", double, "MYFUNCTIONNAME", int ;

This example statement declares a DLL function MYFUNCTIONNAME that is exported from the DLL file MyDLLFile.dll. The file MyDLLFile.dll is located in the C:\MyDLLs\ subdirectory. The function MYFUNCTIONNAME receives one integer argument and returns one double-precision float value.

It is a good practice to provide names for DLL function arguments and return values using comment braces, as in the following external statement:

external: "c:\MyDLLs\MyDLLFile.dll", {OscA}double, "MYFUNCTIONNAME", {Length}int ;

3.) IEasyLanguageObject Pointer (self)

The reserved word self is used to provide a pointer to the IEasyLanguageObject interface associated with a given analysis technique. For a DLL function to use the interfaces provided in the SDK, the DLL function will need a pointer to the IEasyLanguageObject interface associated with the EasyLanguage analysis technique. Using this pointer, the DLL function may derive other

SDK interface objects (IEasyLanguageVariable, IEasyLanguageDateTime, ITradeStationPlatform, etc.)

Here is some EasyLanguage code that declares a DLL function, then calls it:

```
external: "MyDll.dll", double, "MyADX", IEasyLanguageObject {self}, int {Length};

inputs:
        Length( 10 ) ;

variables:
        int MyELVar( 0 ) ;

MyELVar = MyADX( self, Length ) ;
```

The DLL prototype for the function MyADX might look like this:

```
int __stdcall MyADX( IEasyLanguageObject * pELObj, int nLength ) ;
```

If the first argument received by a DLL function is a pointer to an IEasyLanguageObject interface, the reserved word method may be used in conjunction with external to pass this pointer automatically. For example, the following two declarations are equivalent:

```
external method: "MyDll.dll", double, "MyADX", int {length};

external: "MyDll.dll", double, "MyADX", IEasyLanguageObject {self}, int {length};
```

4.) Standard C Calling Convention

DLL functions must be exported using the "standard C" calling convention in order for TradeStation to locate them. Additionally, exported functions should be listed in the EXPORTS section of the DLL project's module-definition (.DEF) file.

The C++ code example below illustrates the use of __stdcall notation in a function prototype. This is the function prototype for a DLL function called EXAMPLEONCREATE, a function which receives a pointer to an IEasyLanguageObject as its sole argument, and returns an integer:

```
int __stdcall EXAMPLEONCREATE( IEasyLanguageObject * pELObj ) ;
```

5.) Tracking Analysis Techniques

In place of the legacy Dll_Add and Dll_Free functions, EasyLanguage has added support for “events”. Currently, two events are supported: “OnCreate” and “OnDestroy”. The OnCreate event occurs when an analysis technique is first inserted into a chart, any time a chart reload occurs, or when the status of the analysis technique is cycled from OFF to ON. You may create functions in your DLL that can be executed when these events occur.

Any DLL function that should be executed when an analysis technique that uses a DLL first loads should be referenced in an OnCreate statement. Similarly, any DLL function that should be executed when an analysis technique that uses a DLL is disabled should be referenced in an OnDestroy statement.

By providing a means of executing DLL functions when an analysis technique first loads and when it unloads, the OnCreate and OnDestroy events give programmers a method of tracking their analysis techniques, initializing variables, and performing housekeeping when analysis techniques that use DLL functions load or unload.

DLL functions to be executed when OnCreate or OnDestroy events occur must receive one, and only one, argument from the calling EasyLanguage analysis technique. That argument must be a pointer to the IEasyLanguageObject interface for the calling analysis technique. (As will be shown below, this argument may be passed explicitly, but will be passed automatically by EasyLanguage even if not explicitly passed in the EasyLanguage call to the DLL function.) Also, the DLL function(s) to be executed when the OnCreate and OnDestroy events occur must return an integer.

If an OnCreate or OnDestroy call is used in a strategy, the OnCreate and/or OnDestroy DLL functions will be called twice. Strategies behave a bit differently in this respect than other studies. When a strategy is applied to a chart, a "strategy group" is created internally in order to accommodate the possibility that multiple strategies, which interact with each other, will be inserted into the chart. Because a strategy group is recognized internally as a module by EasyLanguage, the OnCreate and OnDestroy events occur twice, once for the strategy itself and once for the strategy group. This is what prompts the second call to the OnCreate and OnDestroy functions when the strategy is activated (OnCreate) or deactivated (OnDestroy).

Here is some EasyLanguage code that demonstrates that the self pointer need not be explicitly passed to the DLL:

```
external:  "Sample.dll", int, "EXAMPLEONCREATE", IEasyLanguageObject ;

external:  "Sample.dll", int, "EXAMPLEONDESTROY", IEasyLanguageObject ;

#events
        OnCreate = EXAMPLEONCREATE ;
        OnDestroy = EXAMPLEONDESTROY ;
        { Note:  self pointer not explicitly passed in these two function
          calls }
#end ;
```

Here are the prototypes for the DLL functions that will be called when the OnCreate and OnDestroy events occur:

```
int __stdcall EXAMPLEONCREATE(IEasyLanguageObject * pEL);
int __stdcall EXAMPLEONDESTROY(IEasyLanguageObject * pEL);
```

If multiple DLL functions are to be run when the OnCreate or OnDestroy events occur, each DLL function prior to the last one to be run must return a value of zero (or FALSE) to indicate normal completion. If a non-zero value is returned, then subsequent functions to be run during the event will not be called.

To call multiple DLL functions when an event occurs, either of the following syntaxes may be used:

A.) More than one function may be listed in each OnCreate and OnDestroy statement. Functions will be executed in the order listed. Here is some example code:

```
#Events
        OnCreate = MyFunc1, MyFunc2, MyFunc3;
        OnDestroy = MyFunc4, MyFunc5, MyFunc6;
#End
```

B.) Multiple OnCreate and/or OnDestroy statements may be used. Functions will be executed in the order listed. Here is some example code:

```
#Events
        OnCreate = MyFunc1;
        OnCreate = MyFunc2;
        OnDestroy = MyFunc4;
        OnDestroy = MyFunc5;
#End
```

This completes the section of this reference devoted to discussion of using the SDK. We have discussed the five basic requirements of TradeStation-compatible DLL's that use the SDK. To review, those requirements are:

1.) Importation of tskit.dll
2.) Declaration of DLL Functions
3.) IEasyLanguageObject Pointer
4.) Standard C Calling Convention
5.) Tracking Analysis Techniques

Below, the contents of the TSKit type library will be discussed in detail.

Type Library Reference

1.) Available Data Types

The following data types are supported and available for use with the EasyLanguage Extension SDK Library:

In EasyLanguage use:	In your C++ DLL use:
IEasyLanguageObject	IEasyLanguageObject*
	ITradeStationPlatform*
Double	double
Float	float
Int	int
Int64	__int64
String or LPSTR	LPSTR or char*

While strings may be passed to and from TradeStation-compatible DLL's, no attempt should be made, inside a TradeStation-compatible DLL, to change the length of a string that is an EasyLanguage variable. However, the characters of such strings may be sorted or otherwise reorganized, as long as the overall string length remains unchanged by the DLL.

2.) Legacy Compatibility Notice

The following Data Type keywords, available for use with the EasyLanguage DLL Extension Kit for TradeStation 2000i and TradeStation 6, were previously provided for future use. Please note that these words should no longer be used with either the legacy SDK (elkit32.dll) nor the current EasyLanguage Extension SDK (tskit.dll). This is due to the fact that the legacy platforms did not fully support their use, and the EasyLanguage Extension SDK no longer requires them:

LPBYTE BYTE
LPDOUBLE DOUBLE
LPWORD WORD

3.) Enumerated Data Types

The following Enumerated Data Types are declared in the EasyLanguage Extension SDK Library and are available for use in custom user DLLs:

A.) EN_DATA_STREAM

This Enumerated data type may be used to identify a specific data stream where required. The defined elements for this data type are as follows:

Identifier	Value
dataDefault	255
data1	0
data2	1
data3	2
data4	3
data5	4
data6	5
data7	6
data8	7
data9	8
data10	9
data11	10
data12	11
data13	12
data14	13
data15	14
data16	15

Identifier	Value
data17	16
data18	17
data19	18
data20	19
data21	20
data22	21
data23	22
data24	23
data25	24
data26	25
data27	26
data28	27
data29	28
data30	29
data31	30
data32	31
data33	32

Identifier	Value
data34	33
data35	34
data36	35
data37	36
data38	37
data39	38
data40	39
data41	40
data42	41
data43	42
data44	43
data45	44
data46	45
data47	46
data48	47
data49	48
data50	49

The element “dataDefault” refers to the data stream for which the calling analysis technique is applied. All others refer to the specific data stream requested by the identifier.

B.) enDataType

This enumerated data type is used to identify the native data type of the object specified. The return value of the DataType property for some of the interface objects described below will return a value in this data type’s range. The defined elements for this data type are as follows:

Identifier	Value
dtUnknown	1
dtBoolean	2
dtTrueFalse	2
dtString	3
dtInteger	4
dtInt64	5
dtFloat	6
dtDouble	7
dtPointer	11

C.) enPlatformType

This enumerated data type has been implemented for future use. It’s purpose is to identify the platform type of the application running the analysis, specifically from the PlatformType property of the ITradeStationPlatform interface described below. The defined elements for this type are as follows:

Identifier	Value

Identifier	Value
ptUnknown	-1
ptDesktop	0
ptServer	1

Until this is implemented, developers can expect the PlatformType property of the ITradeStationPlatform interface to always return ptDesktop.

4.) Structures

A.) TSRuntimeErrorItem

This defined data structure is used when registering a custom runtime error item in the user DLL. It is composed of the following items:

sCompany – A BSTR variable which should contain the name of the developer/company providing the DLL.

sErrorCategory – A BSTR variable (**undefined at this time**).

sErrorLocation – A BSTR variable (**undefined at this time**).

sSourceString – A BSTR variable (**undefined at this time**).

sLongString – A BSTR variable which stores the string that appears in the details section of the TradeStation Events Window, when the specific TSRuntimeErrorItem is highlighted in the Event Window's main list.

sShortString – A BSTR variable which stores the string that appears in the TradeStation Events Window. This should be summary of the error encountered.

nParameters – An integer variable (**undefined at this time**).

nErrorCode – An integer variable (**undefined at this time**).

nErrorID – An integer variable (**undefined at this time**).

5.) Tables of Interface Properties

The following interfaces are supported and available for use with the EasyLanguage Extension SDK Library:

IEasyLanguageObject
IEasyLanguageVariable
IEasyLanguagePrice
IEasyLanguageDateTime
ITradeStationPlatform
IEasyLanguageErrors
IELFrameworkArray
IEasyLanguageSystem
IEasyLanguageProperties

IEasyLanguageServerField

The following interfaces are for internal use only. These interfaces are not intended for direct access from your DLL:

IEasyLanguageDataElement
IEasyLanguageVector
IEasyLanguageReadOnlyVector
IEasyLanguageEvent

A.) IEasyLanguageObject - Property Reference

Property Name	Description	Syntax	Notes
Close	Returns Closing price for the bar requested as a double precision decimal value.	*InterfaceObject.**Close[**nBarsBack**]***	1
CloseMD	Returns an IEasyLanguagePrice interface object, which contains the close for the data stream requested.	*InterfaceObject.**CloseMD[**enDataStream**]***	2, 3
CurrentBar	Returns CurrentBar value associated with the parent analysis technique to the interface object, as an integer.	*InterfaceObject.**CurrentBar[**enDataStream**]***	3
DataStream	Returns the data stream of the interface object as an EN_DATA_STREAM enumerated value (see the Data Type Reference).	*InterfaceObject.**DataStream***	
DateTime	Returns the DateTime value of the interface object as a double precision decimal value.	*InterfaceObject.**DateTime[**nBarsBack**]***	1, 4
DateTimeMD	Returns an IEasyLanguageDateTime interface object, which contains the date-time information for the data stream requested.	*InterfaceObject.**DateTimeMD[**enDataStream**]***	2, 3
DownTicksMD	Returns an IEasyLanguagePrice interface object, which contains the downticks for the data stream requested.	*InterfaceObject.**DownTicksMD[**enDataStream**]***	2, 3
GetServerField	Returns an IEasyLanguageServerField interface object, which contains the applicable data for the server field requested.	*InterfaceObject.**GetServerField[**szFieldName**][**enDataStream**]***	2, 3, 5
High	Returns the high for the bar requested as a double precision decimal value.	*InterfaceObject.**High[**nBarsBack**]***	1
HighMD	Returns an IEasyLanguagePrice interface object, which contains the high for the data stream requested.	*InterfaceObject.**HighMD[**enDataStream**]***	2, 3
Low	Returns the low for the bar requested as a double precision decimal value.	*InterfaceObject.**Low[**nBarsBack**]***	1
LowMD	Returns an IEasyLanguagePrice interface object, which contains the low for the data stream requested.	*InterfaceObject.**LowMD[**enDataStream**]***	3
MaxBarsBack	Returns the MaxBarsBack value of the parent analysis technique to the interface object, as an integer.	*InterfaceObject.**MaxBarsBack***	
Open	Returns the opening price for the bar requested as a double precision decimal value.	*InterfaceObject.**Open[**nBarsBack**]***	1
OpenInt	Returns the open interest for the bar requested as an integer value.	*InterfaceObject.**OpenInt[**nBarsBack**]***	1
OpenIntMD	Returns an IEasyLanguagePrice interface object, which contains the open interest for the data stream requested.	*InterfaceObject.**OpenIntMD[**enDataStream**]***	2, 3
OpenMD	Returns an IEasyLanguagePrice interface object, which contains the open for the data stream requested.	*InterfaceObject.**OpenMD[**enDataStream**]***	2, 3
Tag	This property may be used to store or retrieve any extra data for a	*InterfaceObject.**Tag (** = nValue)*	6

A.) IEasyLanguageObject - Property Reference

Property Name	Description	Syntax	Notes
	particular need. Example: Store pointer to customized data type objects.		
UpTicksMD	Returns an IEasyLanguagePrice interface object, which contains the upticks for the data stream requested.	*InterfaceObject.**UpTicksMD[**enDataStream**]***	2, 3
Variables	Returns an IEasyLanguageVariable interface object, which contains data for the variable requested.	*InterfaceObject.**Variables[**nVar | nVarName**]***	2, 7
VariablesCount	Returns the number of variables declared in the parent analysis technique to the interface object, as an integer.	*InterfaceObject.**VariablesCount***	
Volume	Returns the volume for the bar requested as an integer value.	*InterfaceObject.**Volume[**nBarsBack**]***	1
VolumeMD	Returns an IEasyLanguagePrice interface object, which contains the volume for the data stream requested.	*InterfaceObject.**VolumeMD[**enDataStream**]***	2, 3

IEasyLanguageObject Property Table Notes:

1.) nBarsBack - Required. An integer representing the number of bars back from the current bar for which the item is requested. This value must be greater than or equal to zero and an integer or a runtime error will result.

2.) Please refer to properties that apply to the returned interface type (IEasyLanguagePrice, IEasyLanguageDateTime, IEasyLanguageServerField, etc) for details on how to return the value in a specific format.

3.) enDataStream – Required. An EN_DATA_STREAM enumerated value representing the data stream for the close price requested. Please refer to the Data Type section for details on the EN_DATA_STREAM data type.

4.) DateTime format - The whole number portion of the result is returned in standard Windows Date format (i.e. *day.time*). The decimal portion of the result represents the amount of time that has transpired for the current day. For example, the return value for the DateTime property on the 12:00:00pm bar of 10/24/2005 is ***38649.50000***.

5.) szFieldName - Required. A string value representing the server field requested. The name requested should be sent exactly as it appears in the elf_info.txt file. Also, the data returned from this property will be the value as of the moment that EasyLanguage calculates the study and calls the function containing this property. Your EasyLanguage study will NOT update when the requested server field updates, unless the EasyLanguage code itself refers to the specified server field.

6.) nValue - Required. Stores an integer value for user storage of custom data. May be used to store pointers to user-defined objects.

7.) nVar | nVarName – One or the other is required. Do not use both. nVar is an integer value representing the enumerated value of the variable requested. If used, nVarName is not used. nVarName is a string value representing the name of the variable requested. Proper case for the variable name is not required. If nVarName is used, nVar is not used.

B.) IEasyLanguagePrice - Property Reference

Property Name	Description	Syntax	Notes
AsDateTime	Returns the value of the interface object for the bar requested as a double precision decimal value, which is compatible with the standard Windows Date/Time format.	*InterfaceObject.**AsDateTime[**nBarsBack**]***	1, 2
AsDouble	Returns the value of the interface object for the bar requested as a double precision decimal value.	*InterfaceObject.**AsDouble[**nBarsBack**]***	1
AsInteger	Returns the integer portion of the interface object's value for the bar requested.	*InterfaceObject.**AsInteger[**nBarsBack**]***	1
AsString	Returns the value of the interface object for the bar requested as a BSTR object.	*InterfaceObject.**AsString[**nBarsBack**]***	1
AsTrueFalse	Returns the value of the interface object for the bar requested as a Boolean (TrueFalse) value.	*InterfaceObject.**AsTrueFalse[**nBarsBack**]***	1, 3
BarsBack	Returns the number of bars of price data available to the analysis technique. The value is returned as an integer.	*InterfaceObject.**BarsBack***	
DataStream	Returns the data stream of the interface object as an EN_DATA_STREAM enumerated value. Please refer to the Enumerated Data Types section for more details on EN_DATA_STREAM.	*InterfaceObject.**DataStream***	
DataType	Returns the DataType of the interface object as an enumerated data type (enDataType). Please refer to the Enumerated Data Types section for more details on enDataType.	*InterfaceObject.**DataType***	
Max	Returns the largest value in the interface object's data series up to the current calculation point. The value is returned as a double precision floating point number.	*InterfaceObject.**Max***	
Min	Returns the smallest value in the interface object's data series up to the current calculation point. The value is returned as a double precision floating point number.	*InterfaceObject.**Min***	
Name	Returns the case sensitive name assigned to the interface object as declared in the EasyLanguage code. The value is returned as a BSTR.	*InterfaceObject.**Name***	
Size	Returns the native size, in bytes, of the interface object. The value is returned as an integer.	*InterfaceObject.**Size***	
Value	Returns the value stored in the interface object. The return type of this property is a VARIANT. Therefore, the actual value returned is dependent on the type of the receiving variable.	*InterfaceObject.**Value[**nBarsBack**]***	1

IEasyLanguagePrice Property Table Notes:

1.) nBarsBack - Required. An integer representing the number of bars back from the current bar for which the item is requested. This value must be greater than or equal to zero and an integer or a runtime error will result.

2.) DateTime format - The whole number portion of the result is returned in standard Windows Date format (i.e. *day.time*). The decimal portion of the result represents the amount of time that has transpired for the current day. For example, the return value for the DateTime property on the 12:00:00pm bar of 10/24/2005 is ***38649.50000***.

3.) Boolean values – Named for George Boole, these values are considered “False” if equal to zero, and “True” if non-zero.

C.) IEasyLanguageVariable - Property Reference

Property Name	Description	Syntax	Notes
AsDateTime	Returns the value of the interface object for the bar requested as a double precision decimal value, which is compatible with the standard Windows Date/Time format.	*InterfaceObject.**AsDateTime[**nBarsBack**]** (= dValue)*	1, 10
AsDouble	Returns the value of the interface object for the bar requested as a double precision decimal value.	*InterfaceObject.**AsDouble[**nBarsBack **]** (= dValue)*	1
AsInteger	Returns the integer portion of the interface object's value for the bar requested.	*InterfaceObject.**AsInteger[**nBarsBack **]** (= nValue)*	1, 3
AsInt64	Returns the integer portion (64-bit integer) of the interface object's value for the bar requested.	*InterfaceObject.**AsInt64[**nBarsBack **]** (= nValue)*	1, 3
AsString	Returns the value of the interface object for the bar requested as a BSTR object.	*InterfaceObject.**AsString[**nBarsBack **]** (= bsValue)*	1, 4
AsTrueFalse	Returns the value of the interface object for the bar requested as a Boolean (TrueFalse) value.	*InterfaceObject.**AsTrueFalse[**nBarsBack**]**	1, 5
DataStream	Returns the data stream of the interface object as an EN_DATA_STREAM enumerated value (see the Data Type Reference). Please refer to the Data Types section for more details on the enDataType type.	*InterfaceObject.**DataStream***	
DataType	Returns the DataType of the interface object as an enumerated data type (enDataType). Please refer to the Data Types section for more details on the enDataType type.	*InterfaceObject.**DataType***	
Dimensions	Returns the number of dimensions required by the interface object as an integer, if it is an array. Non-array objects will return 0.	*InterfaceObject.**Dimensions***	
DimensionSize	Returns the size of the specified dimension for the interface object. The value is returned as an integer.	*InterfaceObject.**DimensionSize[**nDimension**]***	6
IsSeries	Returns a long value indicating whether the interface object is a series object. Series objects will return 1, non-series objects will return 0.	*InterfaceObject.**IsSeries***	7
Max	Returns the largest value in the interface object's data series up to the current calculation point. The value is returned as a double precision floating point number.	*InterfaceObject.**Max***	8
Min	Returns the smallest value in the interface object's data series up to the current calculation point. The value is returned as a double precision floating point number.	*InterfaceObject.**Min***	
Name	Returns the case sensitive name assigned to the interface object as declared in the EasyLanguage code. The value is returned as a BSTR.	*InterfaceObject.**Name***	
SelectedIndex	Returns the active array element of an interface object.	*InterfaceObject.**SelectedIndex[**nDimension**]** (= nValue)*	6, 8

C.) IEasyLanguageVariable - Property Reference

Property Name	Description	Syntax	Notes
Size	Returns the native size, in bytes, of the interface object. The value is returned as an integer.	*InterfaceObject.**Size***	
Value	Returns the value stored in the interface object. The return type of this property is a VARIANT. Therefore, the actual value returned is dependent on the type of the receiving variable.	*InterfaceObject.**Value[**nBarsBack**]** (= vValue)*	1, 9

IEasyLanguageVariable Property Table Notes:

1.) nBarsBack - Required. An integer representing the number of bars back from the current bar for which the item is requested. This value must be greater than or equal to zero and an integer or a runtime error will result.

2.) dValue - Optional. A double precision floating point value to be assigned to InterfaceObject.

3.) nValue - Optional. An integer value assigned to the property being assigned. Non-integer values will be truncated to an integer. When assigned to the SelectedIndex[nDimension] property, this value must not exceed the value of the property DimensionSize for the selected dimension.

4.) bsValue - Optional. A BSTR value to be assigned to InterfaceObject.

5.) bValue – Optional. Boolean value. Named for George Boole, these values are considered “False” if equal to zero, and “True” if non-zero.

6.) nDimension – Required. Specifies the array dimension to be used. This value must not exceed the value of the property ***Dimensions*** – 1, or an array bounds runtime error will occur.

7.) IEasyLanguagePrice interface objects, like IEasyLanguageVariable interface objects, support this property. However, price data values in EasyLanguage are always series values. So they will always return 1. Therefore, this property is unnecessary for IEasyLanguagePrice interface objects. This property has been purposefully omitted from the IEasyLanguagePrice property reference, above.

8.) The SelectedIndex property must be set for every dimension of the array before any attempts to access individual array elements can be made.

9.) vValue – Optional. A variant value to be assigned to the property.

10.) DateTime format - The whole number portion of the result is returned in standard Windows Date format (i.e. *day.time*). The decimal portion of the result represents the amount of time that has transpired for the current day. For example, the return value for the DateTime property on the 12:00:00pm bar of 10/24/2005 is ***38649.50000***.

D.) IEasyLanguageDateTime - Property Reference

Property Name	Description	Syntax	Notes
AsDateTime	Returns the value of the interface object for the bar requested as a double precision decimal value, which is compatible with the standard Windows Date/Time format.	*InterfaceObject.**AsDateTime[**nBarsBack**]***	1, 2
AsString	Returns the value of the interface object for the bar requested as a BSTR object.	*InterfaceObject.**AsString[**nBarsBack **]***	1
AsDate	Returns the date portion of the DateTime value for the interface object. The result is returned as a double precision decimal value, which is compatible with the standard Windows Date/Time format.	*InterfaceObject.**AsDate[**nBarsBack**]***	1, 2
AsTime	Returns the time portion of the DateTime value for the interface object. The result is returned as a double precision decimal value, which is compatible with the standard Windows Date/Time format.	*InterfaceObject.**AsTime[**nBarsBack**]***	1, 2
Day	Returns the day of the DateTime value for the interface object. The value is returned as an integer, representing the day of the month.	*InterfaceObject.**Day[**nBarsBack**]***	1
Hour	Returns the hour of the DateTime value for the interface object. The value is returned as an integer, representing the hour of the day.	*InterfaceObject.**Hour[**nBarsBack**]***	1
Milliseconds	This property returns the milliseconds portion of the DateTime value for the interface object. The value is returned as an integer, representing the milliseconds portion of the time. This property is not yet functional when used to reference bar data. At present, any time reference in an EasyLanguage object will return 0 for seconds and milliseconds.	*InterfaceObject.**Milliseconds[**nBarsBack**]***	1
Minutes	Returns the minutes portion of the DateTime value for the interface object. The value is returned as an integer, representing the minutes portion of the time.	*InterfaceObject.**Minutes[**nBarsBack**]***	1
Month	Returns the month portion of the DateTime value for the interface object. The value is returned as an integer, representing the month portion of the date.	*InterfaceObject.**Month[**nBarsBack**]***	1
Year	Returns the year portion of the DateTime value for the interface object. The value is returned as an integer, representing the year portion of the date. The year is returned in YYYY format.	*InterfaceObject.**Year[**nBarsBack**]***	1

IEasyLanguageDateTime Property Table Notes:

1.) nBarsBack - Required. An integer representing the number of bars back from the current bar for which the item is requested. This value must be greater than or equal to zero and an integer or a runtime error will result.

2.) DateTime format - The whole number portion of the result is returned in standard Windows Date format (i.e. *day.time*). The decimal portion of the result represents the amount of time that has transpired for the current day. For example, the return value for the DateTime property on the 12:00:00pm bar of 10/24/2005 is ***38649.50000***.

E.) ITradeStationPlatform - Property Reference

Property Name	Description	Syntax	Notes
PlatformType	Not yet functional – reserved for future use. Until this is functional, developers can expect the PlatformType property of the ITradeStationPlatform interface to always return ptDesktop (zero). For additional information, see description of enPlatformType enumerated type, above.	*InterfaceObject.**PlatformType***	
CustomerID	Returns the customer number associated with the interface object. The value is returned as an integer.	*InterfaceObject.**CustomerID***	
NetworkID	Returns the network identification number associated with the interface object. The value is returned as an integer.	*InterfaceObject.**NetworkID***	

F.) IEasyLanguageErrors - Property Reference

Property Name	Description	Syntax	Notes
LastRuntimeError	Returns the last custom runtime error raised by the analysis technique. It is returned as an IEasyLanguageErrorItem object.	*InterfaceObject.**LastRuntimeError***	
RaiseRuntimeError	Allows the developer to generate a custom runtime error from within a DLL function. The parameter errID must first be predefined with a call to the RegisterError property.	*InterfaceObject.**RaiseRuntimeError(**errID**)***	1
RegisterError	This property registers a user-defined runtime error, which can be raised anywhere within the developer's code.	*InterfaceObject.**RegisterError(**errItem, errID**)***	1, 2

IEasyLanguageErrors Property Table Notes:

1.) errID - Required. An integer value which represents an error code defined by the developer for the error being generated.

2.) errItem - Required. A TSRuntimeErrorItem object representing the custom runtime error being registered to the user DLL. Please see the Data Types section for details on this data structure.

G.) IELFrameworkArray - Property Reference

Property Name	Description	Syntax	Notes
Compare	This method will compare the values between two dynamic array references, for the specified index range. The method compares the individual elements the specified number of corresponding elements one to one, beginning from the Index specified for each dynamic array.	*InterfaceObject.**Compare(**nSourceHandle, nDestHandle, nSourceIndex, nDestIndex, nCount**);***	1,2,3,4,5
Copy	This method will copy the values from the source array to the destination array. The method will copy the specified number of elements one to one, beginning from the Index specified for each dynamic array.	*InterfaceObject.**Copy(**nSourceHandle, nDestHandle, nSourceIndex, nDestIndex, nCount**);***	1,2,3,4,5
Count	This property returns the total number of elements in the specified array. The highest indexed value will be the return value minus one (i.e. If Count returns 1, the highest Index value allowable will be 0). The value returned is an Integer.	*InterfaceObject.**Count[**nHandle**];***	6
DataType	This property returns the data type of the values stored in the array, as an enumerated data type value (see **enumerated values** above).	*InterfaceObject.**DataType[**nHandle**];***	6
GetFloatValue	This property returns the floating point equivalent of the value passed to the specified array index.	*InterfaceObject.**GetFloatValue(**nHandle, nIndex**);***	6,7
GetIntegerValue	This property assigns the integer equivalent of the value passed to the specified array index.	*InterfaceObject.**GetIntegerValue(**nHandle, nIndex, nValue**);***	6,7,8
GetValue	This method returns the value of the specified index as a Variant type, for the dynamic array handle passed into it.	*InterfaceObject.**GetValue(**nHandle, nIndex**);***	6,7
IsValidHandle	This property returns a long value. It will return a non-zero value when the parameter passed is designated to a valid handle for a dynamic array, and a zero when it does not. For evaluation purposes, a call to the property may be used in logical expressions.	*InterfaceObject.**IsValidHandle(**nHandle**);***	6
Resize	This method resizes an array to the specified length. If the new size is smaller than the old one, the extra elements are truncated from the array. If the new size is larger than the existing array, the new elements are assigned the initialized value that was originally assigned to the array.	*InterfaceObject.**Resize(**nHandle, nNewSize**);***	6,9
SetFloatValue	This property assigns the floating-point value passed to the	*InterfaceObject.**SetFloatValue(**nHandle, nIndex,*	6,7,10

Property Name	Description	Syntax	Notes
	specified array index.	*fValue**)***	
SetIntegerValue	This property assigns the integer equivalent of the value passed to the specified array index.	*InterfaceObject.**SetIntegerValue(**nHandle, nIndex, nValue**)***	6,7,8
SetValue	This method sets the value of the specified index, for the dynamic array handle passed into it. The value will internally format, according to the datatype that was declared or detected for the dynamic array.	*InterfaceObject.**SetValue(**nHandle**,** nIndex, vValue**)***	6,7,11
SetValueRange	This method sets the value of the specified index range, for the dynamic array handle passed into it. The value will internally format, according to the data type that was declared or detected for the dynamic array.	*InterfaceObject.**SetValue(**nHandle**,** nBegin, nEnd, vValue**)***	6,11
Sort	This method will sort the values in the specified index range, for the dynamic array handle passed into it. The value will sort in Ascending, or Descending order, depending on the Ascending value passed to it.	*InterfaceObject.**Sort(**nHandle**,** nBegin, nEnd, bAscending**)***	6
Sum	This method returns the sum of the values within the specified data range. Values are returned as a double-precision floating point value.	*InterfaceObject.**Sum(**nHandle, nBegin, nEnd**)***	6

IELFrameworkArray Property Table Notes:

1.) nSourceHandle - Required. Represents the handle ID of the first array.

2.) nDestHandle - Required. Represents the handle ID of the second array.

3.) nSourceIndex - Required. An integer that identifies the index to begin the comparison from for the first array.

4.) nDestIndex - Required. An integer that identifies the index to begin the comparison from for the second array.

5.) nCount - Required. An Integer that identifies the number of elements to perform the comparison upon.

6.) nHandle - Required. Represents the handle ID of the requested array.

7.) nIndex – Required. An integer that identifies the array element requested.

8.) nValue – Required. The Integer value that will be assigned to the array element specified.

9.) nNewSize – Required. The An Integer representing the new size of the array.

10.) fValue – Required. The value that will be assigned to the array element specified.

11.) vValue – Required. A variant that will be used to set the new value of the specified array index.

H.) IEasyLanguageSystem - Property Reference

Property Name	Description	Syntax	Notes
Array	This property returns an IELFrameworkArray interface object for the Array Framework associated with the EasyLanguage Object.	*InterfaceObject.**Array***	

I.) IEasyLanguageProperties - Property Reference

Property Name	Description	Syntax	Notes
Items	This is an indexed property. It is a map of COM BSTR vs. a COM Variant. It can be used in a manner similar to that described above for the Tag property of the IEasyLanguageObject interface (that is, for storage of user data of any type).	*VARIANT x = Properties..**Items["MY_DATA"]**;*	1, 2
ItemsByInteger	This is an indexed property. It is a map of COM integer vs. a COM Variant. It can be used in a manner similar to that described above for the Tag property of the IEasyLanguageObject interface (that is, for storage of user data of any type).	*VARIANT x = Properties..**ItemsByInteger[MY_DATA]**;*	1, 2

IEasyLanguageProperties Property Table Notes:

1.) General: This interface is a collection of properties in the format: field = value. Field can be a string or an integer. Value is always a Variant. This property is made available for developer storage of data of any type - an integer, a double, a pointer to an object, an interface, etc. The Items property can be considered to be an expanded version of the Tag property of the IEasyLanguageObject interface, described above. The Tag property allows the developer to store a single item. However, the IEasyLanguageProperties Items property allows for developer storage of multiple items. The Items property can be used to store per-instance or per-program data.

2.) When large amounts of data are being stored, the ItemsByInteger property is preferred to the Items property for best computational speed performance. This is because ItemsByInteger can be accessed without using the slower string operations that Items requires.

J.) EasyLanguageServerField - Property Reference

Property Name	Description	Syntax	Notes
AsDateTime	This property returns the value of the interface object for the bar requested as a double precision decimal value, which is compatible with the standard Windows Date/Time format.	*InterfaceObject.**AsDateTime[**nBarsBack**];*	1
AsDouble	This property returns the value of the interface object for the bar requested as a double precision decimal value.	*InterfaceObject.**AsDouble[**nBarsBack **];*	1
AsInteger	This property returns the integer portion of the interface object's value for the bar requested.	*InterfaceObject.**AsInteger[**nBarsBack **];*	1
AsString	This property returns the value of the interface object for the bar requested as a BSTR object.	*InterfaceObject.**AsString[**nBarsBack **];*	1
AsTrueFalse	This property returns the value of the interface object for the bar requested as a TrueFalse, or boolean value. NOTE: The property will return "False" for values equal to zero, and "True" for all non-zero values.	*InterfaceObject.**AsTrueFalse[**nBarsBack**];*	1
DataStream	This property returns the DataStream of the interface object as an enumerated data stream type (EN_DATA_STREAM). Please refer to the Data Types section for more details on the EN_DATA_STREAM type.	*InterfaceObject.**DataStream;*	
DataType	This property returns the data type of the interface object as an enumerated data type (enDataType). Please refer to the Data Type section for more details on the enDataType type.	*InterfaceObject.**DataType;*	
IsSeries	This property returns a long value indicating whether the interface object is a series object. Series objects will return 1, non-series objects will return 0. NOTE: IEasyLanguagePrice interface objects support this property, however, price data values in EasyLanguage are always series values. They will always return 1; therefore, this property is unnecessary for IEasyLanguagePrice interface objects. This property has been purposefully omitted from the IEasyLanguagePrice property reference.	*InterfaceObject.**IsSeries;*	
Max	This property returns the largest value in the interface object's data series up to the current calculation point. The value is returned as a double precision floating point number.	*InterfaceObject.**Max;*	

Property Name	Description	Syntax	Notes
Min	This property returns the smallest value in the interface object's data series up to the current calculation point. The value is returned as a double precision floating point number.	*InterfaceObject.**Min;***	
Name	This property returns the case sensitive name assigned to the interface object as declared in the EasyLanguage code. The value is returned as a BSTR.	*InterfaceObject.**Name;***	
Size	This property returns the native size, in bytes, of the interface object. The value is returned as an integer.	*InterfaceObject.**Size;***	
Value	This property returns the value stored in the interface object. NOTE: The return type of this property is a VARIANT; therefore, the actual value returned is dependant on the type of the receiving variable.	*InterfaceObject.**Value[**nBarsBack**];*	1

IEasyLanguageServerField Property Table Notes:

1.) nBarsBack - Required. An integer representing the number of bars back from the current bar for which the item is requested. **NOTE:** This value must be a positive integer or a runtime error will result. Please see technical notes for details

Demonstration DLL Code

```
#import "C:\Program Files\TradeStation 8.1 (Build 3006)\Program\tskit.dll" no_namespace
#include <atlbase.h>


//////////////////////////////////////////////////////////////
// Calculate simple moving average value

double __stdcall MovAvg
( IEasyLanguageObject * pELObj, int iAvgLength )
{
	double dMovAvg ;
	dMovAvg = 0 ;

	// verify that sufficient bars have passed before back-referencing historical prices
	if( pELObj->CloseMD[data1]->BarsBack > iAvgLength && iAvgLength > 0 )
	{
		double dSum = 0.0 ;
		for (int i = 0; i < iAvgLength; i++)
		{
			dSum += pELObj->CloseMD[data1]->AsDouble[ i ] ;
		}
		dMovAvg = dSum / iAvgLength ;
	}
	return dMovAvg ;
}

//////////////////////////////////////////////////////////////
//  Generate a run-time error in TradeStation
//  This function is called internally by other DLL functions when an error in TradeStation
//  is to be produced.

void fnGenRunTimeError
( IEasyLanguageObject * pEL, int iErrorNum )
```

```
{
	TSRuntimeErrorItem tsItem;
	int m_HistErr ;
	tsItem.sCompany = _bstr_t("TradeStation Securities, Inc.").copy();
	tsItem.sErrorLocation = _bstr_t("Example Code Library").copy();
	tsItem.sErrorCategory = _bstr_t("Error").copy();
	tsItem.sLongString = NULL;
	tsItem.nParameters = 0;

   switch ( iErrorNum )
	{
	case 1:
		{
			// Error 1
				tsItem.sShortString = _bstr_t("Error – Description of Error 1 goes here.").copy();
			tsItem.sSourceString = _bstr_t("Additional detailed description goes here.").copy();
			tsItem.nErrorCode = iErrorNum ;
		}
		break ;
		case 2:
		{
			// Error 2
			tsItem.sShortString = _bstr_t("Error – Description of Error 2 goes here.").copy();
			tsItem.sSourceString = _bstr_t("Additional detailed description goes here.").copy();
			tsItem.nErrorCode = iErrorNum ;
		}
		break ;
		default:
		{
			// Generate a run-time error of undefined type or origin
			tsItem.sShortString = _bstr_t("Undefined error in "
				"MyDLL.dll.").copy();
			tsItem.sSourceString = _bstr_t("Error origin undefined.").copy();
			tsItem.nErrorCode = 9999 ;
		}
	}
```

```
        m_HistErr = pEL->Errors->RegisterError( &tsItem ) ;
        pEL->Errors->RaiseRuntimeError( m_HistErr ) ;
}


//////////////////////////////////////////////////////////////
// Server field call
double __stdcall GETSERVERFIELD
(IEasyLanguageObject *pEL, LPSTR szFName)
{
        IEasyLanguageServerField *iSF = pEL->ServerField[_bstr_t(szFName)][dataDefault];
        return iSF->AsDouble[0];
}


//////////////////////////////////////////////////////////////
// Sort an EasyLanguage array using this DLL function

void __stdcall MYARRAYSORT
( IEasyLanguageObject* pELObj, char* MyArrayName )
{
        int nOuter, nInner ;
        IEasyLanguageVariable* pELVar = pELObj->Variables[ MyArrayName ] ;
        if ( pELVar->Dimensions == 1 )
        {
                int nSize = pELVar->DimensionSize[0] ;
                for ( nOuter = 0; nOuter < nSize - 1; nOuter++ )
                {
                        for ( nInner = nOuter + 1; nInner < nSize; nInner++ )
                        {
                                pELVar->SelectedIndex[0] = nInner ;
                                double dSecondValue = pELVar->Value[0] ;
                                pELVar->SelectedIndex[0] = nOuter ;
                                double dFirstValue = pELVar->Value[0] ;
                                if ( dSecondValue < dFirstValue )
                                {
                                        double dTmp = dFirstValue ;
```

```
                                        pELVar->Value[0] = dSecondValue ;
                                        pELVar->SelectedIndex[0] = nInner ;
                                        pELVar->Value[0] = dTmp ;
                                }
                        }
                }
        }
}


//////////////////////////////////////////////////////////////
// Fill dynamic EasyLanguage array

void __stdcall FILLDYNARRAY
(IEasyLanguageObject * pEL, LPSTR PassedVar)
{
        IEasyLanguageVariablePtr pMyVar = NULL;
        IELFrameworkArrayPtr pDynArray = NULL;

        IEasyLanguageDateTime * pMyDateTime = pEL->DateTimeMD[ dataDefault ] ;
        double myDT = pMyDateTime->AsDateTime [0] ;

        long nTotVars = pEL->VariablesCount;
        for (long n = 0; n < nTotVars; n++)
        {
                pMyVar = pEL->Variables[n];
                _bstr_t mybstr = pMyVar->Name;
                if (pMyVar->DataType == dtHandle)
                {
                        int nValue = pMyVar->GetAsInteger(0);
                        CComVariant xx(pEL->Close[0]);
                        if (pEL->System->Array->IsValidHandle( (int) nValue ) && (mybstr == (_bstr_t) PassedVar))
                        {
                                pDynArray = pEL->System->Array;
                                pDynArray->SetValue(nValue, 0, CComVariant(xx));
                        }
                }
```

```
        }
}


//////////////////////////////////////////////////////////////////
// Sum upticks and downticks from intraday chart to get volume in DLL – store in global DLL variable

double dMyDLLDouble = 0; //global variable

int __stdcall SetdMyDLLDouble
( IEasyLanguageObject * ELObjP )
{
        dMyDLLDouble = ELObjP->UpTicksMD[dataDefault]->AsDouble[0]
                + ELObjP->DownTicksMD[dataDefault]->AsDouble[0] ;
        return 1;
}


//////////////////////////////////////////////////////////////////

// Place a 64-bit integer value into an EasyLanguage variable whose data type is 64-bit integerint __stdcall Int64Example( IEasyLanguageObject*
pEL, __int64 Int64Value, LPSTR VarName  )
{

        pEL->Variables[VarName]->AsInt64[0] = Int64Value ;

        return 1 ;
}
```

Support

For support with the use of the EasyLanguage Extension SDK, please visit the Support Forum at TradeStation.com. In the TradeStation & EasyLanguage Support area of the Support Forum there is a support area dedicated to the use of TradeStation-compatible DLL's in conjunction with EasyLanguage. Additional code examples are available in this area of the Support Forum.

Here is a direct link to the EasyLanguage-DLL category of the Support Forum:

https://community.tradestation.com/discussions/forum.aspx?forum_id=213&selcategory=1853&subcategory=easylanguage-dll

If you have TradeStation-related DLL questions, please create a new topic in the above-linked area of the Support Forum. Click on the link labeled "New Topic" at the top of that web page. This will take you to the "New Topic" form. When completing the form, choose as the "Category" for your topic "EasyLanguage". As the sub-category for your question, choose "EasyLanguage-DLL."