All Data Everywhere

Introduction

The "All Data Everywhere" library (or ADE for short) is a set of EasyLanguage functions and indicators that allow you to store any kind of data for any symbol and bar interval. You can then access that data from any study or strategy, regardless of the symbol or bar interval.

One powerful use for ADE is to store higher timeframe data for a symbol and then access that data from a lower timeframe. For example, you can calculate and store ADX and RSI for a 30 minute chart, and then you can access that data from a 5 minute chart.

Another powerful use for ADE is to look at data for other symbols. You can use ADE to store data (OHLC, volume, indicators, etc) for an entire portfolio of symbols, and then you can access the data for any symbol from any other symbol. This makes it possible to perform analyses that depend on the relationships between different symbols in your portfolio.

ADE includes the ability to save data to text files and load it back. This means that you can pre-calculate and store data for any symbols and timeframes you want, and you can retrieve that data whenever you want. For example, you can store five years of data for a 30 minute MSFT chart. If you open a 30 minute MSFT chart with only one month of data, your study or strategy can load the five years of historical data and append only the new data. It is not necessary to recalculate the entire five years every time.

ADE uses the ELCollections library, so you must install ELCollections along with ADE. Also, taking the time to learn how ELCollections works will allow you to make the most of ADE.

Examples

Before we dive into the details of how ADE works, here are a few examples to show you what it can do.

The following simple indicator stores the Open, High, Low, Close, and Volume for the current symbol and bar interval. It will include the data for every bar on the chart.

```
Vars:
   Vol(0);

// Calculate correct total volume regardless of bar type
Vol = IFF(BarType < 2, UpTicks + DownTicks, Volume);

Value1 = ADE.PutOHLCV(GetSymbolName, ADE.BarInterval, ADE.BarID,
   Open, High, Low, Close, Vol);
```

The ADE.PutOHLCV function stores the OHLCV data for the specified symbol, bar interval, and bar identifier. The indicator just uses GetSymbolName to pass the current symbol. It calls the ADE.BarInterval function to pass a bar interval that works with daily, monthly, and weekly charts as well as intraday charts. The ADE.BarInterval function returns 0 for daily charts, -1 for

weekly charts, and -2 for monthly charts. For intraday charts, it returns the number of minutes per bar, just like the BarInterval reserved word.

The indicator calls the ADE.BarID function to pass a standardized bar identifier that specifies the date and time of the bar. The Bar ID is calculated as Date + Time / 10000.

Since the indicator runs on every bar, it will store the OHLCV data for every bar. If you apply this indicator to a 30 minute chart for MSFT, it will automatically store the OHLCV values for all the bars in the chart.

To retrieve these stored values, you can use the following indicator:

```
Inputs:
   Interval(30);

Vars:
   MyOpen(0),
   MyHigh(0),
   MyLow(0),
   MyClose(0),
   MyVolume(0);

Value1 = ADE.GetOHLCV(GetSymbolName, Interval, ADE.BarID,
   MyOpen, MyHigh, MyLow, MyClose, MyVolume);

Plot1(MyOpen, "MyOpen");
Plot2(MyHigh, "MyHigh");
Plot3(MyLow, "MyLow");
Plot4(MyClose, "MyClose");
```

This indicator uses the ADE.GetOHLCV function to retrieve the stored values for the current symbol and for the bar interval specified by the Interval input. If you now apply this indicator to a 5 minute chart for MSFT, it will plot the 30 minute OHLC values that correspond to each 5 minute bar. It uses the 30 minute interval because that is what the Interval input specifies.

The ADE library includes more sophisticated versions of these indicators called “ADE Save OHLCV” and “ADE Plot OHLCV”. These indicators are similar to those shown above, but they provide the option to save the stored data to a text file and read it back.

Now suppose that you would like to calculate some metrics in one chart and store them for retrieval in another chart. The ADE.PutBarInfo and ADE.GetBarInfo functions allow you to store and retrieve arbitrary data. Here is an example indicator that calculates and stores ADX and RSI:

```
Vars:
   Class(“ADX&RSI”),   // identifies our metrics with a unique name
   InfoMap(MapSN.New); // used to pass data to ADE

// Put the information we want to store in our InfoMap
Value1 = MapSN.Put(InfoMap, "ADX", ADX(14));
Value1 = MapSN.Put(InfoMap, "RSI", RSI(Close, 14));
```

```
// Tell ADE to store this info for the current symbol and bar interval
Value1 = ADE.PutBarInfo(Class, GetSymbolName, ADE.BarInterval,
   ADE.BarID, InfoMap);
```

The first thing this indicator does is to define an ADE data class. An ADE "class" is simply a name that identifies one or more data series that will be stored together. These data series are called the "members" of the class. Classes provide a way to organize your data series into logical groupings. In this case, the indicator uses a class called "ADX&RSI" to identify the metrics we want to store.

Next, this indicator creates a local map (InfoMap) which it uses to pass our information to ADE. The indicator adds the desired data to the map, and then it passes the map to the ADE.PutBarInfo function. This function reads the information from the InfoMap and stores it for the specified class, symbol, bar interval, and Bar ID. Note that the function does not actually store the InfoMap itself. The map is simply a convenient way to pass arbitrary information to the function. You can use an InfoMap to pass two metrics or twenty metrics with equal ease.

Now you can retrieve these metrics in another chart with the following indicator:

```
Inputs:
   Interval(30);

Vars:
   Class("ADX&RSI"),   // identifies our metrics with a unique name
   InfoMap(MapSN.New), // used to retrieve data from ADE
   MyADX(0), MyRSI(0);

// Retrieve the info for the current symbol and bar interval into InfoMap
Value1 = ADE.GetBarInfo(Class, GetSymbolName, Interval,
   ADE.BarID, InfoMap);

// Fetch the values from the InfoMap into variables
MyADX = MapSN.Get(InfoMap, "ADX");
MyRSI = MapSN.Get(InfoMap, "RSI");

// Plot them
Plot1(MyADX, "ADX");
Plot2(MyRSI, "RSI");
```

This indicator calls ADE.GetBarInfo to copy the data for the specified symbol, bar interval, and Bar ID into the local InfoMap. You can then retrieve the metrics from the InfoMap and plot them.

Suppose you apply the "Put" indicator to a 30 minute MSFT chart and the "Get" indicator to a 5 minute MSFT chart. The 5 minute chart will now display the 30 minute ADX and RSI.

The ADE library includes more sophisticated versions of these indicators called "ADE Save Info Test" and "ADE Plot Info Test". These indicators are similar to those shown above, but they include additional metrics, and they also provide the option to save the stored data to a text file

and read it back. You can use these indicators as templates to build your own studies and strategies.

Important Note: The names "Date" and "Time" are reserved by ADE, so you cannot use them for class members. When ADE saves data to a text file, it identifies the bars with Date and Time columns instead of Bar IDs. This makes the file much easier to read. When ADE reads a file, it converts the dates and times back to Bar IDs. ADE needs to reserve the names "Date" and "Time" in order to perform the conversions from Bar IDs to dates and times and back again.

If you try to use the names "Date" or "Time" in a class, ADE will raise a runtime error.

Concepts

This section describes the underlying data structures used by ADE. It is possible to use the ADE functions and indicators without knowing about these data structures, but understanding the concepts behind ADE will allow you to take full advantage of all its capabilities. The following discussion assumes that you are already familiar with the ELCollections library, and that you have a basic knowledge of maps, lists, shared collections, and collections of collections.

ADE allows you to group different kinds of data together by class. For example, you might use a class called "TrendInfo" to store various trend indicators, and you might use a class called "Oscillators" to store various oscillators. The name "OHLCV" is reserved by ADE for a class that holds Open, High, Low, Close, and Volume data.

For each class, ADE allows you to store data for any symbol and bar interval that you want. For example, the "TrendInfo" class might contain data for MSFT at 5 min, MSFT at 15 min, CSCO at 5 min, CSCO at 15 min, etc.

ADE creates a shared map for each class of data. The suffix ":ADE" is appended to the class name to prevent conflicts with your own map names or with other libraries.

The data for a particular symbol and bar interval is stored in a sub-map, which in ADE is called a *DataMap*. A DataMap contains the following elements:

- A list of numeric Bar IDs (calculated as Date + Time / 10000). Each Bar ID identifies a bar by its date and time. Thus, any two bars that close at the same date and time will have the same Bar ID (even if they have different bar intervals).
- A list of numbers for each kind of data that you add to the map.
- A BarMap that associates each Bar ID with its index in the list. This allows ADE to find the index for any Bar ID so that it can look up the corresponding data in any of the lists.

For example, the "OHLCV" DataMap for MSFT at 5 minute intervals might look like this:

```
:Bar -> (1041112.0635, 1041112.0640, 1041112.0645)
:BarMap -> (1041112.0635 -> 1, 1041112.0640 -> 2, 1041112.0645 -> 3)
Open -> (27.16, 27.08, 27.03)
```

```
High -> (27.20, 27.08, 27.04)
Low -> (27.05, 26.96, 26.95)
Close -> (27.08, 27.03, 26.99)
Volume -> (5308743, 4145896, 2776433)
```

Notice that the ":Bar" and ":BarMap" keys each start with a colon. This indicates that these are special keys which are predefined by ADE.

The other elements in the DataMap above are just the data series for Open, High, Low, Close, and Volume. Each value in one of these series corresponds to the Bar ID at the same position in the ":Bar" series.

Remember that each DataMap stores data for a single symbol and bar interval. ADE stores all the DataMaps for a given class in a shared MasterMap. Suppose you create "OHLCV" DataMaps for CSCO and MSFT at 15 minute and 60 minute bar intervals. The "OHLCV" MasterMap would then have the following structure:

```
CSCO ->
   15 -> DataMap
   60 -> DataMap
MSFT ->
   15 -> DataMap
   60 -> DataMap
```

Whenever you request a DataMap for a symbol and bar interval that are not in the MasterMap, ADE will automatically create the DataMap and add it to the MasterMap. For example, suppose you request a DataMap for CSCO at 30 minute intervals. The MasterMap will now look like this:

```
CSCO ->
   15 -> DataMap
   30 -> DataMap
   60 -> DataMap
MSFT ->
   15 -> DataMap
   60 -> DataMap
```

Notice that it is perfectly acceptable to have different bar intervals for different symbols. The MasterMap structure is very flexible, and it allows ADE to quickly find the DataMap for any symbol and bar interval.

Creating Your Own Class Functions

You can use the ADE library as-is to store and retrieve any data you want by class, symbol, and bar interval, since the ADE.PutBarInfo and ADE.GetBarInfo functions give you complete flexibility over the information you store and retrieve. On the other hand, these functions are not as efficient as the ADE.GetOHLCV and ADE.PutOHLCV functions. If you frequently store and retrieve certain metrics, and if you want to improve the efficiency of these operations, you can create your own specific Get and Put functions. ADE provides a special indicator called "ADE Generate Class" to assist you in this process. You don't have to write the code yourself: you can just create a simple class definition and ADE will generate the code for you.

In order to keep user-defined class functions and indicators separate from the core ADE library, the class generator uses the "ADC" prefix (which stands for All-Data-Class). The generator creates functions and indicators for your class. The functions will be called ADC.Get*Class* and ADC.Put*Class*, where *Class* is the name of your class. The indicators will be called "ADC Save *Class*" and "ADC Plot *Class*".

You can use the generated "Save" indicator to store data for your class (just like you can with "ADE Save OHLCV"). However, before you use the indicator, you will need to modify it to include the calculations for the data you wish to store.

Once you have stored data for the class, you can use the generated "Plot" indicator to plot the data on another chart.

Defining the Class

An ADE class is just a set of data series that you want to group together for efficient access and storage. To define a class, perform the following steps:

1. Open a text editor such as Notepad.

2. Type the names of the data series that you wish to include in the class. Each name should appear on a separate line. For example:

```
ADX
DMIPlus
DMIMinus
```

3. If you wish, you may also define *parameters* for the class. This allows you to calculate and store class data for different input values (e.g. the length used to calculated ADX). For more information, see "Parameterized Classes" below. This is a more advanced subject, so it's fine to skip that section and come back to it once you are familiar with the class generation process.

4. Save the text file in the C:\ADE\Classes directory as ClassName.txt, where "ClassName" is the name of the class you want to create. For example, you might save the list above in a file called TrendInfo.txt. (If you are not using C:\ADE as your ADE root directory, substitute the appropriate directory. Just make sure the file is saved in the Classes subdirectory.)

Parameterized Classes

Many of the calculations that you want to store in an ADE class will depend on various input values. You may wish to store class data only for your preferred inputs, in which case a simple class works well. However, if you want to store and access data for different inputs, you will need to create a *parameterized class*. This kind of class identifies the DataMap not only by the class, symbol, and interval, but also by the parameters (inputs) used to generate the data.

Here's how it works. When you define a parameterized class, ADE generates a PD (ParamDesc) function that accepts your parameters and creates a string description of them. For example, suppose you create a "Stochastics" class, and you specify three parameters: the stochastics length (StochLen) and the two smoothing lengths (Length1, Length2). The class generator will create a function called ADC.StochasticsPD which accepts these three parameters as arguments. If you pass 14, 3, 3, the function will return "#14,3,3". If you pass 20, 4, 3, the function will return "#20,4,3". Thus, the function returns a string that uniquely identifies a particular set of parameter values.

The result of the PD function can be stored in a variable called ParamDesc. (It's best to do this only on the first bar, since string operations in TradeStation can be quite slow.) This variable should then be passed to the ADC.Get*Class* and ADC.Put*Class* functions. The ParamDesc will always be the argument following BarID, and it is only present if the class if parameterized. Furthermore, if you call the ADE.OpenMap or ADE.SaveMap functions, you should always append the ParamDesc to the class name, like this:

```
Value1 = ADE.OpenMap(Class + ParamDesc, GetSymbolName, Interval);
```

The generated indicators do all of this for you. In addition, they include the specified parameters and their default values in the Inputs section of the indicator. It is worth studying the generated indicators closely to see the correct usage for parameterized classes.

To create a parameterized class, add the keyword #PARAMS to your class file, followed by the parameter names with their default values. The keyword and parameter names must each appear on a separate line, and they must follow the member names. You may add a blank line between the member names and the #PARAMS keyword. Note that #PARAMS must be in all-caps.

For example, here is how you would define the "Stochastics" class discussed above:

```
FastK
FastD
SlowK
SlowD

#PARAMS
StochLen(14)
Length1(3)
Length2(3)
```

The values you specify for the parameters will be used as the default values for the inputs in the indicators. You may then change the defaults by formatting the indicators.

The parameter values perform another important function: they tell ADE how many decimals to include in a ParamDesc. The number of decimals that you use in the class file will be used whenever a ParamDesc is generated by the PD function. **It is very important for a parameter definition to include the maximum number of significant decimals that you will use for the class.** Otherwise, the decimal values for a parameter value may not be included in the generated description, which would make the ParamDesc inaccurate.

For example, suppose you decide to define two parameters called LowThreshold and HighThreshold. The defaults for these are 2 and 8, but you might specify up to two decimals for the parameters. You should define these parameters as follows in the class file:

```
#PARAMS
LowThreshold(2.00)
HighThreshold(8.00)
```

Since you have included two decimals in the parameter definitions, the PD function will always include two decimals as well. Thus, if you pass 3.25 and 7 to this function, it will return "#3.25,7.00".

Creating the Functions and Indicator

When you define a class, ADE will generate the code for you, but you will need to create the EasyLanguage functions and indicator yourself. This just involves copying the generated code into the EasyLanguage documents. Here are the steps you need to perform:

1. Open a Chart window. The symbol that appears in the window is not important.

2. Insert the "ADE Generate Class" indicator into the chart.

3. Format the indicator. For the Class input, type the name of the class enclosed in quotes (e.g. "TrendInfo").

4. The indicator will generate code for two functions and an indicator. The code is stored in three text files in the C:\ADE\Code directory.

5. Open the ADC.Get*Class*.txt file (where *Class* is the name of your class). Select all the text and copy it to the clipboard.

6. Create an EasyLanguage function called ADC.Get*Class*. Paste the copied text into the function and verify it.

7. Open the ADC.Put*Class*.txt file. Select all the text and copy it to the clipboard.

8. Create an EasyLanguage function called ADC.Put*Class*. Paste the copied text into the function and verify it.

9. If this is a parameterized class, open the ADC.*Class*PD.txt file. Select all the text and copy it to the clipboard. Create an EasyLanguage function called ADC.*Class*PD. Set the Return Type to "string (text)". Paste the copied text into the function and verify it.

10. Open the "ADC Save *Class*.txt" file. Select all the text and copy it to the clipboard.

11. Create an indicator called "ADC Save *Class*". Paste the copied text into the indicator and verify it.

12. The Get and Put class functions are now ready to use. Before you use the "Save" indicator to store data, you will need to modify it to include the calculations for the data you wish to store.

Indicators

The following sample indicators are provided with ADE. You can use them as models to build your own indicators and strategies.

ADE Save OHLCV

This indicator stores the Open, High, Low, Close, and Volume data for the current chart. This information can be retrieved by another chart using the ADE.GetOHLCV function.

The "UseFile" input tells the indicator whether to load existing data from a text file (if it exists) and whether to save the updated data at the end of the chart. By setting this to true, you can precalculate and store large amounts of data. You can even perform incremental updates on the data by setting the first bar of the chart slightly before the last bar of the previous run. (Some overlap is okay and ensures that you don't miss any bars.) The default value of this input is determined by the ADE.UseFile function. Just modify the function to change the default value (which is false). Of course, you can also change the input for specific charts.

If "UseFile" is true, the file is saved in the Data subdirectory of your ADE directory. For example, if you accepted the default location of "C:\ADE" when you installed ADE, then your data is saved in the C:\ADE\Data directory.

Note that the file name is automatically constructed from the class, symbol, and bar interval using the ADE.FileName function.

ADE Plot OHLCV

This indicator retrieves and plots the Open, High, Low, Close, and Volume data for the current symbol and the specified bar interval. This allows you to retrieve information for a bar interval different from the current chart. In order to retrieve the information, it must have been either stored in memory or saved in a file by ADE Save OHLCV.

The retrieved data is displayed as OHLC bars in a subgraph. In real applications, you would probably use the higher-timeframe data for analysis rather than simply displaying it.

The "Interval" input tells the indicator what bar interval to retrieve the data for.

The "UseFile" input tells the indicator whether to read the data from a file (if it exists). Note that the data will only be read from the file if the DataMap is empty; otherwise, the existing data will be used. ADE looks for files in the Data subdirectory of your ADE directory.

Note that the file name is automatically constructed from the class, symbol, and bar interval using the ADE.FileName function.

ADE Save Info Test

This indicator shows you how to calculate and store arbitrary metrics using the ADE.PutBarInfo function. It is intended as an example that you can use as a starting point for your own studies and strategies.

The indicator calculates and stores ADX, RSI, SlowK, and SlowD for the current symbol and bar interval.

The "UseFile" input has the same meaning as in "ADE Save OHLCV".

ADE Plot Info Test

This indicator shows you how to retrieve arbitrary metrics using the ADE.GetBarInfo function. It is intended as an example that you can use as a starting point for your own studies and strategies.

The indicator retrieves and plots ADX, RSI, SlowK, and SlowD for the current symbol and the specified bar interval. The data must already have been calculated and either stored in memory or saved in a file by the "ADE Save Info Test" indicator.

The "Interval" and "UseFile" inputs have the same meaning as in "ADE Plot OHLCV".

ADE Functions

ADE provides the following EasyLanguage functions.

High Level Functions

```
Value1 = ADE.GetBarInfo(Class, Sym, Interval, BarID, InfoMap);
```

Retrieves data for the specified class, symbol, bar interval, and Bar ID. The function copies the data into InfoMap for easy access. InfoMap should be a local map created with MapSN.New. For each data series in the DataMap, the function will create a Name->Value pair in InfoMap, where Name is the name of the data series, and Value is the value in the data series for the specified bar.

In order for this function to work, the data must have been previously stored with ADE.PutBarInfo, or it must have been loaded from a file with ADE.OpenMap.

If there is no entry in the DataMap for the specified bar, the InfoMap will remain unchanged. This allows you to use the values from previous GetBarInfo calls until data for a new bar is retrieved. For example, if the data is stored at 30 minute intervals but the current chart uses 5 minute intervals, there will only be new data every six bars (when the 5 minute bar aligns with the close of the 30 minute bar). In between the 30 minute bars, the InfoMap will retain the data from the most recent 30 minute bar.

See the "ADE Plot Info Test" indicator for an example of how to use this function.

```
Value1 = ADE.GetOHLCV(Sym, Interval, BarID, OpenOut, LowOut,
   HighOut, CloseOut, VolumeOut);
```

Retrieves the Open, High, Low, Close, and Volume values for the specified symbol, bar interval, and Bar ID.

In order for this function to work, the data must have been previously stored with ADE.PutOHLCV, or it must have been loaded from a file with ADE.OpenMap.

If there is no entry in the DataMap for the specified bar, the output arguments will remain unchanged. This allows you to use the values from previous GetBarInfo until data for a new bar is retrieved. For example, if the data is stored at 30 minute intervals but the current chart uses 5 minute intervals, there will only be new data every six bars (when the 5 minute bar aligns with the close of the 30 minute bar). In between the 30 minute bars, the output arguments will retain the data from the most recent 30 minute bar.

```
MapID = ADE.OpenMap(Class, Sym, Interval);
```

Gets the DataMap for the specified class, symbol, and bar interval, and populates it with data from a file. The data must have been previously saved with ADE.SaveMap.

This function will only read the data if the specified DataMap does not exist or is empty. Thus, you can call this function from multiple studies or strategies, and it will only read the data the first time it is called. On subsequent calls, it will simply return the already populated map.

```
Value1 = ADE.PutBarInfo(Class, Sym, Interval, BarID, InfoMap);
```

Copies the data in InfoMap to the specified class, symbol, bar interval, and Bar ID. InfoMap should be a local map created with MapSN.New. For each Name-Value pair in InfoMap, the function will find or create a data series with that Name, and it will store the Value in that data series at the specified bar (indicated by BarID).

See the "ADE Save Info Test" indicator for an example of how to use this function.

```
Value1 = ADE.PutOHLCV(Sym, Interval, BarID, OpenIn, LowIn,
   HighIn, CloseIn, VolumeIn);
```

Stores the Open, High, Low, Close, and Volume values for the specified symbol, bar interval, and Bar ID.

```
MapID = ADE.SaveMap(Class, Sym, Interval);
```

Writes the DataMap for the specified class, symbol, and bar interval to a file in the Data subdirectory of the ADE root directory. The file is automatically named based on the class, symbol, and interval. After you have called this function, you can later read back the data with ADE.OpenMap.

Helper Functions

`Date = ADE.BarDate(BarID);`

Returns the numeric date of the specified BarID in the standard EasyLanguage format (YYYMMDD).

`BarID = ADE.BarID;`

Returns a standard bar identifier for the current bar. The identifier is calculated as Date + Time / 10000. This function can be used wherever an ADE function expects a Bar ID.

`Interval = ADE.BarInterval;`

Returns the bar interval for the current chart. Unlike the BarInterval reserved word, this function returns a value that uniquely identifies daily, weekly, and monthly charts as well as intraday charts. It returns 0 for daily charts, -1 for weekly charts, and -2 for monthly charts. For intraday charts, it returns the number of minutes per bar, just like the BarInterval reserved word.

`Time = ADE.BarTime(BarID);`

Returns the numeric time of the specified BarID in the standard EasyLanguage format (HHMM).

`Interval = ADE.Daily;`

Returns the interval code for daily bars (0). You may use this as a more readable alternative to the numeric code.

`Value1 = ADE.DeleteMap(Class, Sym, Interval);`

Deletes the DataMap for the specified class, symbol, and interval. You don't need to worry about deleting DataMaps in most cases – the DataMaps for a class will be deleted automatically when you unload the last analysis technique that uses that class. However, there may be times when you want to perform your own cleanup of DataMaps, and ADE.DeleteMap allows you to do that.

`Success = ADE.DeriveBigBar(Interval);`

Derives a higher timeframe bar from the current chart data and stores the bar values in the OHLCV class. The higher timeframe is designated by the Interval argument. This function must be called on every bar.

The function returns true once the first full bar has been derived; before that, it returns false. You should make sure the function returns true before you try to fetch OHLCV data for the higher timeframe.

ADE.DeriveBigBar just calls ADE.DeriveOHLCV with the open, high, low, close, and volume for the current chart and symbol. Please see the description of that function for more information.

```
Success = ADE.DeriveOHLCV(Sym, Interval,
      OpenIn, HighIn, LowIn, CloseIn, VolumeIn);
```

Derives a higher timeframe bar from the specified open, high, low, close, and volume for the current timeframe, and stores the bar values in the OHLCV class for the specified symbol. The higher timeframe is designated by the Interval argument. This function must be called on every bar.

The function returns true once the first full bar has been derived; before that, it returns false. You should make sure the function returns true before you try to fetch OHLCV data for the higher timeframe.

You can use this function to derive higher timeframe bars all the way up to monthly bars. However, if you derive daily, weekly, or monthly bars from intraday bars, you should be aware that the bars will probably not exactly match the bars on a daily, weekly, or monthly chart. This happens because the closes on these charts reflect the official settlement prices for each day. In addition, there are factors that may cause the daily open to be different, and these differences may result in differing highs and lows as well. Thus, the synthetic bars created by ADE.DeriveOHLCV will be an accurate summary of intraday activity, but there will often be small differences from the bars on daily or higher charts.

If you derive weekly or monthly bars from daily bars, the derived bars should match those on weekly or monthly charts, since they start with daily data.

The function will not start deriving the first bar until there is a bar transition (e.g. from one 60 minute bar to the next). This ensures that the first bar contains all the relevant data.

The ADE.DeriveOHLCV and ADE.DeriveBigBar functions have been tested on regular sessions for stocks, futures (both day and extended sessions), and Forex. In addition, they have been tested on stock charts with pre- and post-market sessions included. They may not work correctly on other custom sessions.

Daily and higher derived bars will always use the regular session. However, you can derive daily bars on a stock chart that includes the pre- and post-market sessions; the function will simply ignore these sessions when calculating daily or higher bars.

Since deriving higher-timeframe bars is a fairly complex process, it is strongly recommended that you carefully test any indicators or strategies that use these functions to make sure that they are doing what you intend.

```
Directory = ADE.Directory;
```

Returns the ADE root directory (C:\ADE by default). The ADE functions will store data in a subdirectory of this directory called “Data”. If you use the “ADE Generate Class” indicator, it will read class definitions from a subdirectory called “Classes”, and it will output the generated functions and indicator to a subdirectory called “Code”.

The ADE directory is specified when you install ADE. The installer saves the location of this directory in the Windows registry, and the ADE.Directory function reads that location from the registry.

If you move or rename the directory after installing ADE, there's a simple way to update the path in the registry. First, it's a good idea to close the TradeStation platform. Then navigate to your ADE directory in Windows Explorer and double-click on the UpdateADEPath.cmd file. You will see a short message confirming that the path has been updated. The next time you apply a study that uses ADE, the ADE.Directory function will return the new location.

```
Index = ADE.GetBarIndex(Class, Sym, Interval, BarID);
```

Returns the index for the specified class, symbol, bar interval, and BarID. You may use this index to retrieve a value from any of the data series in the class.

This function can be used together with ADE.GetBarIndex to define ADE series functions. For more information about this type of function, see the section on "Series Functions" below.

```
Value1 = ADE.GetFileIntervals(Class, Sym, IntervalList);
```

Fills a list with all of the bar intervals available in data files for the specified class and symbol. The function searches the ADE Data directory to determine which intervals are available. You should pass a ListN for the IntervalList argument.

```
Value1 = ADE.GetFileSymbols(Class, SymbolList);
```

Fills a list with all of the symbols available in data files for the specified class. The function searches the ADE Data directory to determine which symbols are available. You should pass a ListS for the SymbolList argument.

```
Value1 = ADE.GetIntervals(Class, Sym, IntervalList);
```

Fills a list with all of the bar intervals currently in memory for the specified class and symbol. You should pass a ListN for the IntervalList argument.

```
MapID = ADE.GetMap(Class, Sym, Interval);
```

Returns the DataMap for the specified class, symbol, and bar interval. *Class* is a unique name that describes the contents of the map. The class "OHLCV" is reserved by the ADE library for the OHLCV functions, which store and retrieve Open, High, Low, Close, and Volume data for any symbol and bar interval.

If a DataMap does not already exist for the specified symbol and bar interval, ADE will create it and add it to the MasterMap.

You don't need to use this function if you use the higher-level ADE functions. However, it is provided in case you wish to access the DataMap directly.

MapID = ADE.GetRequiredMap(Class, Sym, Interval);

Returns the DataMap for the specified class, symbol, and bar interval, but only if the DataMap already exists and contains data. If the DataMap does not exist, this function raises a runtime error.

ListID = ADE.GetSeries(Class, Sym, Interval, SeriesName);

Returns the List for the specified class, symbol, bar interval, and series name. The series name must be a valid name for the specified class, or the function will raise a runtime error.

For example, to get the List that contains the Close series in the OHLCV class, you can use the following statement:

```
CloseList = ADE.GetSeries("OHLCV", GetSymbolName, Interval, "Close");
```

This function can be used together with ADE.GetBarIndex to define ADE series functions. For more information about this, see the section on "Series Functions" below.

Value1 = ADE.GetSymbols(Class, SymbolList);

Fills a list with all of the symbols currently in memory for the specified class. You should pass a ListS for the SymbolList argument.

Desc = ADE.IntervalDesc(Interval);

Returns a text description of the specified ADE interval, e.g. "30min" for 30, "Daily" for 0, "Weekly" for -1, or "Monthly" for -2.

Higher = ADE.IsHigherTF(Interval);

Returns true if the specified ADE interval is higher than the interval of the current chart. For example, if the current chart is a 30 minute chart, ADE.IsHigherTF(0) will return true, since the interval 0 represents daily intervals.

Interval = ADE.Monthly;

Returns the interval code for monthly bars (-2). You may use this as a more readable alternative to the numeric code.

NormalizedBarID = ADE.Normalize(BarID);

Returns a "normalized" version of the specified BarID. If the current interval is daily, weekly, or monthly, this function sets the time of the BarID to 2400, the last possible time of the day. If the current interval is intraday, the function returns the BarID unchanged.

This function allows you to retrieve data for the same day (or week, or month) from another market, even if that market has a different closing time than the current market. If the other market closes at a later time and you don't normalize the bar, ADE will retrieve the bar from the previous day (because the BarIDs don't match, and the previous day is the closest prior bar). By normalizing the BarID to the last time of the day, you ensure that you get the bar for the same day instead of the previous day.

For example, suppose the current market closes at 1530 and the other market closes at 1600. If you request the data for 1041124.1530, ADE will return the bar for 1041123.1600 (the previous day), because that is the closest prior bar. But if you normalize the current bar ID to 1041124.2400, ADE will return the bar for 1041124.1600, which is the bar for the same day.

Note that you should only normalize BarIDs when you *request* data, not when you *store* data. Otherwise, you will end up storing normalized BarIDs instead of standard BarIDs.

Suppose you are performing an end-of-day analysis of daily bars for multiple markets, and you want to ensure that you retrieve the OHLCV data for matching days in each market, regardless of the time the markets close. Here's how you would do it:

```
Value1 = ADE.GetOHLCV(Sym, ADE.BarInterval, ADE.Normalize(ADE.BarID),
   vOpen, vHigh, vLow, vClose, vVolume);
```

If a particular analysis technique always works with intraday data, or if you work with markets that close at the same time, then you don't need to worry about normalizing BarIDs. This function is intended primarily to help with end-of-day analysis of daily or higher bars for a broad range of markets (e.g. generating a correlation matrix of futures markets).

The ADE.Normalize function can be used safely with intraday data, because it always returns an unchanged BarID for intraday charts.

`Test = ADE.OnNextBar(Class, Sym, Interval);`

Returns true when the next available bar becomes current in the DataMap for the specified class, symbol, and interval. If the DataMap contains higher timeframe data, you can use this function to check whether the next higher timeframe bar has been reached, so that you can avoid performing certain calculations between higher timeframe bars.

For example, the following indicator code displays a moving average for an arbitrary class, series, and interval. It checks ADE.OnNextBar and only updates the average when the next bar appears:

```
Inputs:
   Interval(ADE.Daily),
   Class("OHLCV"),
   Series("Close"),
   Length(20);

Vars:
   Avg(0);

if ADE.OnNextBar(Class, GetSymbolName, Interval) then
   Avg = ADF.Average(Class, GetSymbolName, Interval, ADE.BarID,
      Series, Length);

if Avg <> 0 then
   Plot1(Avg, "ADU Avg");
```

If the specified interval is the same as the current chart, then ADE.OnNextBar will always return true, since a new bar in the DataMap becomes available for each bar on the chart. Thus, you can safely use this function when different intervals may be used, including the same interval as the current chart.

You do not need to call this function in order for your code to work correctly. It is provided as an optional tool to make your code more efficient.

```
Value1 = ADE.ReadDataFiles(Class);
```

Reads all the data files for the specified class into memory. This provides a convenient way to load all the data files for a class at the same time. However, if there are a large number of data files for the class and these files contain many bars, the function may not be able to load them all before the TradeStation infinite loop timeout occurs (even though this function will not really be in an infinite loop). In such cases it is best to load the data for each symbol individually by applying a Save indicator to each chart.

Remember to call this function only when CurrentBar = 1. Otherwise, it can slow down your strategy or study dramatically.

```
UseFile = ADE.UseFile;
```

Returns the default UseFile value (false). This is used by ADE indicators to determine whether to store and load ADE data files. If you wish to change the default, just modify the function to return true instead of false.

```
Interval = ADE.Weekly;
```

Returns the interval code for weekly bars (-1). You may use this as a more readable alternative to the numeric code.

Low Level Functions

```
IsFile = ADE.FileExists(FileName);
```

Returns true if the specified file exists. ADE uses this function to check for the existence of a data file before attempting to load it.

```
FileName = ADE.FileName(Class, Sym, Interval);
```

Returns a standardized file name using the ADE Data directory and the specified class, symbol, and interval. You can use this function to generate file names for storing ADE data.

```
ListID = ADE.FindOrAddListN(MapID, Key);
```

Looks up the ListN associated with Key in the specified map. If it exists, the function returns the list. Otherwise, the function creates a ListN, adds Key->ListID to the map, and returns the ListID.

```
Value1 = ADE.GenerateCode(TemplateFile, OutputFile, Class, Members);
```

This function is used by the "ADE Generate Class" indicator to generate code for a specific function or indicator. *TemplateFile* contains the template code for the function or indicator.

OutputFile is the name of the file in which to save the generated code. *Class* is the name of the ADE class for the generated code. *Members* is a ListS of the members in the class.

`Value1 = ADE.ParseParamStr(ParamStr, ParamName, ParamValue, ParamDec);`

Parses a parameter definition from a class file into a name, value, and number of decimals. This function is used by the ADE.GenerateCode function.

`Value1 = ADE.PutOrPushN(ListID, Index, Value);`

If Index is zero, pushes Value onto the back of the specified list. Otherwise, the function puts Value at the specified Index.

`Value1 = ADE.RaiseError(Message);`

Raises a runtime error and displays the specified message in the Events Log. This function provides more robust error handling than the built-in RaiseRuntimeError. It also identifies the error as an ADE error.

`Result = ADE.ReplaceSepTag(Line);`

Replaces a <SEP=…> tag in a line of template code with the specified separator (which can vary depending on whether the current item is at the end of the list). This function is used by the ADE.GenerateCode function.

`Result = ADE.ReplaceStr(SourceStr, FromStr, ToStr);`

This function is used by the ADE.GenerateCode function to replace placeholders with actual names. The function replaces all instances of *FromStr* in *SourceStr* with *ToStr* and returns the resulting string.

`Value1 = ADE.SortDataMap(DataMap);`

Sorts all of the series in the specified DataMap by the BarIDs in the ":Bars" list, while keeping all the series synchronized with the BarIDs. Data is usually in the correct order when it is loaded, but there are some scenarios which can cause the data to become un-sorted. This function is called by higher-level functions to sort the map when necessary.

`Value1 = ADE.ValidateName(Name);`

Checks whether *Name* is a valid name for a member in a class, and raises a runtime error if it is not. The names "Date" and "Time" are reserved by ADE for writing and reading text files, so you will receive a runtime error if you try to use them in a class. ADE replaces Bar IDs with Date and Time columns in text files in order to make the files more readable.

Series Functions

You can use the ADE.GetSeries and ADE.GetBarIndex functions to create EasyLanguage functions which perform operations on ADE data series. These functions operate on the native time frame of the data series, regardless of the timeframe in which the series is being used. For example, suppose you have stored OHLCV data for MSFT for 30 minute bars. You can use an ADE series function to average the Close values for the 30 minute bars, even though you may be performing this calculation in a 5 minute chart.

In most cases it is probably more efficient to calculate and store the data you need in the native time frame, so that you can just retrieve the pre-calculated data in another chart. However, there may be times when you need to perform additional calculations on the ADE data series, and ADE series functions provide a way to do that.

ADE provides some sample series functions for you to use and study. In order to keep these functions separate from the core ADE library, they are all prefixed with ADF (which stands for All-Data-Function). Here is a description of the functions:

`Result = ADF.Average(Class, Sym, Interval, BarID, SeriesName, Length);`

Takes the data series for the specified class, symbol, and interval, and averages the last *Length* bars of data. The calculation is performed at the bar specified by *BarID*. For example, to average the last ten 30 minute Closes, starting with the bar that corresponds to the current 5 minute bar, you could use the following statement:

```
Result = ADF.Average("OHLCV", GetSymbolName, 30, ADE.BarID, "Close", 10);
```

`Result = ADF.AverageFC(Class, Sym, Interval, BarID, SeriesName, Length);`

This function performs the same calculation as ADF.Average, but it uses a more efficient algorithm. However, it order to calculate the results correctly, this function must be called on every bar. This restriction does not apply to ADF.Average.

`Result = ADF.Lowest(Class, Sym, Interval, BarID, SeriesName, Length);`

Takes the data series for the specified class, symbol, and interval, and finds the lowest value in the last *Length* bars of data. The calculation is performed at the bar specified by *BarID*. For example, to find the lowest low in the last ten 30 minute bars, starting with the bar that corresponds to the current 5 minute bar, you could use the following statement:

```
Result = ADF.Lowest("OHLCV", GetSymbolName, 30, ADE.BarID, "Low", 10);
```

`Result = ADF.LowestFC(Class, Sym, Interval, BarID, SeriesName, Length);`

This function performs the same calculation as ADF.Lowest, but it uses a more efficient algorithm. However, it order to calculate the results correctly, this function must be called on every bar. This restriction does not apply to ADF.Lowest.

`Result = ADF.Highest(Class, Sym, Interval, BarID, SeriesName, Length);`

Takes the data series for the specified class, symbol, and interval, and finds the highest value in the last *Length* bars of data. The calculation is performed at the bar specified by *BarID*. For example, to find the highest high in the last ten 30 minute bars, starting with the bar that corresponds to the current 5 minute bar, you could use the following statement:

```
Result = ADF.Highest("OHLCV", GetSymbolName, 30, ADE.BarID, "High", 10);
```

```
Result = ADF.HighestFC(Class, Sym, Interval, BarID,
     SeriesName, Length);
```

This function performs the same calculation as ADF.Highest, but it uses a more efficient algorithm. However, it order to calculate the results correctly, this function must be called on every bar. This restriction does not apply to ADF.Highest.

```
Result = ADF.Summation(Class, Sym, Interval, BarID,
     SeriesName, Length);
```

Takes the data series for the specified class, symbol, and interval, and sums the last *Length* bars of data. The calculation is performed at the bar specified by *BarID*. For example, to sum the last ten 30 minute Opens, starting with the bar that corresponds to the current 5 minute bar, you could use the following statement:

```
Result = ADF.Summation("OHLCV", GetSymbolName, 30, ADE.BarID, "Open", 10);
```

```
Result = ADF.SummationFC(Class, Sym, Interval, BarID,
     SeriesName, Length);
```

This function performs the same calculation as ADF.Summation, but it uses a more efficient algorithm. However, it order to calculate the results correctly, this function must be called on every bar. This restriction does not apply to ADF.Summation.

Writing Your Own Series Functions

You can study the provided series functions and use them as a model to write your own series functions. If you want to post your function library for others to use, please follow these guidelines to prevent confusion:

- Prefix the function with "ADF." This will identify it as an ADE series function.
- List all the "standard" arguments first in the following order: Class, Sym, Interval, BarID, and SeriesName.
- Follow the standard arguments with any arguments which are specific to your function.

Support for TypeZero Bars

ADE includes several functions that make it possible to work with tick and volume bars, as well as with virtual bars that are derived from these bar types. Since the BarType reserved word returns zero for these kinds of bars, we refer to them collectively as "TypeZero bars".

Type Codes and Interval Codes

There are four kinds of standard TypeZero bars:

☑ Tick bars with Trade Volume for volume
☑ Tick bars with Tick Count for volume
☑ Volume bars with Trade Volume for volume

☑ Volume bars with Tick Count for volume

The type of bar is specified by a numeric *Type Code*. The following Type Codes are predefined for the standard TypeZero bar types:

Type Code	Bar Type	Nickname
11	Tick Bar with Trade Volume	TickBarV
12	Tick Bar with Tick Count	TickBarT
21	Volume Bar with Trade Volume	VolBarV
22	Volume Bar with Tick Count	VolBarT

In addition to the *type* of the bar, there is also an *interval size*. For tick bars, this is the number of ticks in each bar; for volume bars, it is the number of shares/contracts in each bar.

Both the Type Code and the interval size must be encoded in a single number that can be passed to the Interval input which is used by many ADE functions and indicators. The function that does the encoding is called ADE.TypeZeroInterval, and it is defined as follows:

```
Inputs:
   TypeCode(NumericSimple),
   IntervalSize(NumericSimple);

ADE.TypeZeroInterval = -TypeCode - IntervalSize / 1000000000;
```

This function makes the Type Code negative in order to distinguish it from minute bars, and it includes the interval size as a decimal. The resulting number is called an *Interval Code*. For example, the interval code for a 5000-share volume bar would be -21.000005000. You don't really have to worry about these details – you can just use ADE.TypeZeroInterval to construct the appropriate interval code whenever an ADE function or indicator needs an Interval.

ADE also provides four functions that construct interval codes for the standard TypeZero bars: ADE.TickBarV, ADE.TickBarT, ADE.VolBarV, and ADE.VolBarT. These functions are named with the nicknames in the table above, and they take a single argument, which is the interval size. Thus, for 5000-share volume bars that use Trade Volume for volume, you can specify either ADE.TypeZeroInterval(21, 5000) or ADE.VolBarV(5000). You may find the "nickname" functions easier to use, since you won't have to remember the Type Codes for the standard TypeZero bar types. These functions are analagous to ADE.Daily, ADE.Weekly, and ADE.Monthly for time bars.

TypeZero BarIDs

In addition to specialized interval codes, TypeZero bars require specialized BarIDs. This is necessary because there can be more than one TypeZero bar in a single minute. As a result, standard BarIDs (which include only the date, hour, and minute) cannot provide reliably unique IDs for TypeZero bars. ADE solves this problem by including a counter in the BarID along with the date and time. The counter resets to one on each new minute and increases by one for each bar that appears during that minute. This ensures that each TypeZero bar has a unique ID, and it

is also compatible with ordinary BarIDs, since the date and time are still encoded in the same way.

To get the BarID for a TypeZero bar, you must call the ADE.TypeZeroBarID function. *This is important!* You will get unreliable results if you use ADE.BarID for TypeZero bars.

The ADE.TypeZeroBarID function updates the bar counter, divides it by one billion, and adds it to the standard BarID. For example, suppose a chart contains three TypeZero bars for 01/25/2005 at 11:37. These bars would have the following BarIDs:

```
1050125.113700001
1050125.113700002
1050125.113700003
```

This format allows up to 99,999 bars per minute, which should be sufficient for the foreseeable future.

Note that the standard ADE indicators such as "ADE Save OHLCV" call the ADE.BarID function, so you can't use these indicators with TypeZero bars. You will generally need to write different versions of indicators for TypeZero bars than for time-based bars, or you will need to download indicators that have been specifically designed for TypeZero bars. The "TypeZero Sync" library contains several such indicators, including a "TZU Save OHLCV" indicator that saves OHLCV data for TypeZero bars. This library is discussed in the next section.

Synchronizing TypeZero Bars

ADE provides the basic functions for storing and retrieving data for TypeZero bars, but you will need more than this if you want to pass data between TypeZero charts with different bar intervals. Unlike time-based charts, you can't use the BarID for the current TypeZero chart to retrieve data from a different TypeZero chart (unless it uses the same symbol and interval).

A simple example will illustrate the problem. Suppose you have 250-share bars in one chart and 1000-share bars in another. Furthermore, suppose that a particular minute contains eight of the 250-share bars and two of the 1000-share bars. What happens if you use the BarID for the second 250-share bar to retrieve a 1000-share bar? The counter component in the BarID is 2 (since it is the second bar in that minute), so it will retrieve the second 1000-share bar in that minute, which also has a counter equal to 2. But the second 1000-share bar is almost certainly later than the second 250-share bar in the same minute! Thus, you would be accessing data that is later in time (if you are backtesting), which is a very bad idea. It will probably make your results look great, but they will be totally fictitious.

The solution to this problem is provided by the "TypeZero Sync" framework, which is posted in a separate topic in the TSW File Library. This framework uses cumulative session activity (ticks or volume, depending on the bar type) to synchronize the bars properly. Although this sounds complicated, the framework hides the details and does most of the work for you.

If you want to use ADE with TypeZero charts, please go to the "TypeZero Sync" topic for more information. In addition to the framework itself, this topic includes some sample indicators and workspaces that show you how to work with TypeZero bars. It is strongly recommended that you study these indicators carefully and use the demonstrated techniques in your own code.

Remember: If you want to pass data between TypeZero charts, you must use TypeZero Sync!

TypeZero Bars and "Update Values Intrabar"

Although it is possible to use ADE and TypeZero bars with the "Update values intrabar" option enabled, there are several important things you should know:

1. When "Update values intrabar" is enabled, ADE needs to create a BarID as soon as a bar opens, since the EasyLanguage code executes on every tick.

2. When a TypeZero bar opens, there is no way to predict what time it will close, since the close of the bar is based on ticks or volume rather than time.

3. Therefore, ADE has no choice but to use the open time for the bar rather than the close time, since it needs to create a BarID right away.

This will not cause problems *as long as both the sending and receiving chart have "update intrabar" enabled.* If one chart has update intrabar turned on and the other doesn't, then the first chart will use bar open times for real-time bars and the second will use bar close times, so the BarIDs will be incompatible. Also, the charts must be set up at the same time so that intrabar updates are being used for the corresponding bars in both charts. It is strongly recommended that the charts be set up before the trading session opens, since this will ensure that intrabar updates start at the same time for both charts. However, it should also work if the charts are saved in workspaces and you open them within the same minute during the trading day.

Furthermore, it is not a good idea to save data to files if it was stored with "update intrabar" enabled. Since the data stored in ADE files is typically historical data, it will use the bar close time rather than the bar open time. If you then save real-time data that was stored with "update intrabar", you will have mixed BarIDs that use both bar close and bar open times. This is likely to cause problems, so you should make sure that the UseFile input is false when using "update intrabar".

The safest course is not to use "update intrabar" at all with ADE and TypeZero bars. However, if you really need it, you can make it work if you observe these precautions:

- Both sending and receiving charts must use the same setting for "update intrabar".
- Set up the charts at exactly the same time, preferably before the start of the trading session.
- Don't save data to files if it was stored with "update intrabar" enabled.

Virtual Bar Types

It is possible to use ADE and TypeZero bars to create and store virtual bars. ADE itself does not implement this functionality, but it provides support for defining Type Codes and nicknames for custom bar types. This allows virtual bar data to be stored in data files with meaningful names, and it also allows ADE to display a description of the bar type in error messages.

To tell ADE about virtual bar types, create a text file called VirtBarTypes.cfg in your ADE directory. For each bar type, add a line with a Type Code and name, separated by a comma and no spaces. For example, the following file defines two new virtual bar types: constant-range bars and Renko bars:

```
201,vbCRB
301,vbRenko
```

The bar type names should be clear but concise, with no punctuation, since they will be included in the file name if you save data for virtual bars. Also, if you develop a virtual bar utility and decide to post it on TSW, please use Type Codes of 100 or greater, and make sure that they don't conflict with the codes used by any other ADE virtual bar utilities that have been posted. Type Codes under 100 are reserved for possible future use in the ADE library.

TypeZero Functions

Here is a summary of the TypeZero functions.

`Interval = ADE.TypeZeroInterval(TypeCode, IntervalSize);`

Returns an interval code for a TypeZero bar with the specified TypeCode and IntervalSize. This interval code can be passed to any ADE function or indicator that has an Interval input.

`BarID = ADE.TypeZeroBarID;`

Returns a BarID for the current TypeZero bar. This function must always be called instead of ADE.BarID for TypeZero bars.

`Value1 = ADE.TypeZeroParseID(BarID, oDate, oTime, TZCount);`

Parses the specified TypeZero BarID into its date, time, and count components, and returns these values in the oDate, oTime, and TZCount arguments.

`Interval = ADE.TickBarV(IntervalSize);`

Returns the interval code for a Tick Bar with Trade Volume for volume.

`Interval = ADE.TickBarT(IntervalSize);`

Returns the interval code for a Tick Bar with Tick Count for volume.

`Interval = ADE.VolBarV(IntervalSize);`

Returns the interval code for a Volume Bar with Trade Volume for volume.

```
Interval = ADE.VolBarT(IntervalSize);
```

Returns the interval code for a Volume Bar with Tick Count for volume.