

Trading Blox Builder's Guide

© 2013, Trading Blox, LLC. All rights reserved.



Trading Blox Builder's Guide

By Traders... For Traders

Trading Software for Mechanical Systems Traders

© 2013, Trading Blox, LLC. All rights reserved.

*This user's guide and all its contents are copyright
2003-2013, Trading Blox LLC*

*Trading Blox LLC
www.tradingblox.com
508 SE Osceola St
Stuart, FL 34996
978-222-3111*

Table of Contents

Foreword	0
Part I Getting Started Tutorial	2
1 What Are Blox?	3
2 Creating a New System	4
1. New System Blox	6
2. Adding Parameters	10
3. Adding Indicator	14
4. Entering Code	17
5. Building A System	19
6. Creating A Suite	23
3 Improving a New System	25
Protective Position Pricing	26
Copy System Items	29
Protective Exit Orders	37
Entry Order Protection	44
Active Order Protection	48
Order Sizing	52
Trading Risk	58
Money Management	62
Part II Trading Blox Architecture	65
1 Working with Systems, Blox & Scripts	69
Working with Systems	69
Working with Blox	72
Working with Scripts	75
Basic Scripts	78
2 Process Flow	79
3 Simulation Loop	80
4 Comprehensive Simulation Loop	81
Part III Blox Module Reference	86
1 Blox Types	87
Portfolio Manager	88
Entry	89
Exit	90
Money Manager	91
Risk Manager	92
Auxiliary	93
2 Blox Script Access	94
3 Blox Script Timing	96
4 Global Script Timing	98
5 Script Section Type Details	101
6 Scripts Common to Many Blox	103

7 Script Section Descriptions	104
Before Simulation	106
Before Test	107
Rank Instruments	108
Filter Portfolio	109
Before Trading Day	110
Before Instrument Day	111
Before Bar	112
Exit Orders	113
Entry Orders	114
Unit Size	115
Can Add Unit	116
Before Order Execution	117
Update Indicators	118
Can Fill Order	119
Exit Order Filled	120
Entry Order Filled	121
After Instrument Open	122
After Bar	123
Adjust Stops	124
Initialize Risk Management	125
Compute Instrument Risk	126
Compute Risk Adjustment	127
Adjust Instrument Risk	128
After Instrument Day	130
After Trading Day	131
After Test	132
After Simulation	133
 Part IV Blox Basic Language Reference	 135
1 Basic Keywords	136
2 Colors	137
3 Constants Reference	142
4 Data Groups and Types	144
Boolean	147
Floating	149
Instrument - BPV	149
Integer	152
Money	152
Percent	153
Price	155
Selector	155
Series	157
Numeric Series.....	158
String Series.....	160
String	161
5 Data Scope Reference	163
6 Data Script Comments	165
7 Data Variable Names	166
8 Data Variables	168

9 FunctionReference	171
Custom Functions	171
Custom User Functions.....	172
Date Time Functions	176
ChartTime.....	176
DateToJulian.....	179
DayMonthYearToDate.....	180
DayOfMonth.....	181
DayOfWeek.....	182
DayOfWeekName.....	183
DaysInMonth.....	184
Hour.....	185
JulianToDate.....	186
Minute.....	187
Month.....	188
MonthName.....	189
SystemDate.....	190
SystemTime.....	191
TimeDiff.....	192
WeekNumberISO.....	193
Year.....	195
File & Disk Functions	196
ClearLogWindow.....	197
CloseLogWindow.....	198
CopyFile.....	198
CreateDirectory.....	199
DeleteFile.....	199
EditFile.....	200
Extract.....	200
FileExists.....	200
FileSize.....	201
MoveFile.....	201
OpenFile.....	201
OpenFileDialog.....	202
OpenLogWindow.....	203
SaveFileDialog.....	203
General	205
BuildDividendFiles.....	206
ColorRGB.....	207
FileVersion.....	211
FileVersionNumerical.....	212
GetRegistryKey.....	212
LicenseName.....	214
LineNumber.....	216
Message Box.....	217
PlaySound.....	221
Preference Items.....	223
ProductVersion.....	224
ProductVersionNumerical.....	225
SetRegistryKey.....	225
Mathematical Functions	227
AbsoluteValue.....	228
ArcCosine.....	230
ArcSine.....	230

ArcTangent	231
ArcTangentXY	231
Average	231
CAGR	232
Ceiling	233
Correlation	234
CorrelationLog	234
Cosine	235
DegreesToRadians	235
EMA	236
Exponent	237
Floor	238
Hypotenuse	239
IfThenElse	239
IsUndefined	240
Log	241
Max	241
Min	241
RadiansToDegrees	242
Random	242
RandomDouble	243
RandomSeed	243
Round	244
Sign	245
Sine	247
Square Root	247
StandardDeviation	247
StandardDeviationLog	248
SumValues	248
Tangent	249
String Functions	250
ASCII	252
ASCIIToCharacters	253
FindString	254
FormatString	255
GetField	261
GetFieldCount	262
GetFieldNumber	263
LowerCase	264
LeftCharacters	265
MiddleCharacters	266
RemoveCommasBetweenQuotes	267
RemoveNonDigits	269
ReplaceString	270
RightCharacters	271
StringLength	272
TrimLeftSpaces	273
TrimRightSpaces	274
TrimSpaces	275
UpperCase	276
Type Conversion Functions	277
AsFloating	278
AsInteger	279
AsSeries	280

AsString.....	281
IsFloating.....	283
IsInteger.....	284
IsString.....	285
10 Indicator Reference	286
Basic Indicators	286
Creating Indicators	289
Calculated Indicators	292
Custom Indicators	294
Indicator Access	295
11 Indicator Pack 1	297
Indicator Pack 1 Indicators	298
Average Trend Channel.....	301
Chaiken Money Flow.....	303
Commodity Channel Index.....	307
Kaufman Adaptive Moving Average.....	310
Keltner Channel.....	314
Trend Vigor.....	319
Indicator Pack 1 Series Functions	321
EhlersZeroLagEma	322
InstantaneousTrendLine.....	324
MarketNoise.....	324
MedianAbsoluteDeviation.....	326
Momentum.....	328
MRO.....	330
Percentile.....	332
PercentRank.....	333
RateOfChange.....	335
SpearmanCorrelation.....	337
SpearmanCorrelationSync.....	338
SpearmanLogCorrelation.....	339
SpearmanLogCorrelationSync.....	340
WMA - Weighted M-Avg.....	341
Z-Score.....	343
ValueChart.....	345
12 Operator Reference	347
Comparison	348
13 Permanent Variables	349
Data Parameter Reference	349
Block Permanent Variables	351
Instrument Permanent Variables	353
14 Series Functions	357
Average	358
Correlation	359
CorrelationLog	360
CorrelationLogSynch	362
CorrelationSynch	363
CrossOver	363
Data Series Indexing	365
GetReference	367
GetSeriesSize	368
Highest	368

HighestBar	369
Lowest	370
LowestBar	371
Median	372
RegressionEnd	372
RegressionSlope	373
RegressionValue	375
RSI	375
SetSeriesColorStyle	376
SetSeriesSize	381
SetSeriesValues	382
SortSeries	385
SortSeriesDual	387
StandardDeviation	388
StandardDeviationLog	389
Sum	390
SwingHigh	390
SwingHighBars	391
SwingLow	392
SwingLowBars	393
15 Statement Reference	395
Assignment	396
DO	397
ERROR	399
FOR	400
IF	402
PRINT	404
WHILE	405
16 Trouble Shooting Script Problems	406
Debugger	407
Auto-Keyword Changes	409
Part V Trading Objects Reference	413
1 Alternate Objects	414
AlternateBroker Object	415
AlternateOrder Object	416
AlternateSystem Object	417
2 Block	418
Group	420
Name	421
ScriptName	422
System	423
SystemIndex	424
3 Broker	425
Entry Order Functions	429
EnterLongOnOpen	431
EnterShortOnOpen	432
EnterLongOnStopOpen	433
EnterLongAtLimitOpen	434
EnterShortOnStopOpen	435
EnterShortAtLimitOpen	437
EnterLongOnStop	439

EnterShortOnStop	440
EnterLongAtLimit	441
EnterShortAtLimit	442
EnterLongOnClose	443
EnterShortOnClose	444
EnterLongOnStopClose	446
EnterShortOnStopClose	447
EnterLongAtLimitClose	448
EnterShortAtLimitClose	449
Exit Order Functions	450
ExitAllUnitsOnOpen	452
ExitUnitOnOpen	453
ExitAllUnitsOnStopOpen	454
ExitAllUnitsAtLimitOpen	455
ExitUnitOnStopOpen	456
ExitUnitAtLimitOpen	457
ExitAllUnitsOnStop	458
ExitUnitOnStop	459
ExitAllUnitsAtLimit	460
ExitUnitAtLimit	461
ExitAllUnitsOnClose	462
ExitUnitOnClose	463
ExitAllUnitsOnStopClose	464
ExitUnitOnStopClose	465
ExitAllUnitsAtLimitClose	466
ExitUnitAtLimitClose	467
Position Adjustment Functions	468
AdjustPositionOnClose	468
AdjustPositionOnOpen	469
AdjustPositionOnStop	470
AdjustPositionAtLimit	471
4 Chart	472
AddBarLayer	479
AddBarSeries	482
AddContourLayer	484
AddLineLayer	487
AddLineSeries	489
AddScatter	493
Make	495
NewPie	497
NewXY	500
SetAxisTitle	502
SetBarGapShape	505
SetPlotArea	509
SetxAxisDates	512
SetxAxisLabels	514
5 Email Manager	517
Email Connect	518
EmailConnectSSL	519
Email Send	521
EmailSendHTML	522
EmailDisconnect	524
6 File Manager	525

Close	528
CountLines	529
DefaultFolder	530
EndOfFile	531
OpenAppend	532
OpenRead	533
OpenWrite	535
PartialLine	537
ReadLine	538
WriteLine	540
WriteString	542
7 Instrument	544
Data Properties	548
DataFunctions	553
AddCommission.....	553
Extract.....	554
GetDateTimeIndex	554
GetDayIndex.....	555
PriceFormat.....	556
RealPrice.....	556
RoundTick.....	557
RoundTickDown.....	557
RoundTickUp.....	557
Correlation Functions	558
ResetCloselyCorrelated.....	558
ResetLooselyCorrelated.....	558
AddCloselyCorrelated.....	559
AddLooselyCorrelated.....	559
Correlation Properties	560
Group Properties	562
Historical Trade Properties	564
Instrument Loading	566
LoadSymbol.....	567
LoadByLongRank.....	569
LoadByShortRank.....	570
LoadExternalData.....	570
LoadIPVFromFile.....	572
Position Functions	575
SetUnitCustomValue.....	575
SetExitStop.....	576
SetExitLimit.....	576
Position Properties	578
Ranking Functions	580
SetLongRankingValue.....	581
SetShortRankingValue.....	582
Ranking Properties	583
Trade Control Properties	584
Trade Control Functions	585
AllowLongTrades.....	586
AllowShortTrades.....	587
AllowAllTrades.....	588
DenyLongTrades.....	589
DenyShortTrades.....	590
DenyAllTrades.....	591

8 Order	592
OrderProperties	596
blockName	598
clearingIntent	599
continueProcessing	600
customValue	601
entryRisk	602
executionType	603
fillPrice	604
isBuy	605
isEntry	606
limitPrice	607
noStopPrice	608
orderPrice	609
orderReportMessage	610
orderType	611
position	612
quantity	613
referenceID	614
ruleLabel	615
sortValue	616
stopPrice	617
symbol	618
systemBlockName	619
timeInForce	620
unitNumber	621
OrderFunctions	622
Reject	623
SetClearingIntent	625
SetCustomValue	626
SetFillPrice	628
SetLimitPrice	629
SetOrderReportMessage	630
SetQuantity	632
SetRuleLabel	634
SetSortValue	636
SetStopPrice	640
SetTimeInForce	642
9 Script	643
Script Functions	650
Execute	651
GetSeriesValue	652
SetReturnValue	653
SetReturnValueList	654
Script Properties	655
ParameterCount	656
ParameterList	657
ReturnValue	658
ReturnValueList	659
SeriesParameterCount	660
StringParameterCount	661
StringParameterList	662
StringReturnValue	663

10 System	664
Global Suite System	665
System Functions	667
Accessing System Portfolio Instruments.....	668
RankInstruments.....	671
SetAccountNumber.....	672
SetAlternateOrder.....	673
System Properties	673
orderExists	675
11 Test	677
Equity Properties	678
General Properties	680
OrderReportPath.....	683
ResultsReportPath.....	685
SummaryResultsPath.....	687
Test String Arrays	688
Miscellaneous Functions	689
AbortSimulation.....	691
AbortTest.....	692
AddStatistic.....	693
CapitalAddsDraws	695
GetSteppedParameter.....	696
SetAlternateSystem.....	698
SetAutoPriming.....	700
SetChartSimulationHtml.....	701
SetChartTestHtml.....	704
SetGeneratingOrders	707
SetSilentTestRun.....	708
UpdateOtherExpenses	709
Test Statistics	710
Trade Properties	713
 Part VI Common Questions	 717
1 The Life of a Test	718
2 How Stops Work	719
3 Shortcut Keys	721
 Index	 723

Getting Started Tutorial

Part



Part 1 – Getting Started Tutorial

Trading Blox Builder Guide

Program Version: 4.3.0

Help Version: Thursday, December 19, 2013

During program development Trading Blox Builder used the code name of "Lego™." Lego's™ name was created by Ole Kirk Christiansen, in Billund, Denmark as the name of his company's small plastic toy blocks. Lego blocks assembled together can create small buildings, or other imagined toys .

In order to understand the philosophy of Trading Blox Builder you need to go back to the days of your youth, and to this game which most all of us have played with at one time or another. Lego Blocks™ allowed kids to create and assemble their own toys by connecting together parts called Lego Blocks. A child using Lego Blocks could build their own toys very quickly.

Trading Blox Builder has the same goal for slightly older kids; kids like us who like to trade.

Trading Blox Builder enables you to create complete trading systems, even very sophisticated trading systems, using basic building blocks we call Blox. With these Blox you can assemble a complete trading system very quickly just like kids can build their own toys with Lego Blocks.

We didn't stop there. We didn't just want to make it easier to develop systems with Blox, we also wanted to make it possible to test ideas, which you cannot test in most other environments. For this reason, we modeled the real world as closely as possible and the simulation test results that Trading Blox Builder creates does not take shortcuts.

This approach might seem harder to those who are used to some other products that have taken a simplified approach, but it should be easy for anyone who understands real trading because in real-world trading you already understand what is required in the way Trading Blox works.

Section 1 – What Are Blox?

Blox Modules

Blox are system modules that encapsulate trading ideas. Most of the Blox are self-contained parts of a trading system designed to be connected with other Blox as a component part of a trading system method. Some Blox can access data outside of their module, and outside of their system when their scope settings are set for external access.

The basic components of a trading idea are:

- **Parameters** - used by indicators to determine their specific computation result, for example: the number of days in a moving average. Not all Blox require parameters when the numerical basis for internal calculations is coded into the scripted code.
- **Indicators** - used by the rules as indicators of market conditions, moving averages, RSI, ADX, etc. Many indicators are available within the Indicator section of a Blox. Those not available in that area can be created by entering the source code required for an indicator calculation result.
- **Rules** - used to determine when to enter or exit; how much to buy or sell, or how much risk is too much, buy on moving average crossover, what instruments to allow and other ideas. A rule can be as simple as Buy If **RSI** > 55 etc.

By encapsulating trading ideas into a stand-alone Blox module, a package is created that can easily be linked to one or more systems that need the trading idea contained within the Blox. Blox are trading objects, and while these objects only need to be created once, they can be many times by other systems to simplify the creation of different system methods.

The simplest system can be created with one, or more Blox modules that do at least three things:

- 1) Enter Orders that define the System Entries.
- 2) Enter Orders for open positions that define the System Exits.
- 3) Define the Order Size for Each signal's Entry Order.

Trading Blox Builder will let you define this behavior using one block that performs all three functions, or three separate Blox that each take care of one of a primary functions. You don't need to worry about that right now. Just remember, Blox are like Lego blocks, they were designed to be connected to other Blox to build a trading system.

See Also:

[Blox Script Timing](#), [Blox Script Access](#)

Section 2 – Creating a New System

Overview:

To get you started with general understanding of how a system can be constructed in Trading Blox this topic and the sub-topics below this topic will provide steps you can follow in the creation of a simple always in the market moving average crossover system.

All the details that follow in this part of our tutorial area and in the following tutorial section are only intended to provide training for how to work with some of Trading Blox features. For understanding of what are some of the important features and options that successful trading methods need, those insights are best found by spending time examining and working with the provided trading systems, and by spending time reading some of the topics in our [Trader's Roundtable Forum](#).

Tutorial System Creation:

For our tutorial we will use a Crossover System based upon two Exponential Moving Averages (EMA) using the MACD as an oscillating indicator to create Buy and Sell signal orders that go in Long or Short position direction.

Crossover will be determined by using the Moving Average Convergence-Divergence (MACD) indicator. This indicator is a process that uses two Moving Average calculation results to create Convergence-Divergence (MACD) indicator. This indicator is one of the simplest and most effective momentum indicators available. The MACD turns two trend-following indicators, moving averages, into a momentum oscillator by subtracting the longer moving average from the shorter moving average. As a result, the MACD offers trend indication and momentum direction. In operation the MACD indicator values will move above and below zero as MACD value averages converge, cross and diverge.

In our new system we will use the "standard" MACD calculation to determine the difference between an instrument's 26-day and 12-day exponential moving averages. This is the formula that is used in many popular technical analysis programs, and quoted in most technical analysis books on the subject. Of the two moving averages that make up MACD, the 12-day EMA is the faster and the 26-day EMA is the slower in responding to market changes. Only the Close prices of the instrument are used to form the moving averages.

For our signal trigger we will generate a Buy signal to go Long when the MACD is positive, and a Sell signal to go Short when the MACD is negative.

Tutorial Steps & Topics:

Lesson:	Topic Description:
1	New System Blox
2	Adding Parameters
3	Adding Indicator
4	Entering Code
5	Building A System
6	Creating A Suite

Notes:

- During our discussion we will to use the term Price Bar, and Bar to refer to any type price record in a data file. For daily data a price bar is a trade record for that date. For intraday

data a price bar is dependent upon the number of minutes, i.e. 5-min, 10-min, 60-min, etc., of trade transactions contained as a series of data records for a specific date. An Intraday data record can be any one of the multiple price records of data aligned to the same trade date. However, each intraday price bar will have a unique time stamp so that the system can distinguish each intraday bar regardless of the time it was created.

Weekly price bars contain the week's range of prices printed by the market during that calendar week. Its prices reflect the Open price as the first market price reported and the Close price is the last price reported. A weekly High price is the highest price and the Low price is the lowest price reported during that same week. Monthly price bars for the same timing logic, but use the calendar month as its timing basis.

Weeks in most markets contain the trading data for each trade day of the week. Most often that means there are 5-trade days in a week, unless there is an exchange holiday or a day when the exchange closed for other reasons. Trading Blox can build weekly data price bars from the daily data file loaded when Trading Blox's **Preferences** dialog shows the **Process Weekly Data** option in the **Data Options** section of the **Data Folders and Option** menu item is enabled.

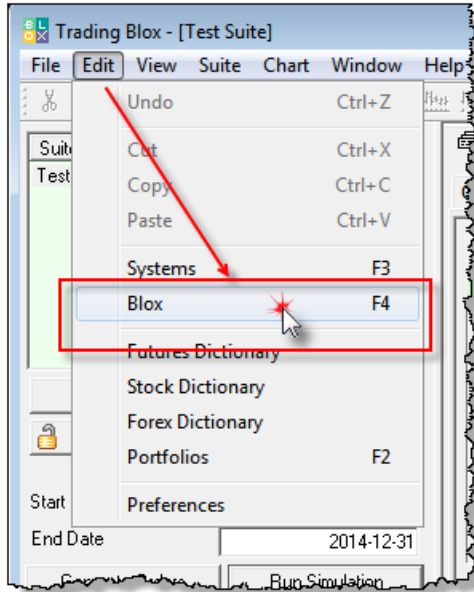
- Take a few minutes to understand how Trading Blox Basic uses its language operators by reviewing the tables on this page: [Operator Reference](#)

Links:

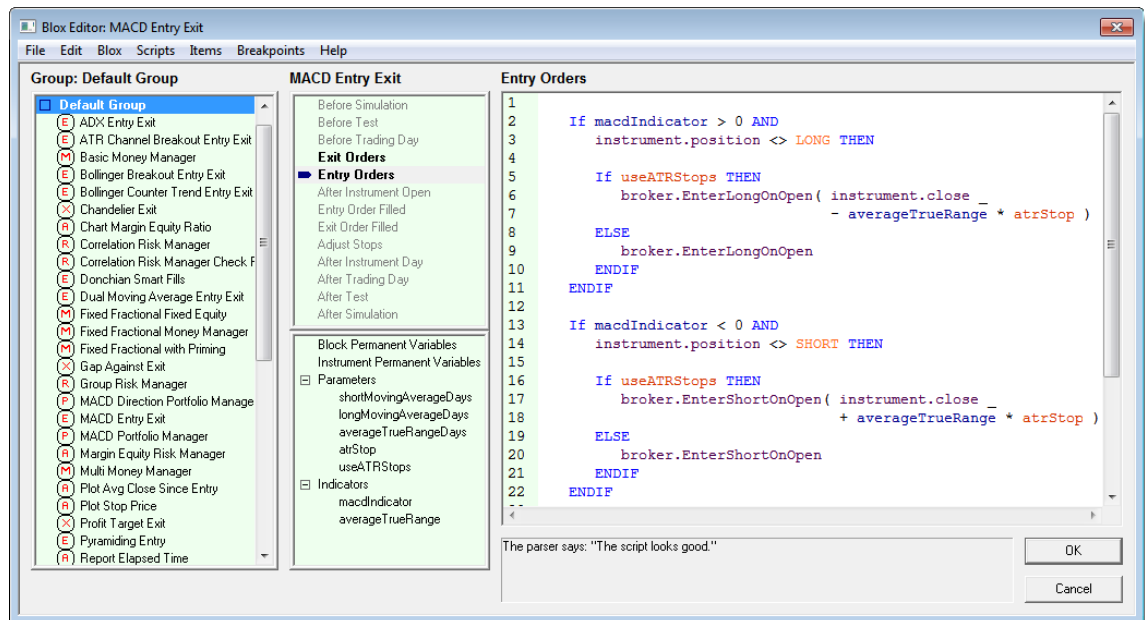
[Operator Reference](#)

2.1 1. New System Blox

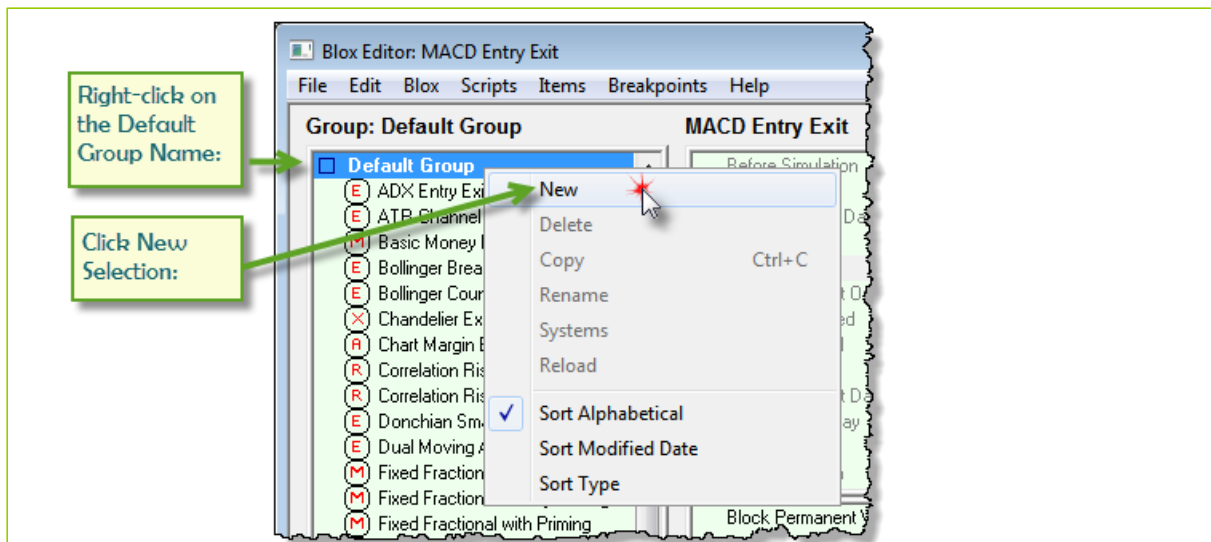
To get started select the **Edit** menu on the main screen, and then use your mouse to click on the **Blox** item on the drop down menu to select. You can optionally use the **F4** key on your keyboard, but some laptops require the Function-Key to use the **F-Keys**. Either way the **Trading Blox Editor** will appear:



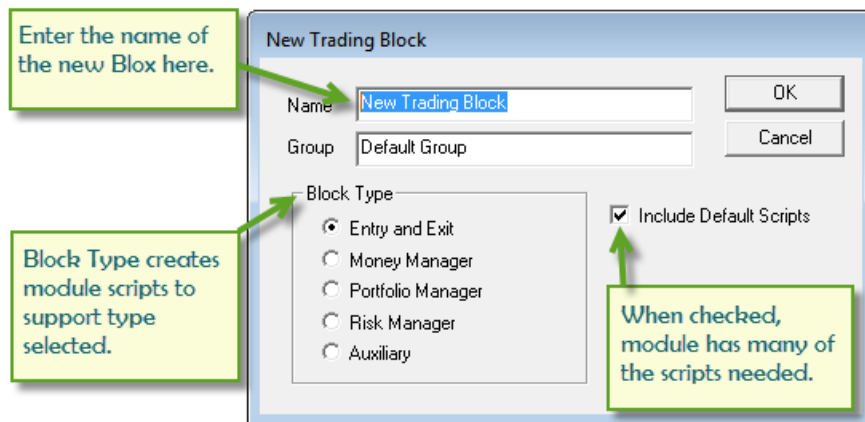
Trading Blox -- Blox Editor will appear and show the Group Listing and available Blox modules on the left. In the center the selected Blox module's script sections will appear, and below the script sections will be the listings of the Block Permanent Variables, Instrument Permanent Variables types, then the module's user parameters, and at the bottom the user created indicators:



Our next step is to create a new Blox module:



When the mouse is released a new dialog will appear that will allow you to name the new Blox and to also decide on what type of Blox your will be creating, and whether you want to add the common script sections for this blox automatically when it appears:

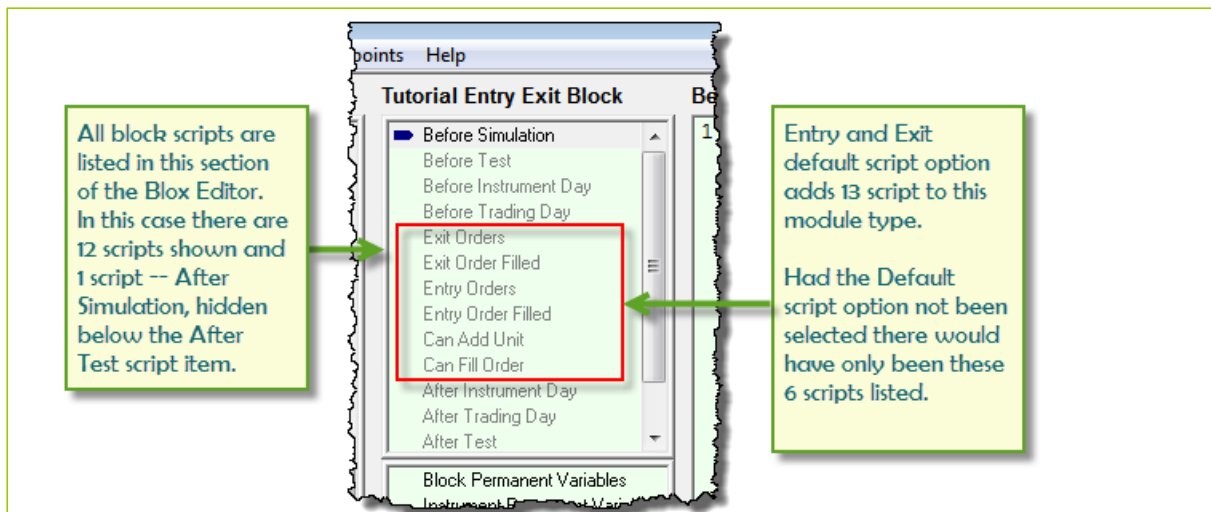


- Name this new Block the "**Tutorial Entry Exit Lesson 1**" by typing in the blox name in the Name field.
- Next check to see if the **Entry and Exit** option is the selected item in the **Block Type** section of the dialog. If **Entry and Exit** is not selected, select it now so the correct type of blox is created.
- Our last step is the enable the option to "**Include Default Scripts**" so we won't have to manually add them later.
- When you've completed these steps press OK so the module will appear in the **Default Group Listing**.

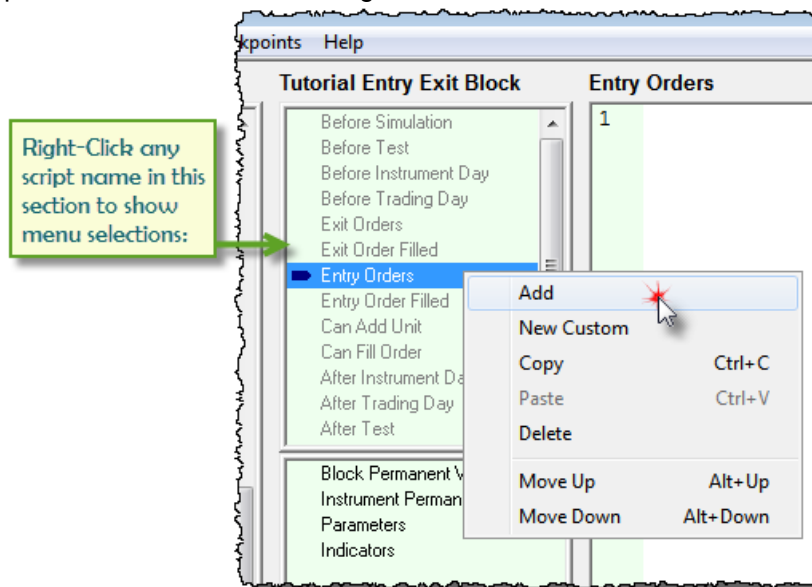
Note:

You could have changed the Group name to something other than Default Group. For example, you could have named a new group to be "**My Work**". When a Group name is entered in the Group field and it doesn't exist in the Group List, Trading Blox will automatically create the Group name and then place the Blox module into that Group section.

We now have a new block in the list, and ready to begin adding rules. When we created this tutorial block we elected to add all the common default scripts. When we review the list of scripts we find there are 13 names in the our list:

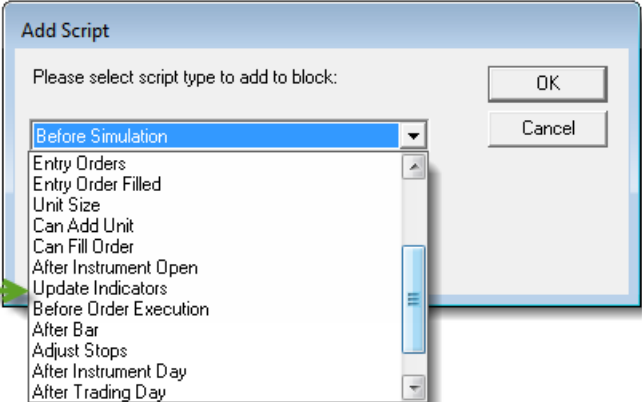


When we created our block, we enabled the Default Script option so the often used scripts in this type of module would be added and shown for this tutorial. Had we not used that option and if we needed a script section, it could have been easily added by right-clicking on any of the script names in the script list section and then selecting the **"Add"** menu item:

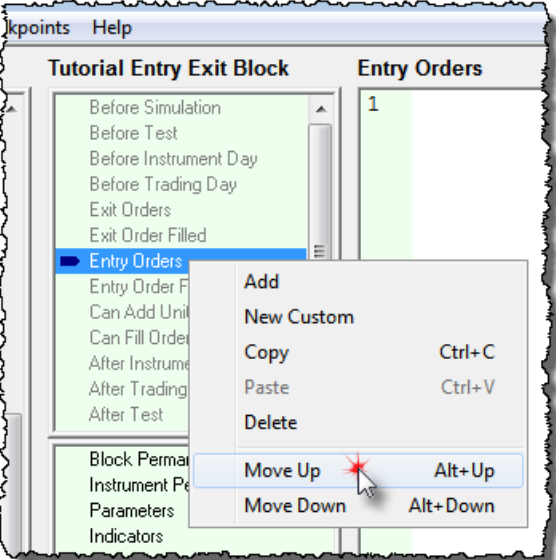


When then **"Add"** menu item is selected another dialog will appear with a drop down listing where all the scripts that are available in Trading Blox can be selected and added to the current module, should the selected script not exists:

Selecting any script name item not already shown as being in the block, will have that script section appear below the script item where the Right-Click Add menu was selected.



It is also possible to add a Custom Script section, but that is an advanced topic that will be explained in the Custom Script Topic section. New Custom scripts can be called using the [Script.Execute](#) function. In addition scripts can be copied from one bloc to another block using copy and paste. They can be deleted from a block if they are not being used, and they moved up or down for visual clarity by using the **Alt-Up-Arrow** and **Alt-DownArrow** key sequence.



This completes this topic with the information we need to move on to the step in this tutorial.

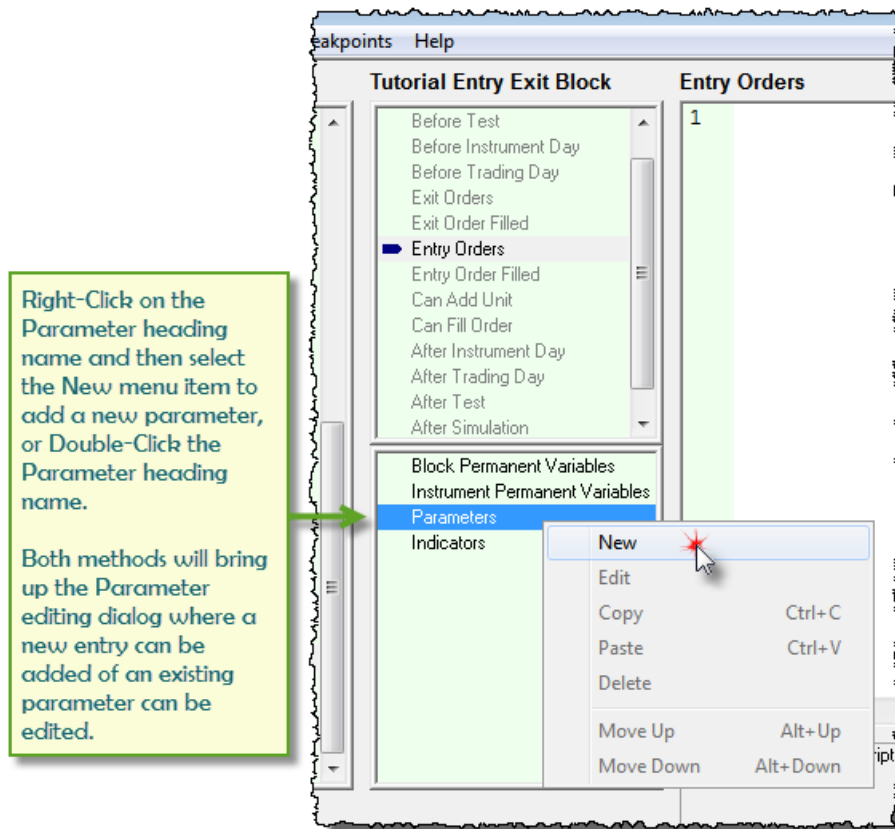
2.2 2. Adding Parameters

Our next step will be to add user menu parameters and scripted rules to the **Entry Orders** script section. Scripts put in the **Entry Orders** script are called for every bar record in every instrument selected in the system's portfolio. Scripts placed in the **Entry Order Filled** script are only executed when an **Entry Order** has been filled.

Exit Order scripts are only executed when an instrument has an active position. **Exit Order Filled** script are only called when an **Exit Order** is filled. Understanding how script are only executed when necessary will help you understand how important it is to place rules into the script section for the purpose that each script section is used, and when it is executed.

User menu parameters are editing fields on the main screen where a selected system shows it parameters. These parameters expose text fields where you will be able to change the values entered to see how the block module changes alter the results of the system. In our simple Moving Average Convergence Divergence system we will need two parameters where we can set the calculation lengths for the MACD's short and long moving averages that will comprise the a MACD indicator.

To create our first parameter right click on the Parameter list item to bring up the Items menu, or use the Items menu once the Parameters list item is selected. In this menu, select the New menu item to create a new Parameter.

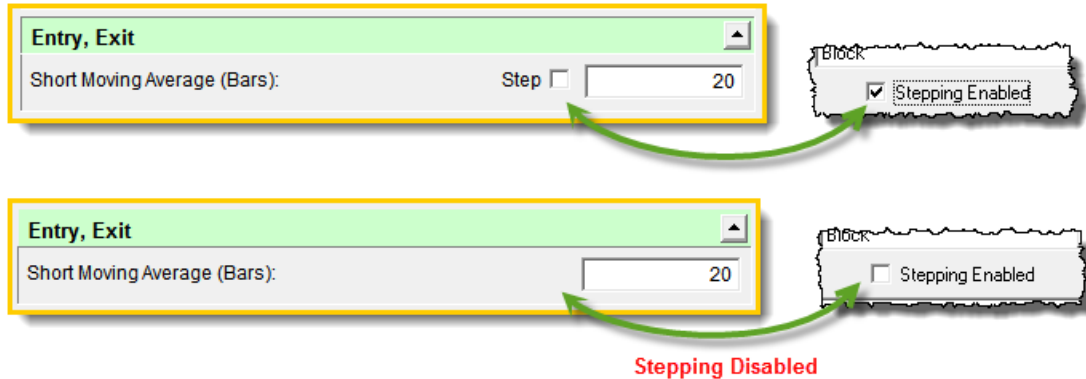


This same menu access process can be used for all items such as Block Permanent Variables (BPV), Instrument Permanent Variables (IPV), Parameters, and Indicators. This brings up a New Parameter dialog Window where new parameters can be added, or existing parameters can be edited.

Parameter Dialog Details:

- Name for Code:** Our first parameter will use the name: `shortMovingAverageBars` for the parameters code name. This name is what we will use when we want to refer to this parameter in a script. Notice how the name is fairly descriptive. By using compound words that are descriptive, the scripted code becomes self documenting making it easier to understand what its intended purpose. In this name we are using the word "**Bars**" to refer to an instrument's price record. For daily data a bar is one daily data record. For weekly records, bar refers to one week record. Parameter code names cannot contain any characters other than alphabetical letters, numbers and an underscore " _ " character, and the name must begin with an alphabetical character.
- Name for Humans:** This is the name users of the block will see. Name can have spaces and special characters, as it is for display purposes only. We normally indicate the unit of measure so it is clear to the user the basis for the parameter. In this case we indicate (**Bars**) so the user will understand the count will be individual data records. For example, a value of 10-bars, for daily data will tell the block to use 10 days or 10 daily records. A weekly data file will use 10-weekly records, and an intraday data file will use 10-intraday records.
- Parameter Type:** We need to let the system know what type of input we are expecting. In this case the input (Bars) will be an integer value. For other parameters we might want to input floating point, percent, or other types.
- Default Value:** Value enter will be the default value. It is also the value displayed when the Blox module is first added to a system. Once added to the system and left in place, the user can change the value and the system's suite file will remember the last value the trader entered. For our block we will use a short moving average length or 20 bars.
- Scope:** Defines whether this value will be available to just this block, to the system, or to the whole testing range of information. For our purposes, we will use the default value of Block.

- **Used for Lookback:** Check this box only when you use this parameter to reference historical values of indicators, or price series records. If this parameter is used as a parameter in an Indicator you do not need to check this box.
- **Stepping Enabled:** Value stepping is controlled by this option. When it is enabled, the value in the parameter can be stepped in an optimization test. In the image pair below the Stepping option is shown how it changes the user's main screen parameter option. When the Stepping option is enabled in the parameter editing dialog, a "**Step**" option is available with the parameter item. When it is not enabled, there isn't a "**Step**" option available and a stepped optimization will not be possible until the parameter's setting is changed.

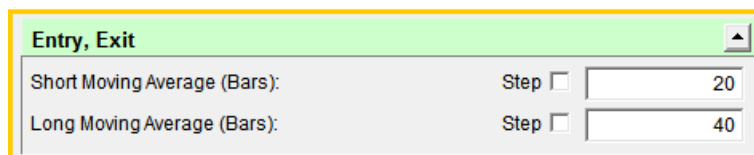
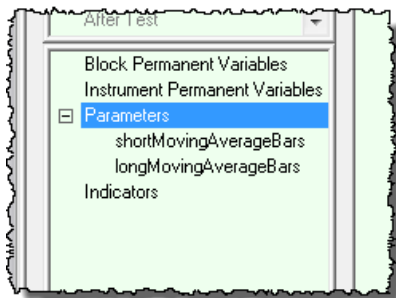


- **Stepping Priority:** This is used to control the order in which the parameter are stepped.

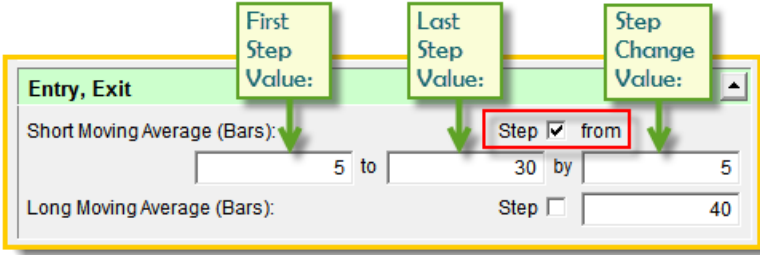
With the above understanding, create the `longMovingAverageBars` parameter, using 40 as its default value and use the same option selection we used for the first parameter.

Once you are finished, press OK. Using the same process, create a .

We now see our two parameters in our Parameter List and in the second image how they will appear when they are connected to a system file and that system file is connected to a Suite on the main screen.



When there is a "**Step**" option associated with a parameter it indicates that parameter can be used in a series of value changes for each step. For example in this next image the Short Moving Average value will be stepped through 6-values starting from a value of 5 to an ending value of 30:

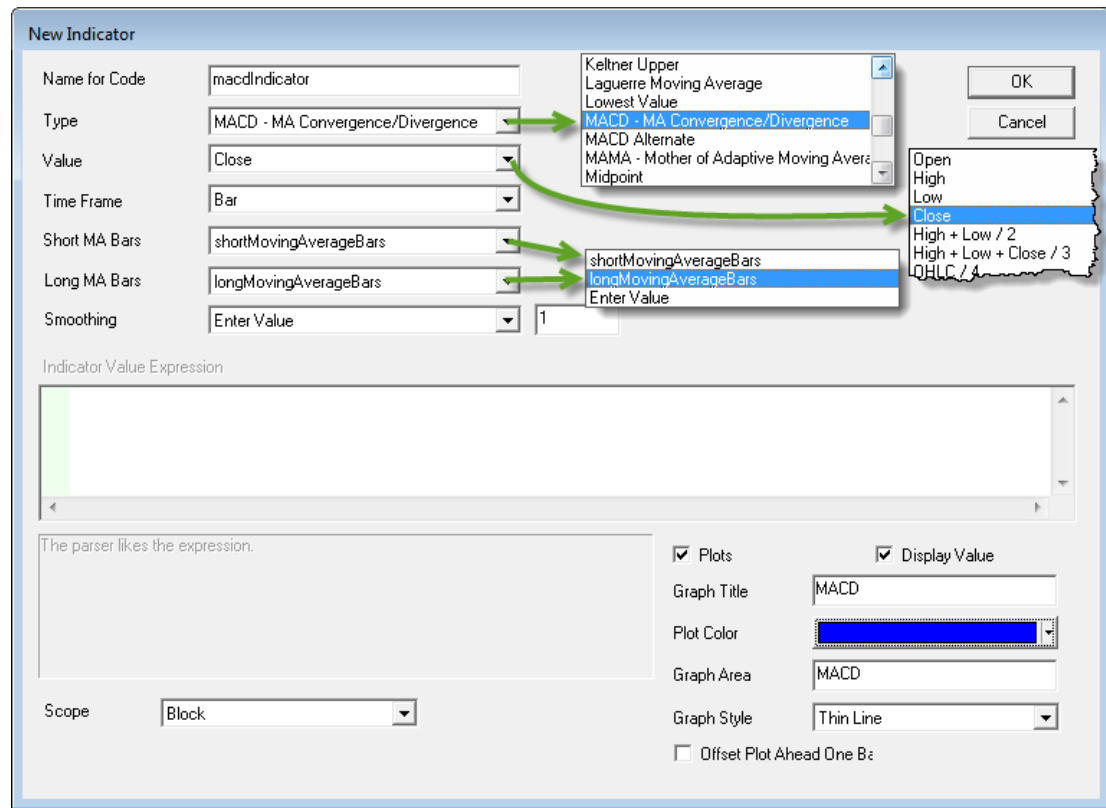


When this system is run through a simulation test it will execute the system through its entire data range listed in the Start and End date period entered. When all the step values have been tested Trading Blox will generate a Performance report showing the results of each stepped value. This tutorial won't step the parameters, but more will be available in other sections of this Help file.

This completes this topic with the information we need to move on to the step in this tutorial.

2.3 3. Adding Indicator

Now we need to create the MACD indicator on which our system will be based. Right click on the Indicators Item and select New. This will bring up the New Indicator dialog box.



- **Name for Code:** Similar to parameters, this is the name you will use to access the value of the indicator from the scripting language. We will call our indicator "**macdIndicator**".
- **Type:** Select "**MACD - MA Convergence/Divergence**".
- **Value:** In this case we use the Close price of each bar record to calculate each moving average. You can select other values from the drop-down list to see how it affects the performance of your system when you are ready to expiration.
- **Short MA Bars:** Select the parameter we setup to hold the short moving average calculation length, which was **shortMovingAverageBars**
- **Long MA Bars:** Select the parameter we setup to hold the long moving average period length, which was **longMovingAverageBars**
- **Plots:** Check this box so the indicator will be plotted on the graph just below the price chart display. Enter the graph area display name, and select a color if you want a different color for the indicator line. Also check the "Display Value" option so you will be able to see the value of the indicator at each price bar location. Indicator value will be displayed in the data window section on the right side of the graph and it will change as you move your mouse cursor across the chart area.
- **Offset Plot by One Day:** Leave this option unchecked. If our system were trading on stops or limits and the indicator displayed over the price bars, this option would show the value at which the next

price bar crossed the the indicator line.

- When you are finished creating our indicator, press OK.

We are going to add one more indicator so the display of the MACD below the price chart area will have a zero reference line displayed on the same MACD Indicator area. This new indicator will be named "**ZeroLine**" and it will the same zero value across the entire graph area.

Use the typed notations on the Edit Indicator dialog to create the indicator and then press OK when you are done.

Edit Indicator

Name for Code: Type: ZeroLine

Type: Select: Calculated

Value:

Time Frame:

Smoothing:

Not Applicable:

Not Applicable:

Indicator Value Expression: Type: zero as a number

Expression looks ok. Type: Zero for indicator label

Type: MACD for Graph Area

Scope:

Plots Display Value

Graph Title:

Plot Color:

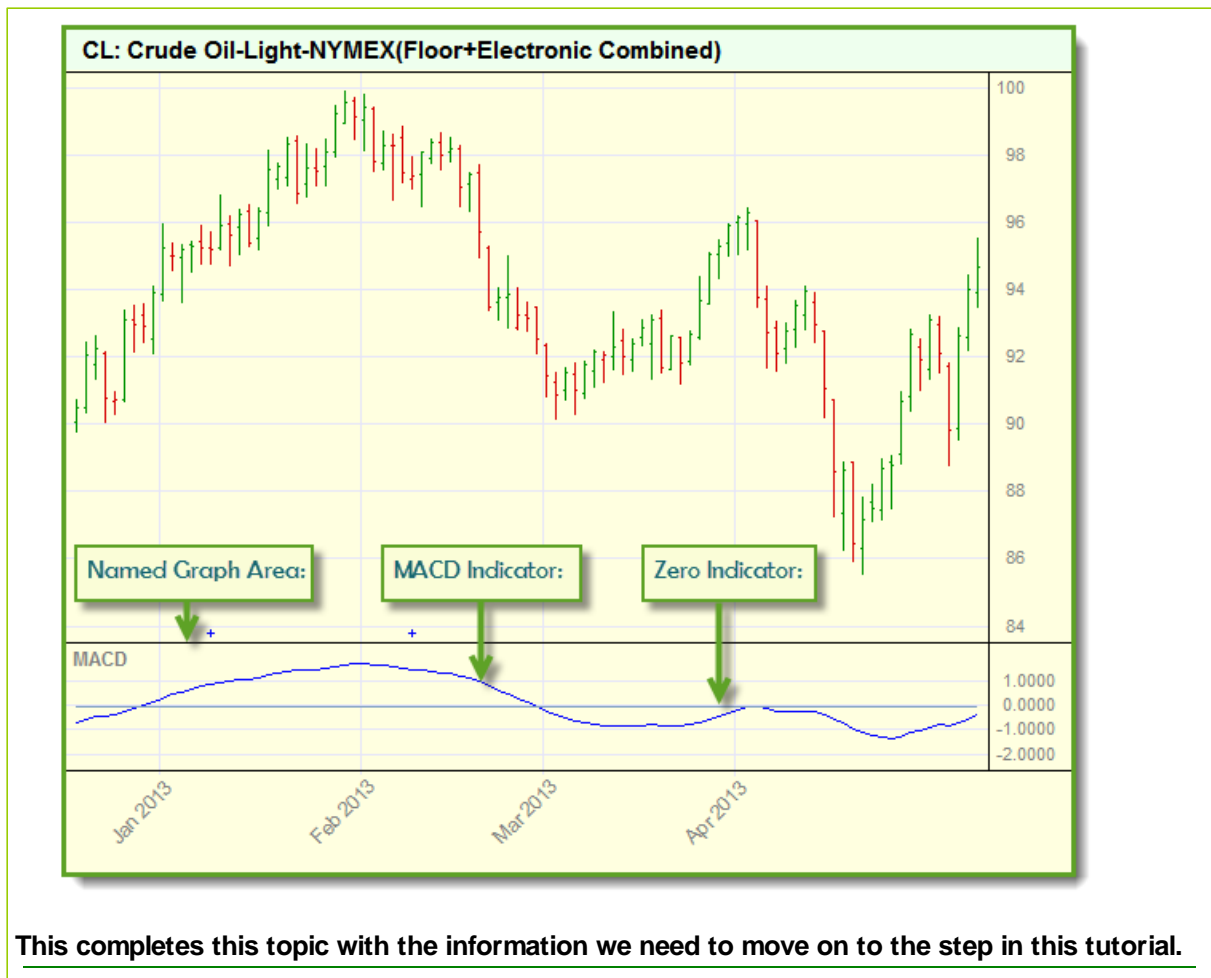
Graph Area:

Graph Style:

Offset Plot Ahead One B:

MACD Indicator Graph Example:

With both indicators created, and once the rest of the tutorial is completed and you are ready to run a test, the chart in this next image will display how both indicators will appear in a chart area:



2.4 4. Entering Code

In this lesson we are going to create the scripting rules that will use this indicator to generate the orders for Long and Short trades. Earlier we mentioned that the Entry Orders script section would execute for each data record in an instrument's file. This is the perfect place to put our entry rule script because we want each price record to be reviewed without failure to ensure we don't miss a signal and create trade that is too late, or goes missing.

Note:

Take a few minutes to understand how Trading Blox Basic uses its language operations by reviewing the tables on this page:

[Operator Reference](#)

- To place our script rules click on the **Entry Orders** script listing the script window on the center-left side of the editing area.
- When the script section appears it will show a blank editing area. Click any place in the editing area so that section has the keyboards focus.
- To create an entry signal we have two parts. The **IF** statement that determines whether to place the trade, and the **BROKER** statement which places the order. We want to say something like, "If the **MACD** is goes positive then enter **LONG** on the **open**. If the **MACD** goes negative then enter short on the **open**."
- This next example show how the above would look if we just need to only provide that information in Blox Basic:

Example:

```

IF macdIndicator > 0 THEN
    ' When the MACD is above 0,
    ' we enter LONG.
    broker.EnterLongOnOpen
ENDIF

If macdIndicator < 0 THEN
    ' When the MACD is below 0,
    ' so enter SHORT.
    broker.EnterShortOnOpen
ENDIF

```

- Notice how the use of comments, colored green and preceded by an apostrophe character " ' " help to make understanding your code rules easier now, and especially later when time has faded some of the details mentioned earlier.
- When our code runs we don't want to continue adding units every day when the **MACD** is positive. To avoid that condition from happening we need to add a little more conditional logic so the rules will know that once we are established in a position, we won't be creating any additional positions in the same direction until that direction changes and the **MACD** changes again. This means we need to put one more piece of logic that requires "If the **MACD** is positive **AND** we are not **LONG** already, **THEN** Buy on the **open**".

- Enter the code in this next example exactly as it appears in this next section into the Entry Orders script section:

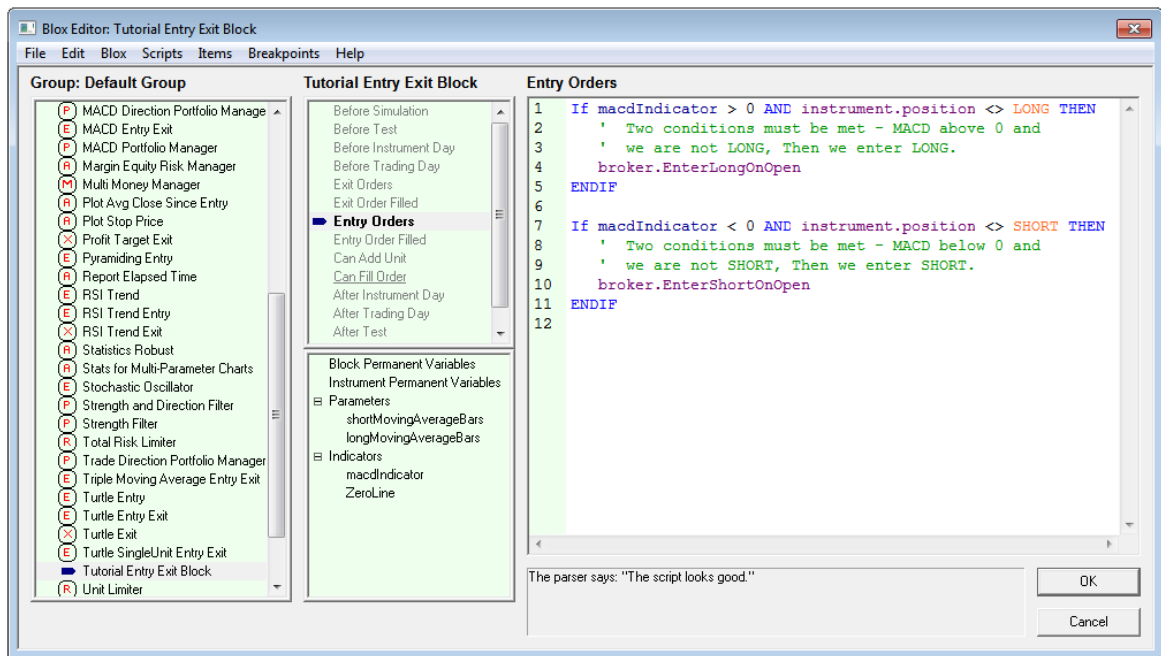
Example (don't type this line):

```
If macdIndicator > 0 AND instrument.position <> LONG THEN
  ' Two conditions must be met - MACD above 0 and
  ' we are not LONG, Then we enter LONG.
  broker.EnterLongOnOpen
ENDIF
```

```
If macdIndicator < 0 AND instrument.position <> SHORT THEN
  ' Two conditions must be met - MACD below 0 and
  ' we are not SHORT, Then we enter SHORT.
  broker.EnterShortOnOpen
ENDIF
```

Example - End (don't type this line):

- What is shown in this last example is all there is to our entry logic. our entry logic. When we view what we've typed it should look like what is shown in this next image:



Our block module now has the **MACD** as an indicator installed along with the parameters needed to control and we also have the rules needed to generate orders entered into the **Entry Orders** script section.

This closes this lesson and we are ready to assemble the modules we will need to create a system. Press the OK button so Trading Blox returns us back in the main screen.

Links:

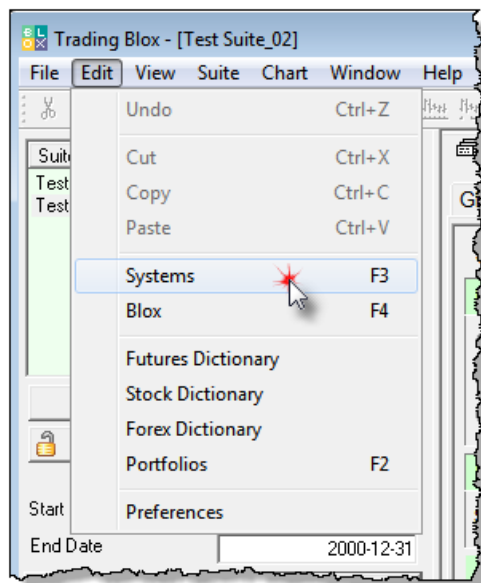
[Operator Reference](#)

This completes this topic with the information we need to move on to the step in this tutorial.

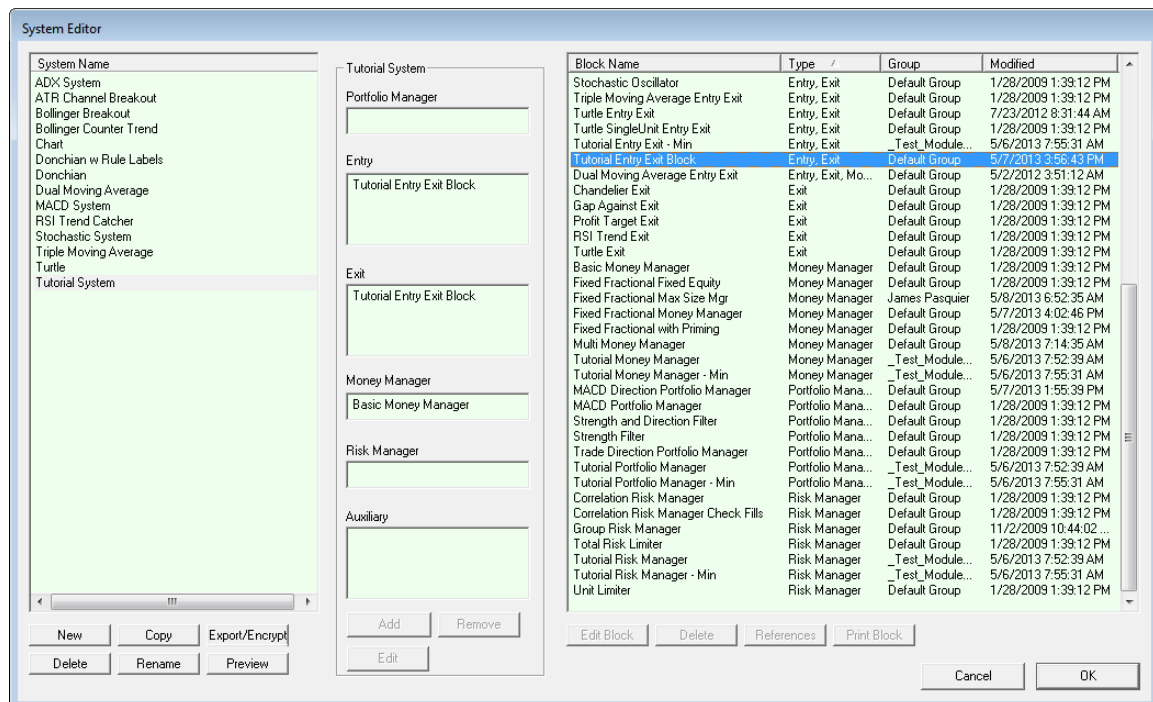
2.5 5. Building A System

A Trading Blox system is a group of selected Blox modules that are available in the System Editor's module list. Each trader who builds their own system structure selects the specific modules they need to achieve the system's intended goal.

- To get started with building our first system we must enter into the System Editor by selecting System menu item under the main screen's Edit menu, or by pressing F3.

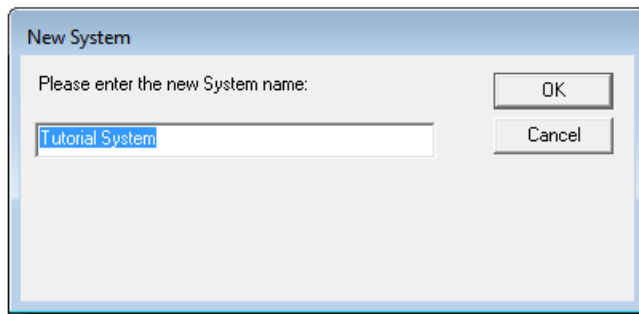


- All we need to do now is create and name a new System, and then include our Entry Exit Blox in a System. When this System Editor dialog appears we will create a new system list and save it:



- Click on the "New" button on the lower left side of the System Editor, and enter the name "**Tutorial**

System" into the New System dialog, and then click the OK button:



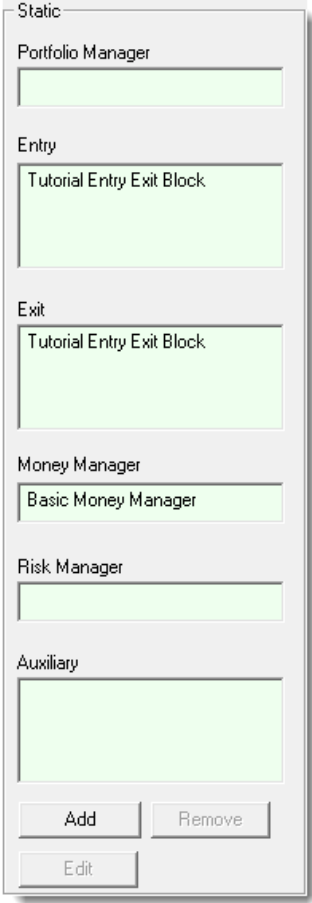
- We now have a new system list named "Tutorial System" where we can add our new Entry Exit Blox.
- Look in the block list on the right side of the screen and scroll the list until you find the Entry Exit blox and see the "**Tutorial Entry Exit Block**" we created in the earlier lesson sections. When you find our tutorial block, right-click on it so that it will appear in the system section list area.

Triple Moving Average Entry Exit	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
Turtle Entry Exit	Entry, Exit	Default Group	7/23/2012 8:31:44 AM
Turtle SingleUnit Entry Exit	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
Tutorial Entry Exit Block	Entry, Exit	Default Group	5/7/2013 3:56:43 PM
Dual Moving Average Entry Exit	Entry, Exit, Mo...	Default Group	5/2/2012 3:51:12 AM
Chandelier Exit	Exit	Default Group	1/28/2009 1:39:12 PM
Gap Against Exit	Exit	Default Group	1/28/2009 1:39:12 PM

- Next locate the "Basic Money Manager" provided with Trading Blox. Click on it and then right-click the it so that is it also placed into the system listing:

Profit Target Exit	Exit	Default Group	1/28/2009 1:39:12 PM
RSI Trend Exit	Exit	Default Group	1/28/2009 1:39:12 PM
Turtle Exit	Exit	Default Group	1/28/2009 1:39:12 PM
Basic Money Manager	Money Manager	Default Group	1/28/2009 1:39:12 PM
Fixed Fractional Fixed Equity	Money Manager	Default Group	1/28/2009 1:39:12 PM
Fixed Fractional Max Size Mgr	Money Manager	James Pasquier	5/8/2013 6:52:35 AM
Fixed Fractional Money Manager	Money Manager	Default Group	5/7/2013 4:02:46 PM
Fixed Fractional with Priming	Money Manager	Default Group	1/28/2009 1:39:12 PM

- Our System Editor static section list of selected module is shown in the center of the screen and should look like this:



The screenshot shows a vertical list of modules for a system, organized into categories. The categories and their respective modules are:

- Static:** Portfolio Manager
- Entry:** Tutorial Entry Exit Block
- Exit:** Tutorial Entry Exit Block
- Money Manager:** Basic Money Manager
- Risk Manager:** (empty)
- Auxiliary:** (empty)

At the bottom of the list, there are three buttons: "Add", "Remove", and "Edit".

- If by chance it doesn't look like the above, remove what is wrong by right-clicking on the wrong items, locate the items mentioned earlier in this section, and then right-click on them so they appear as this center system list detail shows.

Building System Information:

Each system is created by first naming a new system structure that will be used to keep the required modules grouped.

Selecting modules is made possible by locating the a module in the System Editor's Blox Listing, and then adding it to a system by right-clicking on its name.

Removing a module is made possible by right clicking on the module the trader wishes to remove.

There can only be one instance of the following modules assigned to a system. When the Blox type you want to use in a system already has a Blox name displayed in the center system listing and for that Blox type it must be removed before you will be able to assign the Blox you want to use:

- Portfolio Manager
- Money Manager
- Risk Manager

Once a system is assembled with selected modules it must be saved so it can be available.

When a system is first assigned to a Suite for testing, all the Blox modules will show the default

parameters given to the module's parameter. These values can be changed, and once changed the Suite structure will remember the new values until they are changed again.

Removing and adding another module will loose the previous module's edited parameter settings, and the new module will be loaded and display its default value until you change them.

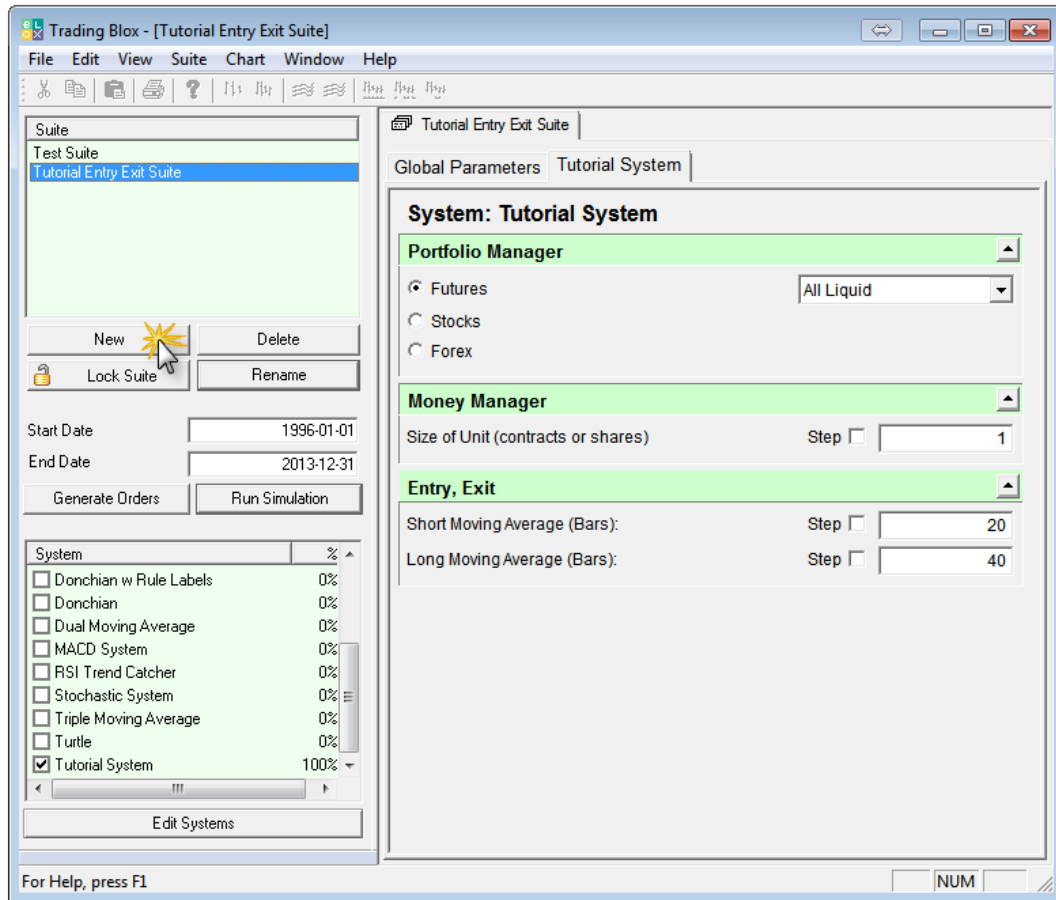
Entry and Exit, and Auxiliary module can have multiple modules of that type listed.

This completes this topic with the information we need to move on to the step in this tutorial.

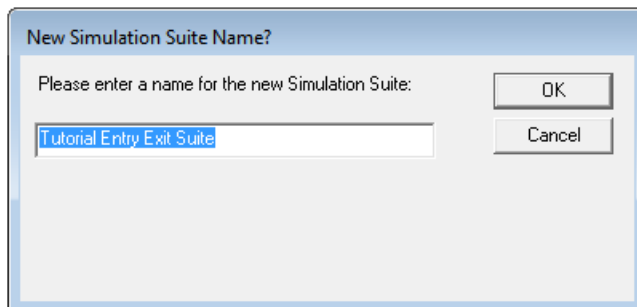
2.6 6. Creating A Suite

All simulation suites are created on this main menu screen.

- To create a new Suite name to match our Tutorial Entry Exit System, click on the **"New"** button at bottom of the Suite listing in the upper right-hand area of the main screen:

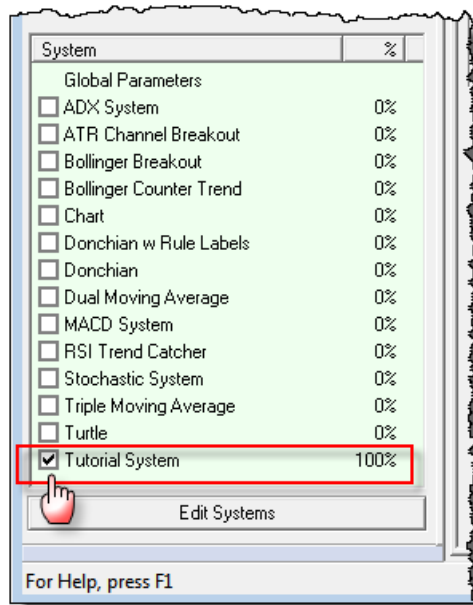


- A new simulation **"Suite Name"** dialog window will appear. Remove the name shown, and enter **"Tutorial Entry Exit Suite"** and then press the **"OK"** button:



- We now have a Suite structure which we can use to attach our new **"Tutorial System"** module. To assign a system module to a Suite it must be found and then its option selection box must be enabled.

- To attach a system module be sure the new simulation suite item in the Suite list has the focus by clicking on it.
- In the lower-left area of the main screen there locate the "**Tutorial System**" item and then enable its check-box so that it will be attached to the "**Tutorial Entry Exit Suite**" simulation suite:



- Your main screen should look close to what is displayed at the top of this lesson where the main screen is shown.
- Notice how our default values show up.
- We can press Run Test to see how our system works!
- Try stepping through many different values and combination of values to find the optimal robust set for this system.
- You can change the portfolio that is used in the portfolio manager, create your own portfolios, and change the global parameter by clicking on the Global Parameters Tab.

This completes our "[Creating a New System](#)" lesson stage.

Section 3 – Improving a New System

In our previous ["Create a New System"](#) section we assembled a basic entry and exit system using the MACD indicator to create a signal of when to use a Long-Entry or Short-Entry order.

All orders generated with our simple Tutorial System were sized using a fixed quantity size of 1-contract. We used a fixed quantity order sizing process because we didn't have any risk information without simple entry order methods. There was not any risk information because there are no protective exit prices on which to base how much risk a single contract position will have without knowing a reasonable unfavorable price move distance on which to estimate the position loss as a percentage of account value.

In some cases this meant that some orders created a small amount of risk and account leverage, and some orders created a larger amount of risk with a large leverage ratio. Leverage increases the utility of the value of an account, but it can be the reason why the account is depleted quickly and a trading strategy fails.

In this second section of the tutorial will introduce protective position pricing order and how they are placed and applied. We will also give an overview of how risk is view and adjusted along with an explanation of the three primary order sizing modules included with Trading Blox. As the lessons expand understanding we will create a method for measure price volatility and show how three different ways to size an entry order. Adding stops to our new system is the goal of this second section and it will proceed to show how to modify our new system so it can trade with less risk.

Tutorial Steps & Topics:

Lesson:	Topic Description:
1	Protective Position Pricing
2	Copy System Items
3	Protective Exit Orders
4	Entry Order Protection
5	Active Order Protection
6	Order Sizing
7	

Links:

[Operator Reference](#)

3.1 Protective Position Pricing

Active positions are not required to have a protective price order entered into the market to be successful. However, positions without a protective price often have larger point losses than would have happened had the position been protected with an active protection order.

When we created the MACD Entry and Exit system in our first tutorial lessons, we didn't use any protective price orders to control how much a failed position would lose. That decision was intentional so we could keep the process of creating a simple trading system uncomplicated. In this lesson we are going to add protective orders as part of our order process and use that initial protective price as a maximum risk amount for each order.

Price Protection Methods:

Determining how to protect a position is best determined by the trader's perception of results from using various protective price calculation methods. For example consider these ideas for calculating a protective price value:

- Money Amount Offset
- Percentage Offset
- Volatility Range Offset

Protective prices are best placed close enough to the current market prices to prevent unreasonable losses, but they must also not be placed too near them so the normal range market's price oscillations interferes with the positions opportunity for larger gains. When prices interfere with the trends normal movement the protective price will prematurely terminate the position leaving the opportunity that position might have provided.

Money Amount Offset:

Protective price is determined by establishing fixed monetary amount and using that value to discover how many points to offset the price. In most cases this protection method is used with a fixed quantity size of 1. When more than one contract or share is required the fixed amount can be the risk amount of each contract, or it can be the total risk when the quantity is greater than 1.

Example:

Fixed quantity position of 1 Long position share for a symbol priced at \$100.00 the offset protective price will be determined multiplying \$1,500 by \$0.01 to discover the protective point offset of \$15.00 and a protective price of \$85.00.

For a fixed quantity of 2-shares of a \$100 market Long position will divide the \$1,500 by 2 to get \$750 protection amount for each share. With \$750 multiplied by \$0.01 the protective offset amount will be \$7.50 subtracted from \$100 to get the protective price of \$92.50 for the total position quantity.

Futures use a monetary basis determined by their contract size and have Big-point value that is used to find the protective price. For example a Crude Oil contract priced at \$100 will use a Big-Point value of \$1,000. A monetary amount offset of \$1,500 divided by \$1,000 determines the a contract protection offset must be placed 1.50 points in price change. This 1.50 point offset for a Long position priced at \$100 will place the protection amount at \$98.50.

Percentage Offset:

Price percentage offsets are frequently used with equity type trades, and they are calculated by

applying a percentage rate to the purchase price of the share. A percentage can be applied for portions instead of the total position quantity by creating various protective price orders for some of the quantity of shares in the trade.

Example:

A Long position of 1 symbol-share will exit the position when a 10% drop in price is printed in the exchange. Using our \$100.00 share we get a \$10.00 price offset with out 10% price drop rate to establish a protective price of \$90.00.

Volatility Range Offset:

Future contract trades often use a measurement of recent market volatility to determine likely price range that a position might experience. An advantage in using a volatility measure is how it gets the trader away from using their wallet size for making critical decision. Instead a volatility approach allows the trader to get an estimate of the likely price range the market is experiencing at the time of entry to estimate where a protective price can be established without it being in place that would cause the trade to terminate with a loss from having the protection to close to the market's current price range.

Volatility estimation used for initial price protection placement use a the most recent period of history to get an estimate of the most recent price range. Recent period lengths are often a user available parameter where the number of bars to observe can be adjusted by the trader. With a period length the range of each price bar is then observed There are various methods available that calculate the range of of each price bar and then average that information to get an estimate of price point movement that can be used in helping the trader establish a protective price.

Along with a volatility estimation of price ranges and method for adjusting the volatility average is needed so the size can be can be increased or reduced by the user. Adjust volatility size usually needs a decimal number to create a offset estimate that will work for all the markets in the portfolio.

Trading Blox provides users with a built-in function known as the Average True Range calculation. This concept was published by J.Welles Wilder Jr. in his book "**New Concepts in Technical Trading Systems**" Chapter III, **Volatility Index**. Wilder's Volatility Index measures a price bar range from its high price to low price, but it also includes the previous price bar's close price if including that price into the range would increase the point value of the range. In that same chapter uses the index in an averaging calculation to estimate the price volatility.

Our lesson plan to add protective exit orders will use Avg.True-Range calculation and we will show how to add it to our tutorial system. For now know that when it is applied along with the parameters required to control its calculation and adjust its volatility application in an entry section consider this type of code will be added as this section makes progress:

Example:

```
' Create a protective offset point protective point sizeL
stopWidth = ATR_AdjustRate * AverageTrueRange

' NOTE:
'   ATR_AdjustRate = AvgTrueRange Entry Adjust Protect Rate

' In each entry order section for a Long and Short position
' the application of points changes:

' Create an initial LONG Position protective price
longStopPrice = (instrument.close - stopWidth)

' Create an initial SHORT Position protective price
shortStopPrice = (instrument.close + stopWidth)
```

Example above shows the price adjusting points to create a Long Position protective price below the current market prices, and also for creating a Short Position protective price above the current market prices.

This completes this topic.

3.2 Copy System Items

Before we modify our new system we will make a copy of the blox so we have the original and a modified version we can compare.

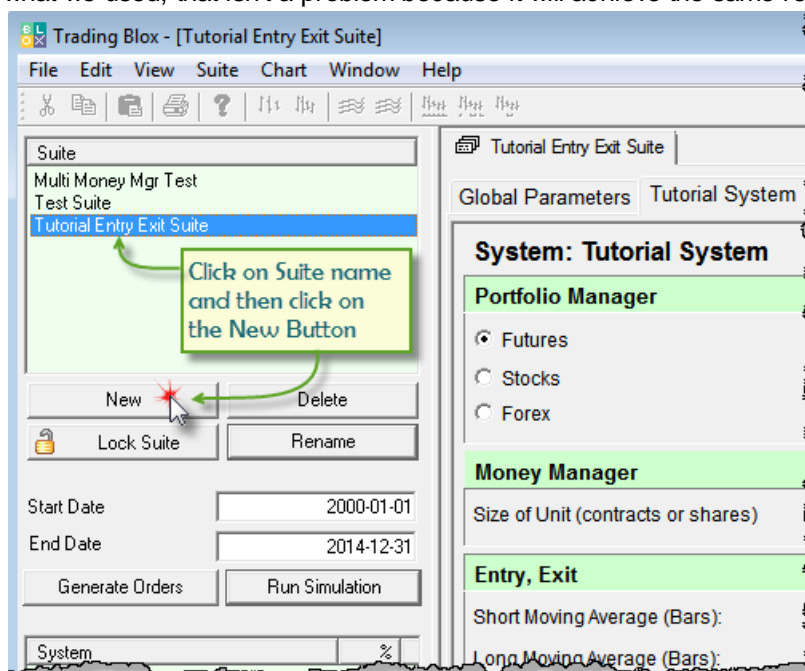
To make a copy of the module we will first start with making a copy of the first tutorial's Suite. This process allows us to create an exact copy of the original tutorial suite Global Parameter Settings along with the a link to the first tutorial's system list we used. We will also make a copy of the system list so we have the same modules as the first tutorial, which will gives us the opportunity learn how to change the modules used in the system list.

When all the original system system items have been copied and changes to what we need we will then add a protective entry prices into our original Entry Orders section for Long and Short new position orders. As that section is completed we will add a code to the Exit Orders section so the original orders are allowed to be active throughout the life of the trades.

Making A Suite Copy:

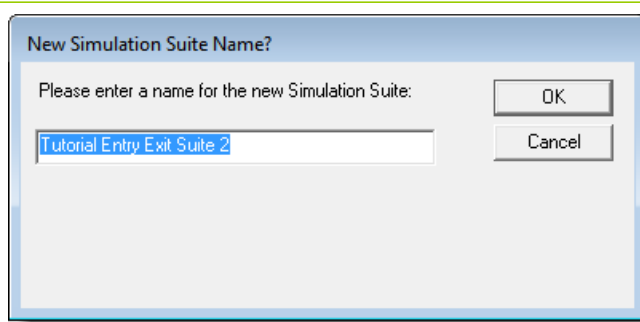
Creating a copy of an existing suite is the easiest way to retain the Global Parameter Settings you have established for a previous suite's simulation controls. In Trading Blox the "New" button acts will copy the selected suite highlighted. It will also retain the system names selected.

Click on the suite name you created in our first tutorial section. If your suite name is different than what we used, that isn't a problem because it will achieve the same result.



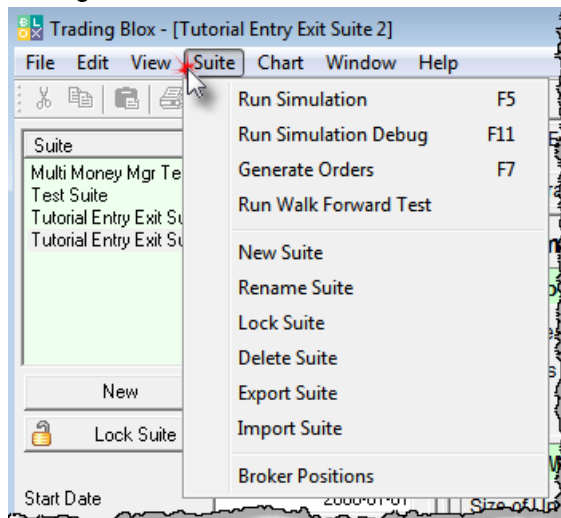
Suite Copy Steps

When the "New" button is clicked the suite simulation dialog window will open and it will automatically add the number 2 at the rightmost side of the name displayed. We are going to use the name with the #2 suffix and will click the OK button now.



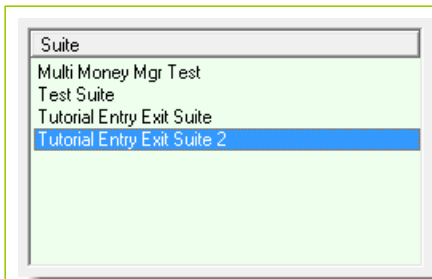
When the new suite name appears in the Suite listing section there will be two suites that will be show the same Tutorial System name is selecting the same system. We are going to make a copy of the original Tutorial System so we can make changes to the Tutorial's Entry and Exit Orders sections, but in actual trading there is often a good reason for a more than one suite to select the same system. A good reason to have two suites select the same system is based in the suite's ability to remember the Global Parameter and System Parameter settings based upon those previously used in each of the suite names. Suites are where the user settings that affect the systems operation are preserved until they are changed. If a system selection is changed, the parameters for that system will be lost, but can easily be entered again. Suite's portfolio selection will also be lost when a selected system is removed.

In Trading Blox Main Suite menu provides a complete range of methods for working with new or existing suites:

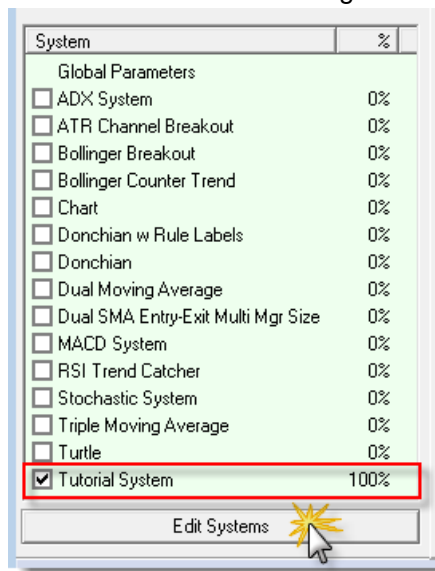


Making A Suite Copy:

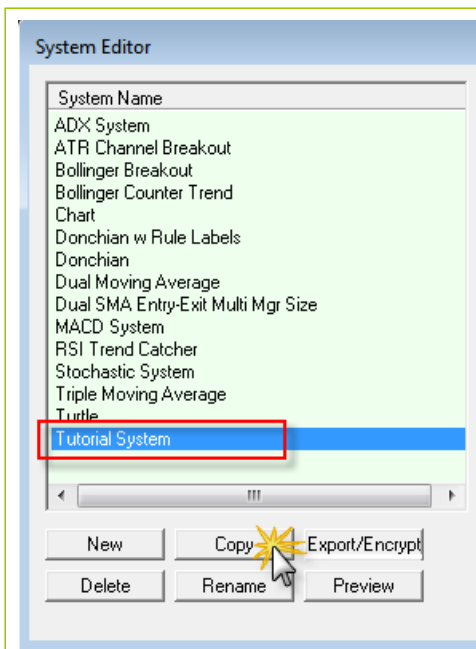
Your copy of our first tutorial suite name should be highlighted. If it isn't click on it to see the selected system.



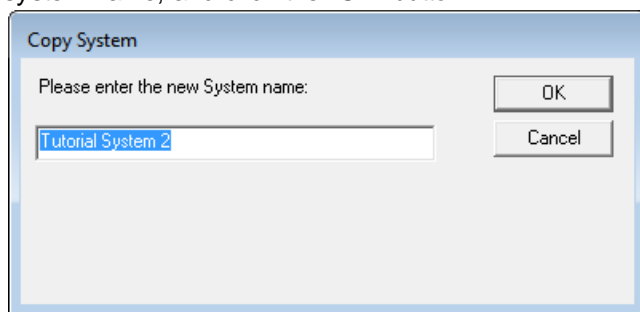
In the System selection list your first tutorial system will be selected. If you used a different system name that won't matter as long as the name you used is selected:



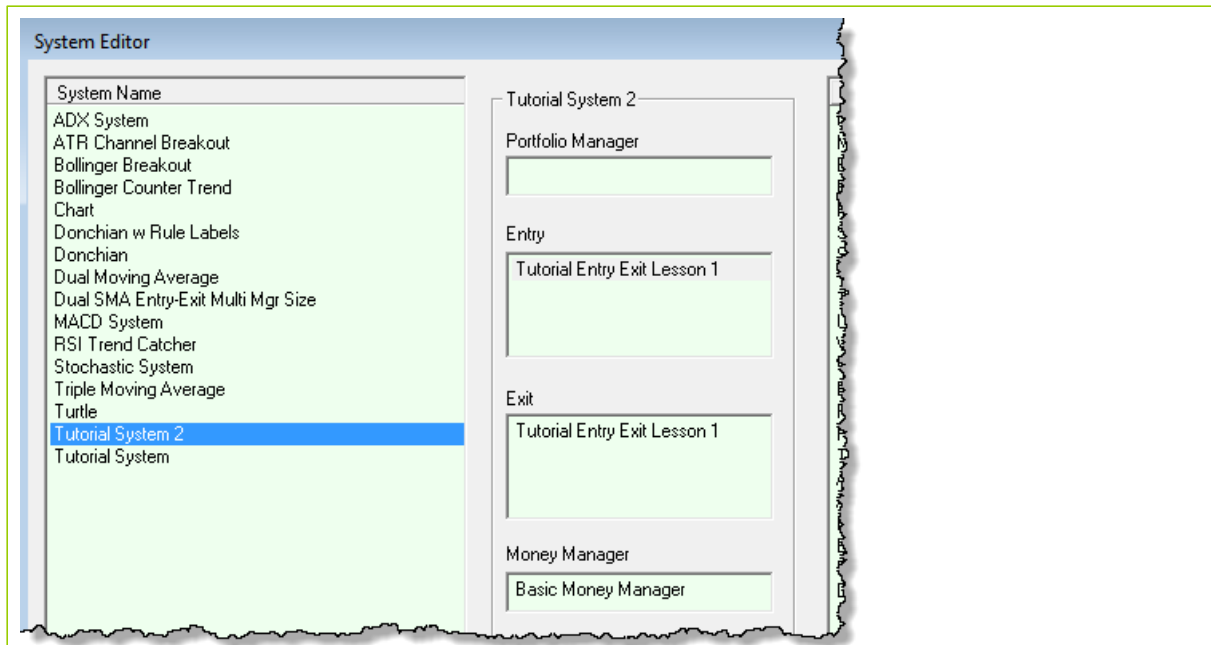
Click on the "Edit System" button at the bottom of the System name selection list so the System Editor dialog opens:



Our first tutorial system name should be highlighted. If it isn't, select it now and then press the "Copy" button so the system copy dialog appears. When a selected system is being copied the Copy System dialog will append a number 2 to the selected system name. We are going to accept the add 2 to the system name, and click the "OK" button.



When the copy system dialog closes the new system name will be listed in the System Name list.



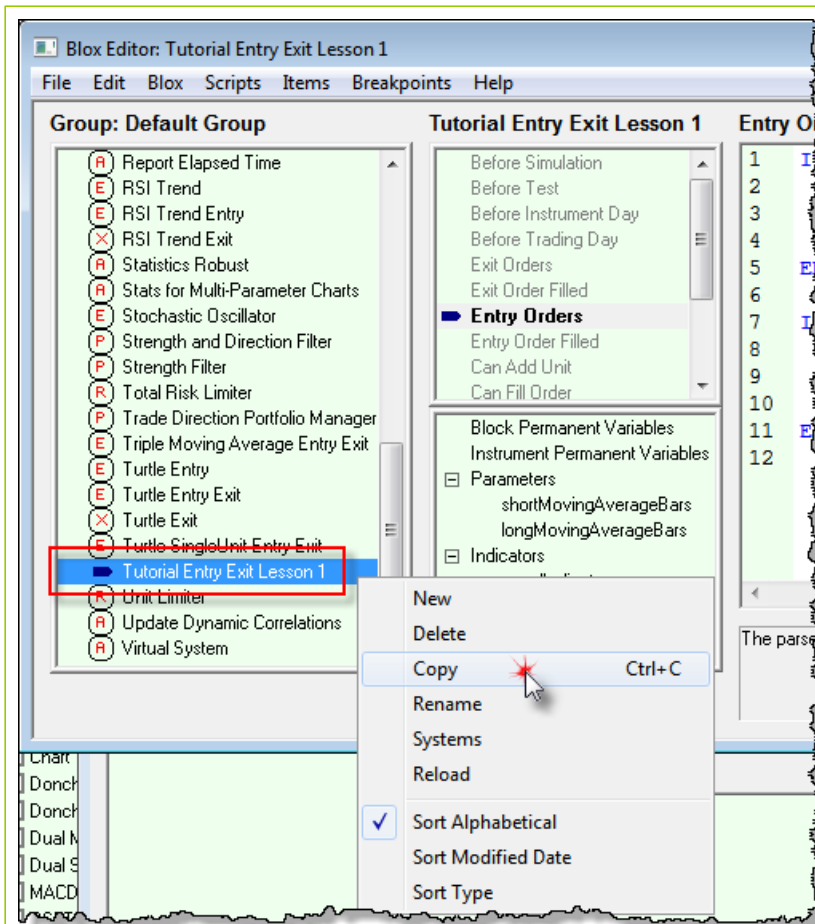
Our new system list will need changes that we don't want in our original entry exit logic. This means we will create a copy of the original tutorial entry exit blox so we can make changes and use the modified module in our new system and suite.

Copy Tutorial Entry Exit Blox:

Blox copies any of the blox in Trading Blox are simple and fast.

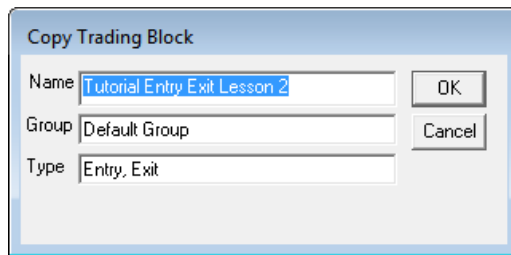
To open the Blox Editor so that the editor will show our original entry exit blox highlighted, double-click on the Entry or Exit name listing in copy of the Tutorial System 2 module list.

When the editor appears it will look something like this next image where the Tutorial Entry Exit Lesson1 blox module is highlighted.



If you used a different name, click on that blox name and when right-click on it using the right-mouse button.

Menu shown in the image above will appear. Click on the Copy menu item so the Copy Trading Block dialog will appear. It should appear with the same name, but with a 2 appended on the right side of the name.



This name works for us. If you want another, make your changes and then click the "OK" button.

Trading Blox - [Tutorial Entry Exit Suite 2]

File Edit View Suite Chart Window Help

Suite

- Multi Money Mgr Test
- Test Suite
- Tutorial Entry Exit Suite
- Tutorial Entry Exit Suite 2**

Remove check mark from our previous system name:

- RSI Trend Catcher 0%
- Stochastic System 0%
- Triple Moving Average 0%
- Turtle 0%
- Tutorial System 2 0%
- Tutorial System 100%

Enable check mark on our new system name:

- RSI Trend Catcher 0%
- Stochastic System 0%
- Triple Moving Average 0%
- Turtle 0%
- Tutorial System 2 100%
- Tutorial System 0%

System Editor

System Name

- ADX System
- ATR Channel Breakout
- Bollinger Breakout
- Bollinger Counter
- Chart
- Donchian w Rule
- Donchian
- Dual Moving Ave
- Dual SMA Entry-Exit Multi Mgr Size
- MACD System
- RSI Trend Catcher
- Stochastic System
- Triple Moving Average
- Turtle
- Tutorial System 2
- Tutorial System

2. Click on previous system name so it appears in the Block Name list:

1. Click on new system name:

3. Right-Click on name, or press Remove button below so old system removed from the new system list:

4. Right-Click on new system name to add it to the new system list:

Block Name	Type	Group	Modified
Chart Margin Equity Ratio	Auxiliary	Default Group	1/28/2009 1:39:12 PM
Margin Equity Risk Manager	Auxiliary	Default Group	4/3/2013 12:31:43 PM
Plot Avg Close Since Entry	Auxiliary	Default Group	1/28/2009 1:39:12 PM
Plot Stop Price	Auxiliary	Default Group	1/28/2009 1:39:12 PM
Report Elapsed Time	Auxiliary	Default Group	3/7/2012 8:30:26 AM
Statistics Robust	Auxiliary	Default Group	1/28/2009 1:39:12 PM
Turtle	Entry, Exit	Default Group	3/6/2011 9:19:30 AM
_Test_Modul...		Default Group	5/6/2013 7:52:39 AM
_Test_Modul...		Default Group	5/6/2013 7:52:39 AM
Walk Forward		Default Group	2/10/2011 11:38:12 ...
Walk Forward		Default Group	3/5/2010 3:25:22 PM
Walk Forward		Default Group	1/28/2013 1:33:46 PM
_Test_Modul...		Default Group	4/27/2013 11:02:12 ...
Pyramiding Entry	Entry	Default Group	1/28/2009 1:39:12 PM
RSI Trend Entry	Entry	Default Group	1/28/2009 1:39:12 PM
Turtle Entry	Entry	Default Group	1/28/2009 1:39:12 PM
ADX Entry Exit	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
ATR Channel Breakout Entry Exit	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
Bollinger Breakout Entry Exit	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
Bollinger Counter Trend Entry Exit	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
Donchian Smart Fills	Entry, Exit	Default Group	1/7/2013 8:13:27 AM
Dual Average Entry-Exit	Entry, Exit	Default Group	6/8/2013 2:42:58 PM
MACD Entry Exit	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
RSI Trend	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
Stochastic Oscillator	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
Triple Moving Average Entry Exit	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
Turtle Entry Exit	Entry, Exit	Default Group	7/23/2012 8:31:44 AM
Turtle SingleUnit Entry Exit	Entry, Exit	Default Group	1/28/2009 1:39:12 PM
Tutorial Entry Exit Lesson 1	Entry, Exit	Default Group	6/8/2013 3:19:16 PM
Tutorial Entry Exit Lesson 2	Entry, Exit	Default Group	7/12/2013 8:59:13 AM
Dual Moving Average Entry Exit	Entry, Exit, Mo...	Default Group	5/2/2012 3:51:12 AM
Chandeller Exit	Exit	Default Group	1/28/2009 1:39:12 PM
Gap Against Exit	Exit	Default Group	1/28/2009 1:39:12 PM
Profit Target Exit	Exit	Default Group	1/28/2009 1:39:12 PM
RSI Trend Exit	Exit	Default Group	1/28/2009 1:39:12 PM
Turtle Exit	Exit	Default Group	1/28/2009 1:39:12 PM
_Accot_Limiting_Adjustable_Sizing	Money Manager	_Dev	5/26/2013 10:57:21 ...
_Fixed Rate Unit Sizing Mgr Plus	Money Manager	_Dev	5/26/2013 1:08:20 PM
Basic Money Manager	Money Manager	Default Group	1/28/2009 1:39:12 PM

Buttons: New, Copy, Export/Encrypt, Delete, Rename, Preview, Add, Remove, Edit, Edit Block, Delete, References, Print Block, Cancel, OK

This completes this topic.

3.3 Protective Exit Orders

This topic show how to add a protection exit price with a new order.

It will also show how to create a volatility measure that can be used to determine how to price a protective exit for the price bar of market entry.

New Entry Order Protection:

New entry orders and active positions can have a protective price order to help the trader limit adverse price move losses. New protective price orders are most often created during the new entry creation using a [Broker Object](#) function that provides a place where a protective exit price can be provided. In most cases new entry orders are created in the **Entry Orders** script section using a Broker function with that contain "**EnterLong...**" or "**EntryShort . . .**" with the order's execution type text appended to the Broker's function name.

Example:

```
broker.EnterLongOnOpen( exitPrice )
```

Entry orders filled by the market and not closed because their protective price wasn't enabled, will appear as an active position available for the next test bar.

Protective orders will have their protective price preserved in the `instrument.unitExitStop` property. While the price is preserved there won't be a protective order for the next test bar unless the system generates a new protective price. It can easily access the entry order's protective and use that value, or it can analyze how the market has changed and use a new protective price.

When the test bar time ends, the system will need to generate a new protective price order when the system is dependent upon position protection to achieve its performance goals. Maintaining protective price orders is a simple process once the system is operational because of how each active position is processed in the **Exit Orders** script section. Only active position will cause Trading Blox to execute the **Exit Orders** script section and Long and Short trade direction orders can be contained within the same script section.

Active positions required to have a protective order in place for each bar of the trade must generate a new protective order after the close of the last trade bar. This is needed because all orders in Trading Blox are considered "Day-Orders" which expire after the close price is printed by the exchange.

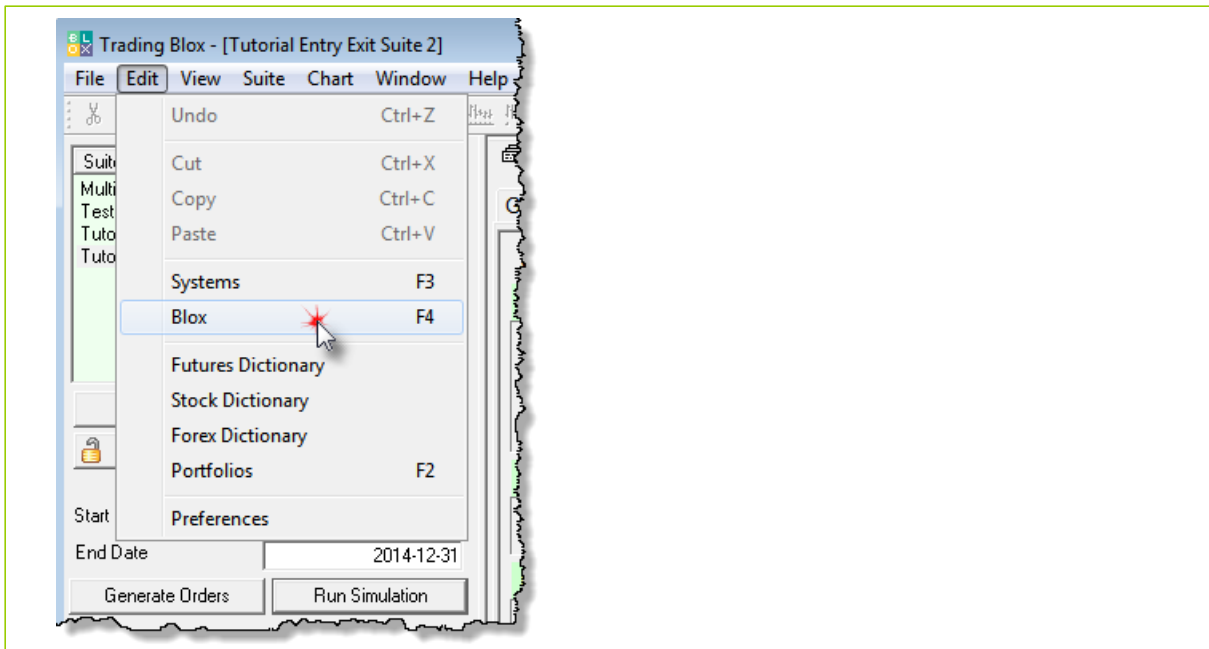
Protective Price Method:

Our example system will use a volatility based point spread for a new entry order in the Futures market. Our approach will use the Average True Range calculation will generate a volatility estimate from a period of recent prices. This volatility estimation is one more popular methods for estimating the offset for a protective price. This calculation requires a period parameter to inform it how much recent price history to include in it calculation. It in most cases also works better with a second parameter that allows the trader to adjust the size of volatility points to a larger or smaller value to fit in with the needs of a system.

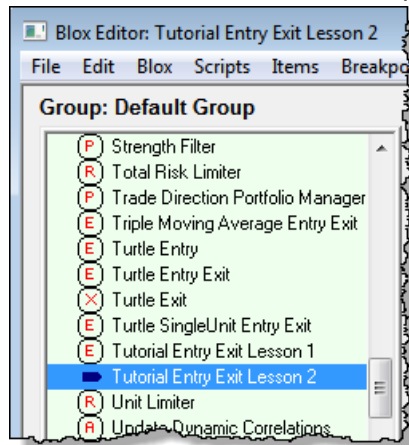
To get started we will first add two parameters in the parameter section of our recently copied Entry Exit block:

First parameter will be called `atrLength` to our parameter section. Our second parameter will be named `atrAdjustRate` to handle the volatility point adjustment.

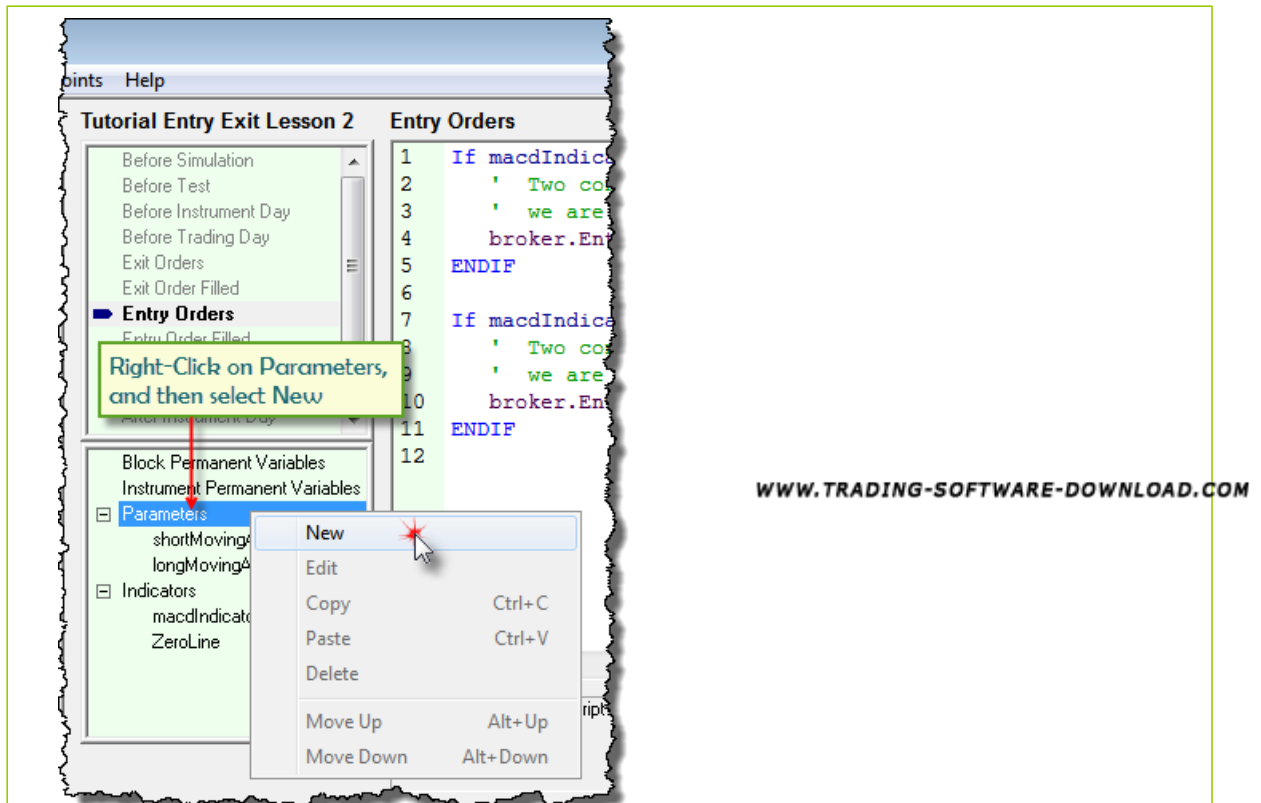
- To create this parameter, open the **Blox Editor** by using the menu item **Blox**, or pressing **F4** on your keyboard:



- When the **Blox Editor** opens locate the new copy we made of our **Tutorial Entry Exit Lesson 2** and select it so we are sure it is displayed as the active module in the **Blox Editor**:



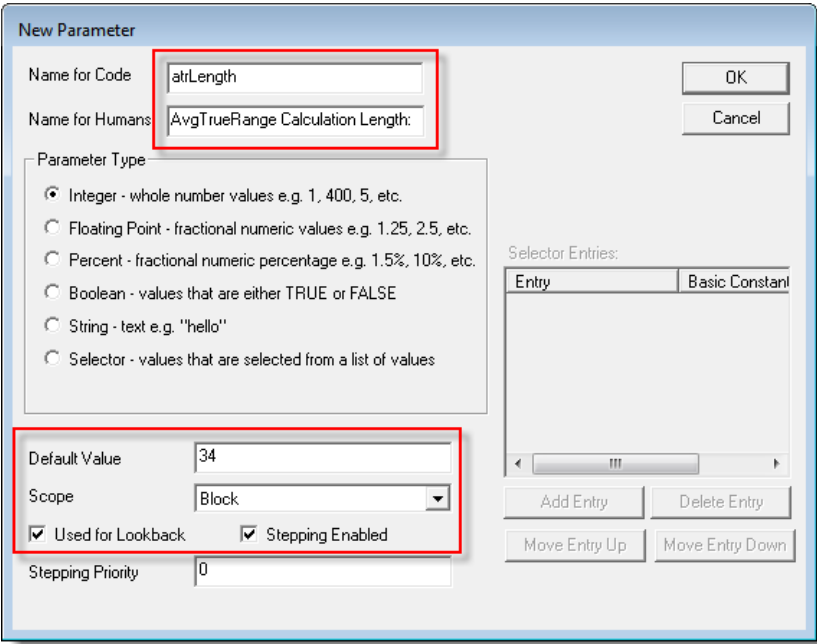
- Click on the word **Parameters** in the lower list area in the center section of the Blox Editor:



Right-Click on Parameters, and then select New

WWW.TRADING-SOFTWARE-DOWNLOAD.COM

- When the parameter dialog appears, add the `atrLength` variable name in the "Name for Code" textbox.
- Next, in the "Name for Humans" textbox enter the description you want to see so you know the parameter is to change the length of the **Avg. TrueRange** calculation period length.



New Parameter

Name for Code: atrLength

Name for Humans: AvgTrueRange Calculation Length:

Parameter Type:

- Integer - whole number values e.g. 1, 400, 5, etc.
- Floating Point - fractional numeric values e.g. 1.25, 2.5, etc.
- Percent - fractional numeric percentage e.g. 1.5%, 10%, etc.
- Boolean - values that are either TRUE or FALSE
- String - text e.g. "hello"
- Selector - values that are selected from a list of values

Default Value: 34

Scope: Block

Used for Lookback Stepping Enabled

Stepping Priority: 0

OK Cancel

Selector Entries:

Entry	Basic Constant

Add Entry Delete Entry

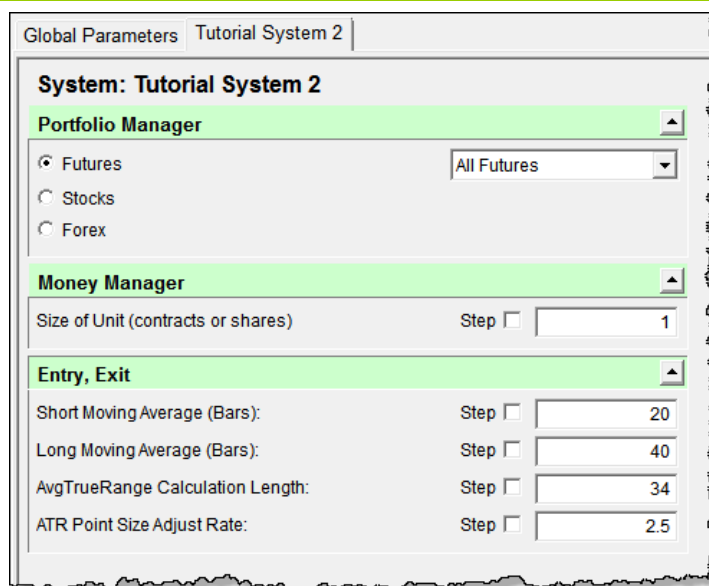
Move Entry Up Move Entry Down

- At the bottom of the dialog, enter a value that is greater than **1** or the value listed in the "**Default Value**" textbox field. Value you enter into this field will be the default value displayed when this module is first added to a system list. Once the module is displayed in the menu area the value used by the trader can be changed to another value. When the default of any parameter will be retained by the Suite information process so that it is available the next time the software is run.
- Follow the process used for our first parameter addition so we can add our second parameter, `atrAdjustRate`, to control the size of the protective offset points. This second parameter will be a Floating Point type, and we will use a default value of 2.5 as the amount of adjustment we will want when we add our module to a new system.

The screenshot shows the 'New Parameter' dialog box with the following details:

- Name for Code:** atrAdjustRate
- Name for Humans:** ATR Point Size Adjust Rate:
- Parameter Type:** Floating Point (selected)
- Default Value:** 2.50
- Scope:** Block
- Used for Lookback:**
- Stepping Enabled:**
- Stepping Priority:** 0

Our volatility estimation and adjustment parameters are now a part of our Tutorial System Entry Exit 2 module. Click the Blox Editor OK button so we can see how adding a parameter to a module will appear when it appears in the parameter section of the main menu's system display area:

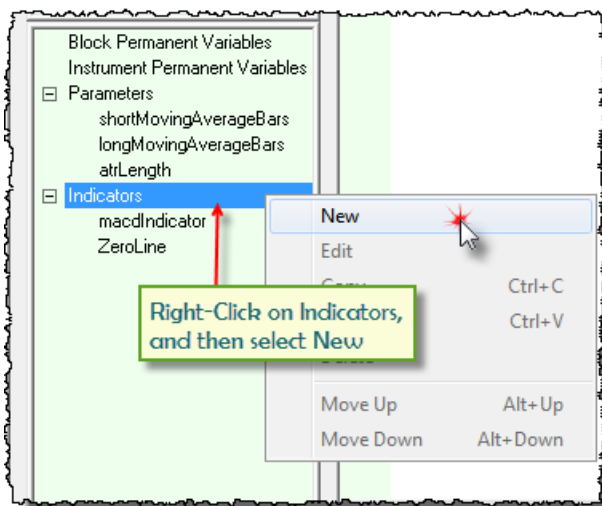


When your screen appears as shown above you will be ready to add the ATR indicator to our Entry Exit module.

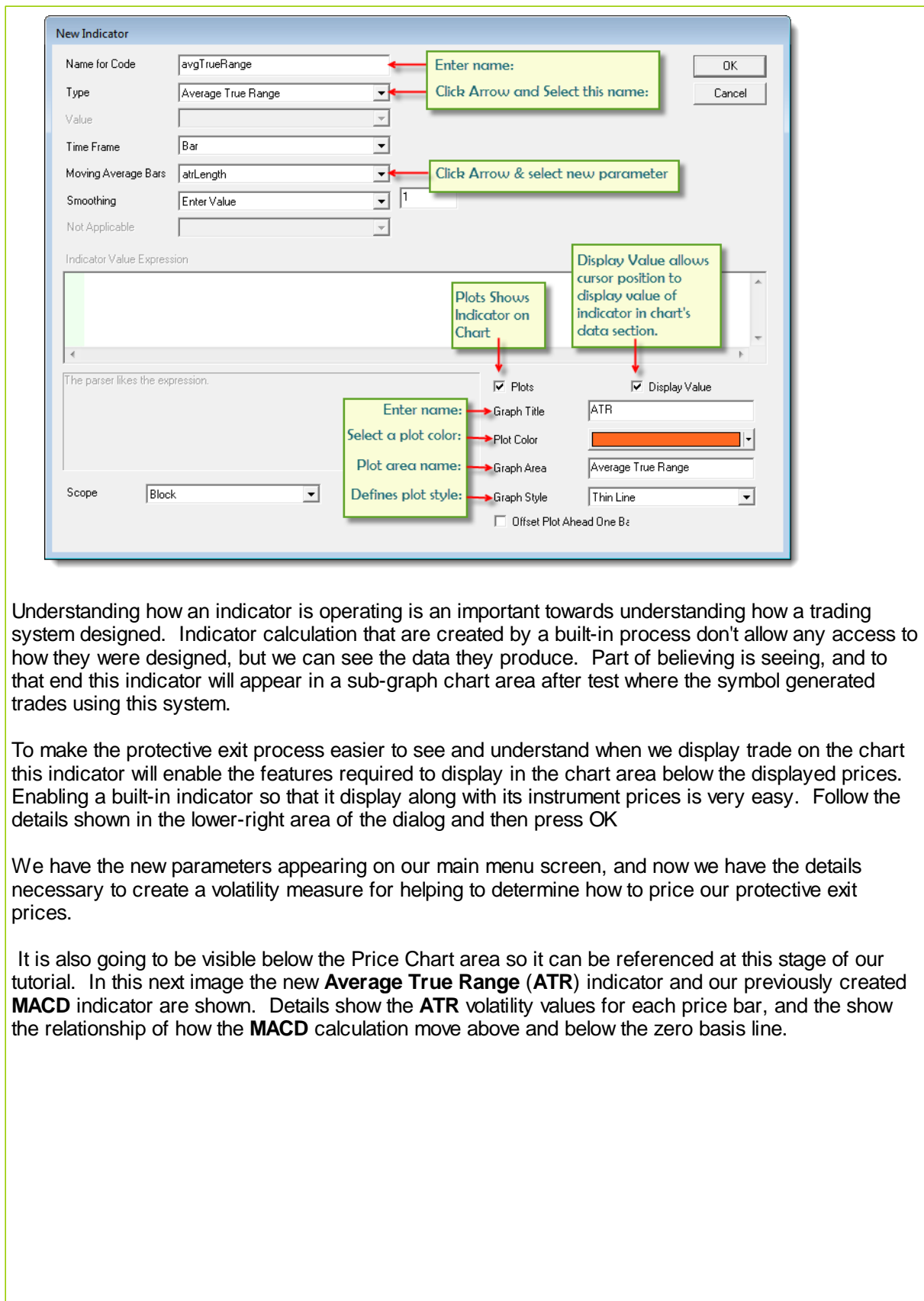
Average True-Range Indicator:

Our process for adding a MACD indicator to the module in lesson 1 will be the same for this new indicator. Indicator name and the selected calculation needed will be different, but the process will be familiar.

- Start with a **Right-Click** on the **Indicators** menu item in the lower center-section of the **Blox Editor**.



- A **New Indicator** dialog will appear where you can enter the name for the code, select the **Average True Range** option, assign the period length parameter, `atrLength`, and enable its information in the dialog's lower-right area so that after a test it can be seen below the price information chart.



Understanding how an indicator is operating is an important towards understanding how a trading system designed. Indicator calculation that are created by a built-in process don't allow any access to how they were designed, but we can see the data they produce. Part of believing is seeing, and to that end this indicator will appear in a sub-graph chart area after test where the symbol generated trades using this system.

To make the protective exit process easier to see and understand when we display trade on the chart this indicator will enable the features required to display in the chart area below the displayed prices. Enabling a built-in indicator so that it display along with its instrument prices is very easy. Follow the details shown in the lower-right area of the dialog and then press OK

We have the new parameters appearing on our main menu screen, and now we have the details necessary to create a volatility measure for helping to determine how to price our protective exit prices.

It is also going to be visible below the Price Chart area so it can be referenced at this stage of our tutorial. In this next image the new **Average True Range (ATR)** indicator and our previously created **MACD** indicator are shown. Details show the **ATR** volatility values for each price bar, and the show the relationship of how the **MACD** calculation move above and below the zero basis line.



3.4 Entry Order Protection

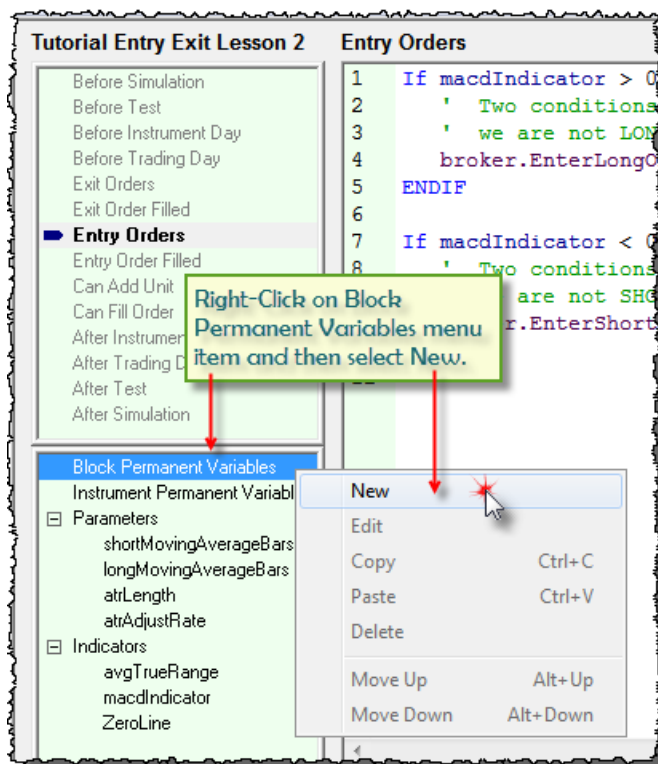
Our **Average True Range (ATR)** indicator is now operating and reporting a volatility measure for each price in the data file after the calculation priming period has primed.

When we create orders they are always based upon the values of price bars that have an **Open**, **High**, **Low** and **Close** price value that considers that price record ready to use. After the last available price bar is provided Trading Blox can use that new information to what it had previously and update the calculations of its indicators.

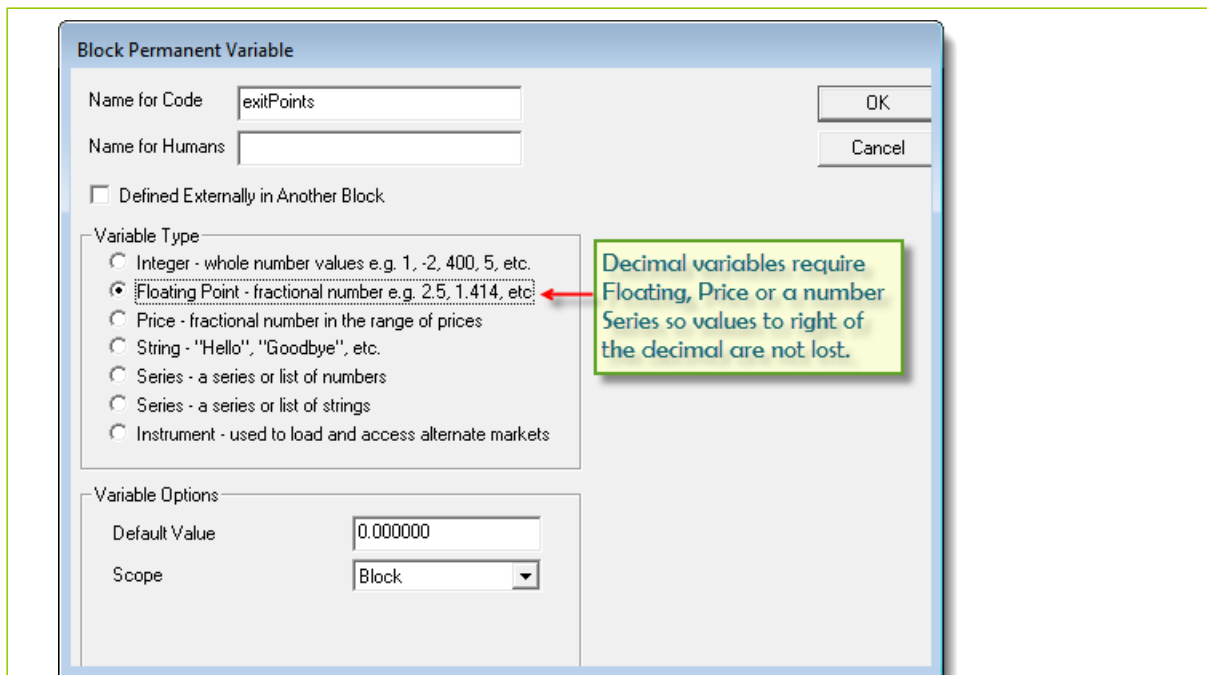
To apply the last bar's volatility measure, we will take the last **ATR** value and then adjust in a way that should help us reduce losses during difficult trading time, and allow a position to stay in place during favorable market periods. In this next section we will program our module to use this new ATR information to generate a point estimation of how much of a distance we should use to place our protective exit prices. Our offset distance will be controlled by a new **Instrument Permanent Variable (IPV)** we will add to our module so that the process can be seen in more clear terms.

Add a Working Variable:

- Open the **Block Editor** and click on the **Entry Orders** script section and select the New menu item to create a new **Block Permanent Variable (BPV) Floating Point** type variable:



- When the new **BPV** dialog appears fill in the details as shown:



Our new **BPV** is created for a stand alone calculation so that the results of the calculation can easily be seen when needed. For example, in the beginning of your experience you might want to see how the value in this BPV changes with each instrument when **Trading Blox Debugger** mode is active.

For this lesson a **BPV** variable was chosen because this is just a temporary working variable. A local variable could have been declared and used, but a **BPV** variables execute a little faster, and their values can be seen from other script sections when needed. It was also chosen to create a variable that improves readability, which often helps to improve understanding. **BPV** variables are intended to not retain a specific value that is only correct for one instrument. Instead they are intended to be used as a working variable or a blox specific that can be used with any instrument. In this case the working value in the **BPV** is only good until it is passed to the **Broker Object** EnterOnOpen function, and is updated before it is used with another instrument.

A value specific to an instrument would be an actual unit entry price, or a unit exit price. These types of values would be placed in an **Instrument Permanent Variable (IPV)** because they only are valid for a specific instrument at a specific time.

Adding Volatility Results:

All the parameters, variables and indicator details we need to add a protective price to our entry orders are in place. To make them work we just need to add two additional lines of programming code, and modify our two Broker functions so the orders are generated with a protective price for the price bar of entry.

Code Changes:

```
' Calculate Current Protective Offset Points      <- Add
exitPoints = avgTrueRange * atrAdjustRate      <- Add
```

```

' Generate a LONG Entry with a Protective Exit below prices      <- Modif
broker.EnterLongOnOpen( instrument.close - exitPoints )        <- Modif

' Generate a SHORT Entry with a Protective Exit above prices   <- Modif
broker.EnterShortOnOpen( instrument.close + exitPoints )       <- Modif

```

- Use the above code details to modify your new Entry Exit module's Entry Orders script area so that your **Entry Orders** script section looks like this next image:

```

Entry Orders
1
2 ' Calculate Current Protective Offset Points
3 exitPoints = avgTrueRange * atrAdjustRate
4
5 If macdIndicator > 0 AND instrument.position <> LONG THEN
6 ' Two conditions must be met - MACD above 0 and
7 ' we are not LONG, Then we enter LONG.
8
9 ' Generate a LONG Entry with a Protective Exit below prices
10 broker.EnterLongOnOpen( instrument.close - exitPoints )
11 ENDIF
12
13 If macdIndicator < 0 AND instrument.position <> SHORT THEN
14 ' Two conditions must be met - MACD below 0 and
15 ' we are not SHORT, Then we enter SHORT.
16
17 ' Generate a SHORT Entry with a Protective Exit above prices
18 broker.EnterShortOnOpen( instrument.close + exitPoints )
19 ENDIF
20

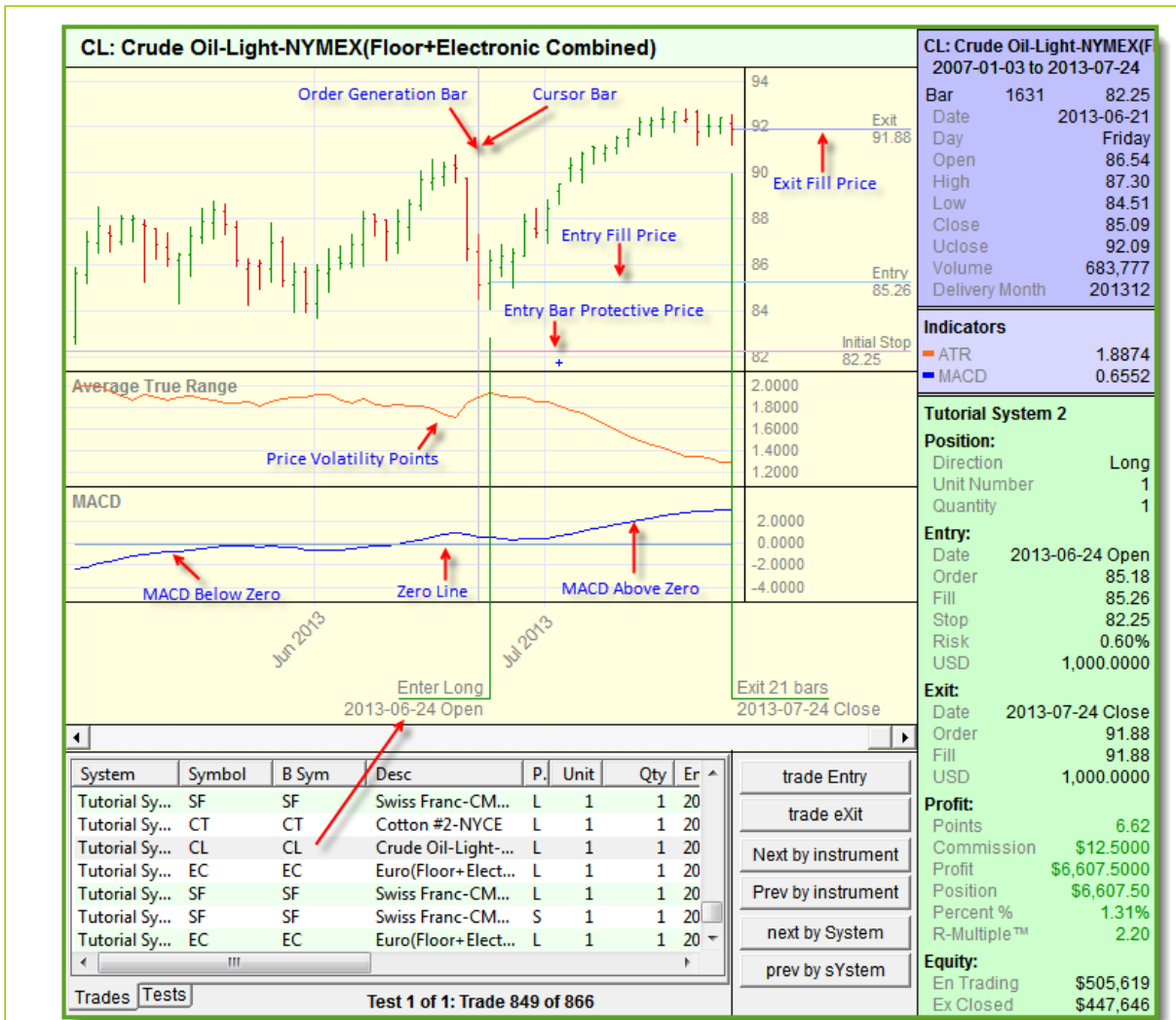
```

When you get your source code to match the changes suggested, orders generated with this module will contain an exit price that will provide an Exit Order for the price bar where your Entry Order was enabled.

As our code is currently written the protective exit order will not be active after the bar of entry, but that can be changed by adding additional instruction into the Exit Orders script section. We will also add script rules to the Exit Orders script section to enable the position to carry the protective price forward for each price bar on which the position is active.

Understanding Chart Display Details:

Image notations point out how the indicators influence the trade action. It also shows how our last change of adding a protective price sits below the current market price and how that protective value establishes the risk this order assumes on entry.



Tutorial System 2 Example Trade Details

Click Image

At this stage of the tutorial we are showing the plot values of the instruments below the chart so their values can be seen. With then in a parallel visual display it is easy to relate how the indicator values relate to what is shown in the price area of the chart. It should also be possible with the aid of the cross-hair cursor data display on the right side of the chart area to see the actual values of everything displayed on the chart.

With the actual indicator and price values available it should be easy to understand how the rules are creating the timing and price values shown for each trade listed in the trade's table below the chart.

All the orders generated with our new system now show an entry date protective initial stop price. This stop price protection is only available on the bar of entry because all orders in Trading Blox are "Day" orders. In our next tutorial topic we'll create the script so that the initial protective prices are available for the active orders after the bar of entry.

Links:

[Operator Reference](#)

This completes this topic.

3.5 Active Order Protection

In our previous lesson we learned how to add a protective price order to our entry order. Our protective exit order would only go active if our entry order was filled by the market. Protective entry orders are designed to limit unfavorable price moves of a new position to the distance in points between the entry fill-price and the protective price, and they only protect if the market's price goes over the protective price using an **On-Stop** execution to close the position.

An **On_Stop** order execution is a buy or sell order that requires the order's **On_Stop** price to be touched or penetrated. When an **On_Stop** price is enabled the order becomes a market order to be filled as soon as possible. **On_Stop** prices are not guaranteed to be filled at the exact execution price. When a fill price differs from the **On_Stop** price the difference is considered slippage. How much slippage happens is based upon the volatility of the market, size of the order, and the volume of trades available after prices have triggered the **On_Stop**-trigger.

All Trading Blox orders are "**Day**" orders.

Definition:

Any order to buy or sell a share or a contract that automatically expires if the market does not enable the order on the day it was placed. This means the order is only valid for one day. If the order's execution type is not enabled by the market in the trading session on the day in which it was placed, the order is automatically canceled at the end of the trading session.

At the end of the entry session the protective exit order will be canceled when its protective **On_Stop** price is not enabled. This means the trading system will need to create a new protective order for each of next trading sessions in which the position is open. This might seem like a problem, but in trading it is the preferred method for most system traders. It is preferred because orders that are not automatically canceled, must be manually canceled by the trader. When they are not canceled they are able to create unwanted positions, and too often in the wrong direction.

Trading Blox only creates "**Day**" orders, and the software makes it easy to keep protective price orders in place for as long as the position is open, but it does require rules be in the system rules to create an updated protective exit order.

Open Position Exit Orders:

Whenever an instrument has an active position the **Exit Orders'** script section executes so as to enable the trading system to place orders for a protective price to exit, remove a position quantity, reduce units, or to close a position. Systems will occasionally place all of the above listed orders when the design of the system requires various methods for managing the position.

Exit Orders Script Section:

Trading Blox **Exit Orders** script section will only executes when an instrument has an open position, so it is always available when a trade is active, and it is out of way when it isn't needed. When more than one module has script section that other modules contain, all the script sections in all of the modules that contain scripted code will execute when script using the same name in other scripts is being executed.

By always executing script sections that have scripted system rules, the timing of when each of the scripts in each of the modules is executed can make a significant difference in how the system performs. Script with the same name will always execute in the order of how each of the modules are displayed in the System Editor center display of system modules.

Trading Blox provides many Broker Exit Order Functions. Prices applied to these exit functions are determined by the script code created specifically for this purpose. In the next section we are going

to limit this next step to only keeping the original protective price order active during the trade so the process can be kept simple and easy to understand.

Open our second Tutorial System and then click on the **Exit Orders** script section. When it appears, type the following into the editor area on the left and save your work:

Exit Orders Script Code:

```
' Enable Entry Protection Exit Order for life of position
broker.ExitAllUnitsOnStop( instrument.unitExitStop)
```

This simple broker statement shown above is all that it takes to keep the original entry order protective price active throughout the life of the trade. It keeps the original protective price active because it is referencing the price value of the trade record so that it can be used with each new protective order. It gets the position's previous entry protective price from the property: [instrument.unitExitStop](#) and it uses that value as the price when the Broker object function [ExitAllUnitsOnStop](#) creates a new protective order. This Broker function will work with Long or Short position because the function knows which instrument is trading as well as the direction of its active trade.

Trading Blox stores a lot of other information with its instrument and it is worth taking the time to review how much information is readily available. Click on this link to browse through the [Data Properties](#) table where each property is listed with a brief example of what it contains.

Adding Protective Stop Indicator:

With the code in place above we now have a position that will be protected using the original protective price we used when the entry order was filled. When Trading Blox displays trade information it won't show where the new protective price orders are located in relation to the price unless we add that ability to the trading system, which is very simple. However, instead of adding a protective price indicator to our tutorial system, we are going to use one that is already available by adding the **Plot Stop Price** block module to our tutorial system.

Go into the **System Editor (F3)**, and be sure our tutorial system module is visible in the System Listing windows in the center of the **System Editor**. Now locate the Auxiliary module shown in this image. Once found, Right-Click on the module and it will appear in the bottom window where our other tutorial modules are listed.

Block Name	Type	Group	Modified
_Show Thread Portfolio Test-TB4	Auxiliary	_Dev	12/30/2012 2:35:21 ...
Chart Margin Equity Ratio	Auxiliary	Default Group	1/28/2009 1:39:12 PM
Margin Equity Risk Manager	Auxiliary	Default Group	4/3/2013 12:31:43 PM
Plot Avg Close Since Entry	Auxiliary	Default Group	1/28/2009 1:39:12 PM
Plot Stop Price	Auxiliary	Default Group	8/6/2013 9:10:38 AM
Report Elapsed Time	Auxiliary	Default Group	3/7/2012 8:30:26 AM
Statistics Robust	Auxiliary	Default Group	1/28/2009 1:39:12 PM
Stats for Multi-Parameter Charts	Auxiliary	Default Group	3/6/2011 9:19:30 AM
Tutorial Auxiliary	Auxiliary	Test Modul	5/6/2013 7:59:39 AM

Right Click on the Block Name: Plot Stop Price

This module is a simple indicator that will place a red-dot above prices on short positions, and below prices on long positions indicating the the position's protective exit price.

Here is the code used in the **Plot Stop Price** indicator block. It will only plot the value of the current [unitExitStop\[1\]](#) when this instrument has an active position.

Plot Stop Price Indicator Code - ADJUST STOP Script Section:

```

' Plots the current stop price for unit one when
' position is active
If instrument.position <> OUT THEN
' Assign Positions Protective Price to Indicator
currentStopPrice = instrument.unitExitStop[1]
ENDIF ' i.position <> OUT

```

This module knows when a position is active because it is referencing the instrument's **position** property. This **position** property can have any of these values:

- 1 for Long Positions
- 0 for Flat or No Active Positions
- -1 for Short Positions

In the example above the variable `currentStopPrice` is an IPV Auto-Indexing Series.

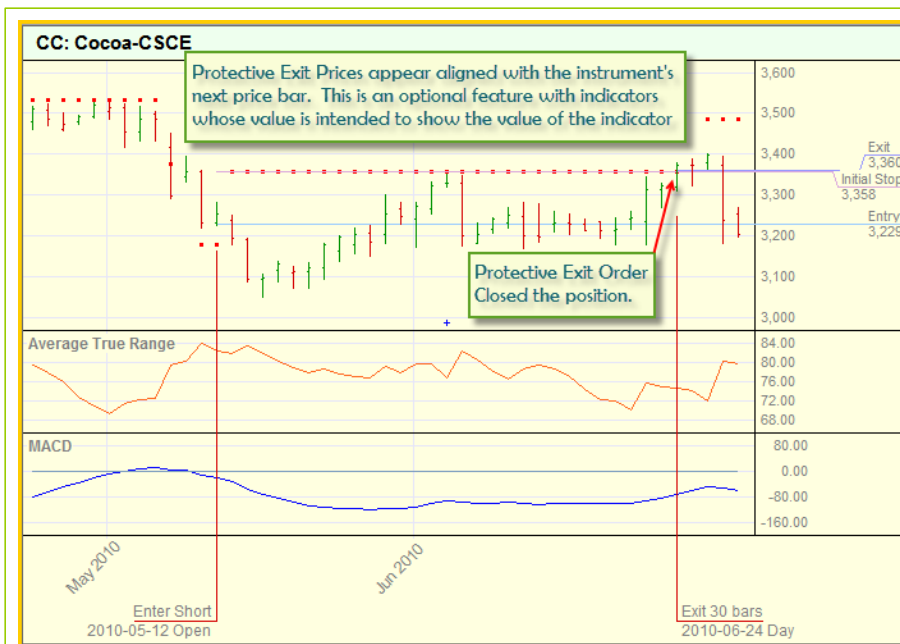
The screenshot shows the 'Instrument Permanent Variable' dialog box with the following settings:

- Name for Code: `currentStopPrice`
- Name for Humans: `Stop Price`
- Defined Externally in Another Block:
- Variable Type:
 - Integer - whole number values e.g. 1, -2, 400, 5, etc.
 - Floating Point - fractional number e.g. 2.5, 1.414, etc.
 - Price - fractional number in the range of prices
 - String - "Hello", "Goodbye", etc.
 - Series - a series or list of numbers
 - Series - a series or list of strings
- Plotting Controls:
 - Plots
 - Display Value
 - Log Scale
 - Plot Color: [Red]
 - Graph Area: Price Chart
 - Graph Style: Small Dot
 - Offset Plot Ahead One Bar
- Variable Options:
 - Default Value: 0.000000
 - Scope: Block
 - Auto-Index -- Uses [n] as Lookback from Current Day

Take a few minutes to examine how this module is created so you'll have some reference on which to create a protective price indicator into your own system, and many other types of position indicators. Also note that the option in the lower right corner of the dialog has enabled the "Offset Plot Ahead One Bar." This option forces the indicator to place its chart information in the area where the next bar will display. This done so the current protective price indicator mark will be placed at the price location where the price of the next bar will trigger the protective order action when the price touches the protective price.

By placing the information ahead one bar for this type of indicator, the information on the screen becomes a little easier to understand that the price touched the protective exit price on the same bar the position was closed.

In operation the Plot Stop Price indicator will appear like this for a Short Position:



Plot Stop Price Indicator Example

In our chart example the protective price doesn't change throughout the life of the trade. This isn't always the best way to protect a trade because it doesn't provide any progressive price action that can preserve gains that favorable price moves may have provided. We are also terminating the position when the price touches our protective price, so alternate protective logic methods that would terminate only when the close price touched or crossed the protective are not explored.

Keep in mind that one of the major issue in trading system design is the lack of finding and applying alternate ideas to see if what we currently have now can be improved by changing some of the methods or logic being used. As you spend time with the various supplied systems that are installed when Trading Blox is installed, notice the various methods for handling entry, exits and protective price methods. Also spend some time searching for various ideas in the [Trader's Roundtable](#) forum's [Blox MarketPlace](#) and [Trading Blox Support](#) sections.

Links:

[Data Properties](#), [Operator Reference](#), [Position Properties](#), [Script Section Type Details](#)

This completes this topic.

3.6 Order Sizing

Orders for the next trading day will appear in the **New Orders Report** section of the **Positions and Order Report** web page that appears when the **Generate Orders** button is used.

All entry orders will need to contain a quantity of future contracts, or shares prior to them being given to a broker for placement. Trading Blox applies order quantities using Money Management modules. These blox modules contain the sizing logic needed to attach a quantity to an entry orders. Quantity calculation methods can vary to fit the trader's idea of how a position should be sized. When Trading Blox is installed it provides three of the more popular methods, and it will support the use of custom sizing methods.

This section will cover some of the methods that can be used for determining order quantity. Each order generated and given a quantity is seen as a unit. Positions can have multiple units each with the same or different amount of contracts or shares. Multiple units can be removed from a position one at a time, or all at once. Units can also be reduced in size, but when the quantity being removed from a unit is equal to quantity of the unit, the unit is closed. When more is being reduced from a unit and there are more units in the position, the remainder not yet removed is taken from another unit.

Determining Entry Order Quantities:

We are only going explore three different methods, but there are many more methods in use. To grasp what other methods are in use, review some of the discussions in the [Trader's Roundtable](#) forum. A lot of knowledge about order sizing and various aspects of money management, and many other aspects of trading is available in our forum. Trading Blox license holders get free full read and write access to the forum, so be sure to sign up for access if you don't already have it.

Money managers can be built into an entry and exit blox, or added as an additional blox that appears in the System Editors Static section in the middle of the System Editor's display. How a money manager is added to the system, isn't important, but not having one included in the system will cause all the orders to show no quantity, and no system performance.

In this section we are only going to show the process of how the size of an order to enter the market using two different sizing options in three different blox modules. Two of three modules use a similar sizing calculation, but one uses an internal volatility estimator to create a single contract risk for orders that don't use any entry bar protective price.

Order quantity provides the multiplier in terms of its quantity that is used to calculate the gain or loss between the entry price and the exit price. When only 1-share or contract is in a unit, the multiplier is one. When two are in the unit, the multiplier is two. However many shares or contracts are in a unit, that is the number by which the results between the entry price and the exit price are multiplied. Those same results are the value assigned to the trade record and used to adjust the account equity being used to calculate the performance of the system.

Let's look at three of the money manager modules that are installed when Trading Blox is installed. As we go through each, the complexity of how they provide size will increase. We will also see how they can control risk when a risk based method is part of the process for determining position size.

Basic Money Manager:

In its simplest form a fixed size of one or more contracts, or shares is entered into the quantity area of the order of parameter field. For our tutorial we will only size the contracts

In a trade where there is an entry price difference, the difference is expressed as the point spread between the two prices. Whatever the spread value, it only represents the price difference for a single share or contact. Spread values are determined by finding the difference between the current Close price and the protective price. That difference is the risk estimate that is used to inform the sizing

logic that is using risk based sizing calculations how to value the risk estimate so it can determine how many contracts or shares to assign to a unit.

When the quantity value is being determined by the Basic Money Manager, and its parameter is set to a value of one, then the price point difference represent the potential loss of the unit. With a quantity value greater than one, the price difference between Close price and the protective price will be multiplied by the value entered into the Basic Money Manager's parameter.

Our first Trading Blox Basic Money Manager can apply a fixed quantity to each unit order. This is the most simple form of adding quantity to an order.

Trading Blox -- Basic Money Manager

Fixed quantity sizing is often used because it is simple. User picks a quantity, enters that number into the parameter and that is the basis for determining the multiple of the price point difference valuations.

While that approach is simple, it doesn't provide the trader with the ability to fix trade size to a percentage or risk, or any other idea that the trader might think will be helpful in controlling risk and in improving the utility of the account's value. However during the development of a trading idea, a fixed quantity size keeps the focus on the new system idea while supplying a reliable quantity during early system testing making the checking of trade results simple.

Basic Money Manager Code:

```

' =====
' UNIT SIZE SCRIPT - START
' =====
' Set the order quantity to the user value entered in the
' parameter field:
order.SetQuantity( sizeOfUnit )
' =====
' UNIT SIZE SCRIPT - END
' =====

```

Fixed Fractional Money Manager:

This order sizing method is favored by professional because it allows the trader to determine the percentage of risk each position is allowed to use. Percentage in this case is determined by the "Risk Per Trade (%)" value entered into the Blox parameter field. A value of 1% is the rate used to determine how much of the account's equity can be made available to the order.

Trading Blox -- Fixed Fractional Money Manager

Let's take an example account that has a value of \$100,000 and then take the 1% rate shown in the Blox image to determine how much risk equity can be allocated to an order. In this case we find \$1,000 dollars is the maximum amount we can risk.

To determine size we need to know how much risk a single contract or share will create between an entry and its protective exit price so we can use the value of the risk to determine how many contracts or shares to use as an order quantity. For our simple example let us also say a single contract will be creating a risk of \$450 because that is the point difference value between our entry price and the order's protective exit price.

In this example with a risk allocation is \$1,000, and a single contract risk of \$450, we can allow the order to have a quantity of two contracts. Here is the math:

Here are the calculation details for Fixed Fractional Sizing:

Account Equity = \$100,000
 Risk per Trade = 1%
 Contract Price Risk = \$450
 Allowed Order Risk = (100,000 × 0.01) = \$1,000
 Max Order Quantity = (1,000 / 450) = 2.22 contracts
 Order Quantity Allowed = 2 contracts

Fixed Fractional Code:

```
' =====
' UNIT SIZE SCRIPT - START
' =====
' Risk Equity Allocation is determined by multiplying
' the current equity available on the bar the order
' is generated by the Risk Rate parameter percentage
' Parameter entered as a Percentage. A decimal value of the
' percentage is used as the multiplier -- 1% = 0.01

riskEquity = system.tradingEquity * riskPerTrade

' When an order is generated with a protective exit price
' the difference between the Close price on the bar where
' order is generated is used as the basis price from which
' the protective exit price is compared. Entry-Risk is the
' point-difference between the bar's Close and order's
' protective-exit price.

' Dollar risk is determined my converting the point difference
' to a monetary value by multiplying the points by the
' instrument's Big-Point value entered into the Future's
' Dictionary for a Future's order, or by using the monetary
' value difference between the Close and Protective Price when
' Stock, Funds, etc. are being used

dollarRisk = order.entryRisk * instrument.bigPointValue

' If the order does not contain a protective exit price,
' there won't be a risk amount in the order. Risk amount is
' is needed to determine the value of the risk. With out a
' risk amount, the calculation for determining the order's
' single contract risk value will be zero. When dollar-risk
' is zero, the order will be rejected.

' When dollar risk is greater than zero, the second part of
' this next conditional statement will calculate a quantity.
```

```

If dollarRisk <= 0 THEN
  ' Set the Order to zero
  order.SetQuantity( 0 )
ELSE
  ' Use the Integer value that results from the division of
  ' the Dollar-Risk for a single contract or share into the
  ' Risk-Equity allocated by the Risk-Per_Trade user value
  order.SetQuantity( riskEquity / dollarRisk )
ENDIF

' When the order quantity is zero or less, reject the order
' and place the order's rejection reason in the Filter.Log

' Instrument.roundLot is the smallest quantity that can be allowed
' to be used for the sizing of an order.

If order.quantity < instrument.roundLot THEN
  ' Place a rejection reason record in the Trading Blox Filter Log.
  order.Reject( "Quantity: " + AsString( riskEquity/dollarRisk, 2 ) _
    + " < Minimum-Round Lot: " _
    + AsString( instrument.roundLot, 2 ) _
    + " Risk Eq: " + AsString( riskEquity,2 ) _
    + " Order-Risk: " + AsString( dollarRisk, 2 ) )
ENDIF

' =====
' UNIT SIZE SCRIPT - END
' =====

```

Note:

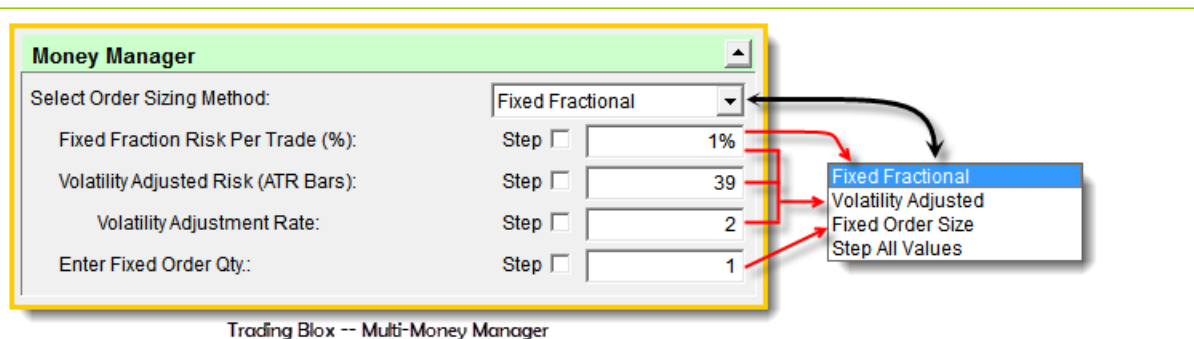
Contracts and Shares must be rounded down to an integer value.

Fixed Fractional Sizing requires a monetary value of risk in order to determine the risk of a single contract. When a system isn't using any protective pricing the Fixed Fractional Money Manager will assume the risk of the order is infinite. An infinite risk will force the math to round down to zero, and the order will not be given any quantity value.

Multi-Money Manager:

This order sizing Blox is a hybrid module that incorporates the two above methods, plus it adds an additional method that will estimate the possible risk amount of a contract or a share by using the Average True-Range volatility indicator adjusted result.

This module is useful during system development and as an all-around sizing module because it gives the trader the option of fixed quantity sizing to flush out the logic and calculation using a consistent order size. It can also be used during development when it might be important to see the performance differences of the three methods in a stepped method simulation.



Trading Blox -- Multi-Money Manager

Using this sizing module must start with the selection of the method intended. **Fixed Order Qty.** and **Fixed Fractional Risk Rate** or **Volatility Adjusted** method. **Fixed Fractional** and **Volatility Adjusted** use almost the same logic as is shown above, but the exception difference is in how the **Volatility Adjusted** will substitute, or provide a risk estimate when no risk estimate is provided.

Volatility Adjust Risk uses the Average True-Range indicator calculation to estimate the True-Range over the previous specified bars. It also has a companion parameter that will allow the user to expand or contract the estimated volatility value which is used as the order's estimate of risk.

Volatility Adjusted Risk Calculation Code:

```

' ~~~~~
' Risk-Equity is the product of Trading-Equity times Fixed Fraction % v
riskEquity = system.tradingEquity * riskPerTrade

' Dollar-Risk is the product of AvgTrueRange * Instrument Point Value
dollarRisk = averageTrueRange * instrument.bigPointValue

' Set the trade size.
If dollarRisk <= 0 THEN
  ' Set Order Quantity to zero
  order.SetQuantity( 0 )
ELSE
  ' Order Quantity is the integer value dividing Risk-Eqty-Amt by Doll
  order.SetQuantity( riskEquity / dollarRisk )
ENDIF

' ~~~~~
' Reject order when quantity is less than 1.
If order.quantity < instrument.roundLot THEN
  ' Reject order - send message to Filter Log Report
  order.Reject( "Order Quantity less than 1" )
ENDIF
' ~~~~~

```

Fixed Fractional Sizing uses the order's point spread between the close of the bar on which the order is generated to the protective price that will be active when the order is filled. This approach when done carefully can be a close approximation of what each contract or share might experience should the trade fail and be filled at the position's protective price. It also uses the current value of the account to determine risk allocation. By using the current value of the account the risk allocation preserves the risk equity intended as the account value changes. In simple terms, a percentage allocation process allows the trader to use a fixed leverage rate that is consistent regardless of account value. This doesn't happen with fixed order sizing because when the account amount is low, a fixed quantity will create a larger risk level than the same size will create as the account's value increases.

Volatility Adjusted Risk estimates its Risk-Equity in the same way as Fixed Fraction Sizing, but the risk estimate is determined by using the point volatility results of the Average True-Range calculation for the period of bars listed in the parameter field. In most cases this volatility result must be adjusted to reflect the method of how the system will exit a failed position if risk-control is an important aspect for the trader.

Adjustments to the Volatility risk-point results can be amplified or reduced by changing the value of the "Volatility Risk Rate" parameter. In most cases for long-term trend trading the value in this parameter will need to be higher if the system's performance is showing excess draw-down percentages that are greater than expected or what would have been the result if a carefully created protective price method had been active for the system.

When the Volatility risk-points are too low, orders will size with more quantity than they should have been given because the risk-points were too small. This means that even though the Fixed Fractional Rate being applied is allocating equity amount of that rate, under estimation of risk that increases size in reality will increase the risk-rate being created when the size is too large.

This sizing module is an important module to use to get a feel for how it works and also because it is a flexible process module. Flexibility can provide a comparison of how the system would perform using various methods in a stepped simulation.

Links:

[Operator Reference](#), [Trader's Roundtable](#)

This completes this topic.

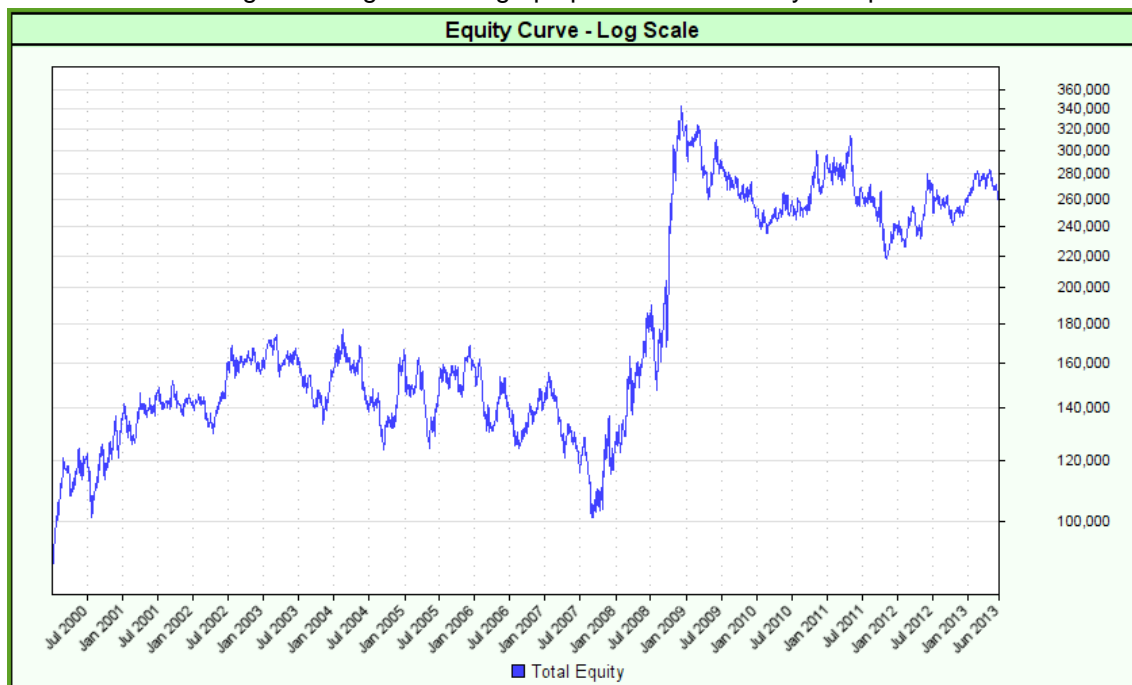
3.7 Trading Risk

This topic is a brief presentation of how fixed and adjustable quantity sizing affects the system's risk profile. As order sizing methods change so does the risk effect and trading system performance change.

In this section we are only going to use the tutorial entry exit system we developed earlier as the method for how we will show the risk and performance differences between fixed and risk rate sizing.

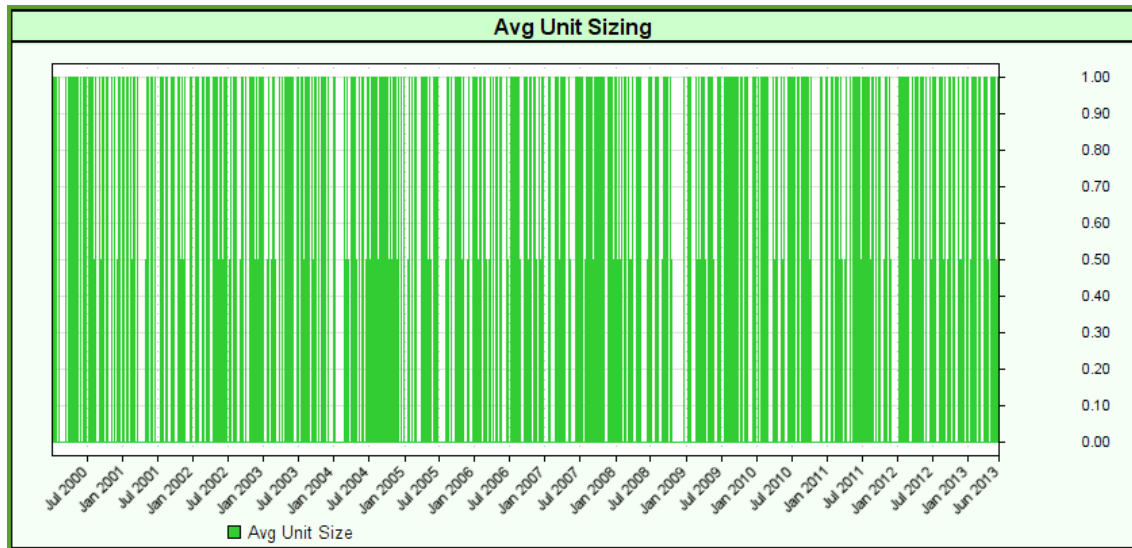
To demonstrate fixed quantity we will use the Basic Money Manager and set the quantity parameter to 1 contract or share, and we will test the trading system from the beginning of 2000 to the current data download date.

At the end of test Trading Blox will generate a graph profile of how the system performed:



Basic Money Manager Fixed Order Size Graph

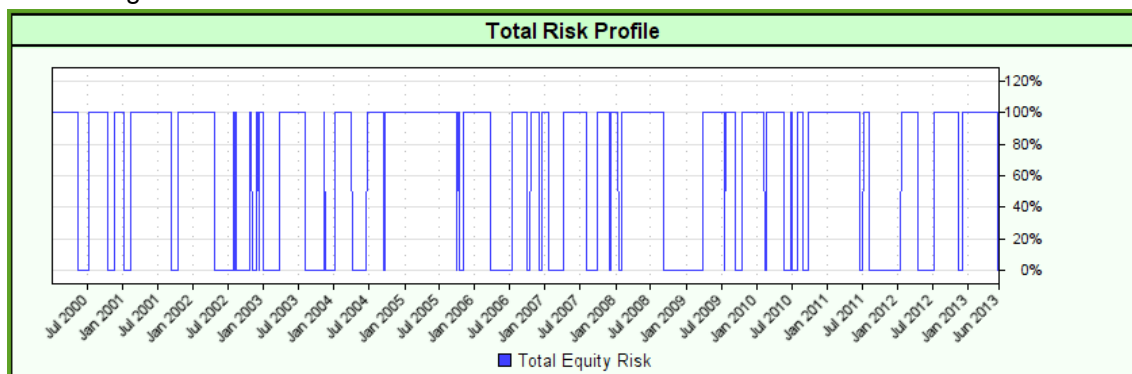
During testing this custom graph created to show the static nature of how the order size of the Basic Money Manager consistently applies the same quantity regardless of how much growth appears in the equity profile graph:



Basic Money Manager Fixed Order Size Graph

Our first tutorial system doesn't include a protective exit price as part of its trading logic. When no protective price is available to generate a position's estimate of loss when trading, the software assumes the risk of loss is infinite and thus shows the Total Risk profiles as being 100% when ever there is a position is active.

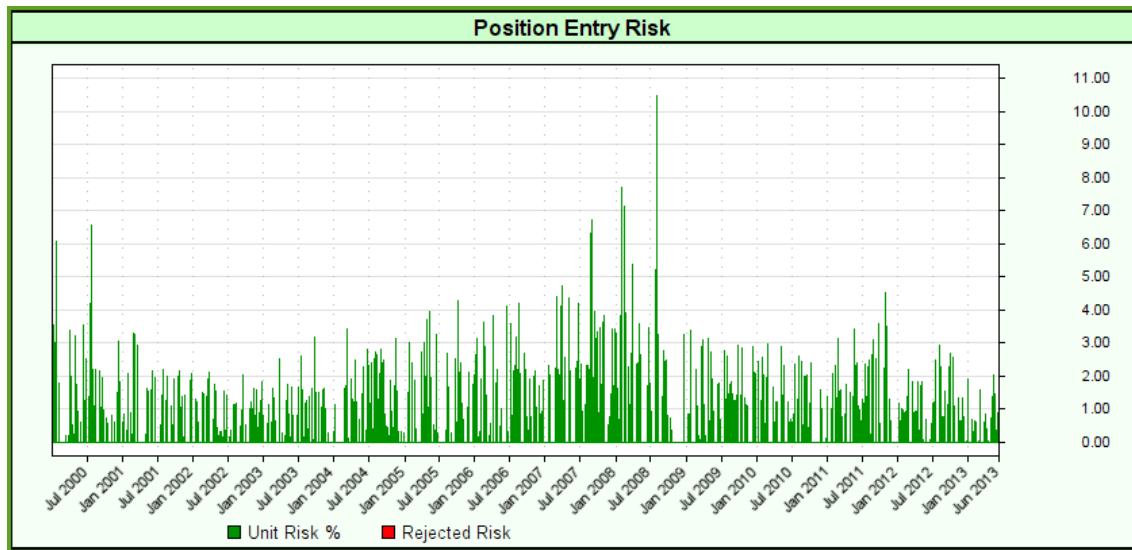
In the standard Performance Results Summary report when the Total Risk Profile option is enabled in the Preference section, the reporting will display an estimate of how much risk that system is creating at all the trade days over the period of dates selected. Position risk is determined by the system's calculation of risk for each of the position active on each date, which is then summarized at the end of each trading record:



Basic Money Manager No Protective Exit Price Total Risk Profile

In the above graph the system is exposing the entire account whenever there is an active position.

When a fixed quantity sizing method is used each order creates a varied risk load on the system's account. In this next graph generated when the orders were sized that created the equity profile shown above, the variations in each orders risk is so varied it would be hard to estimate and limit the system's open risk.

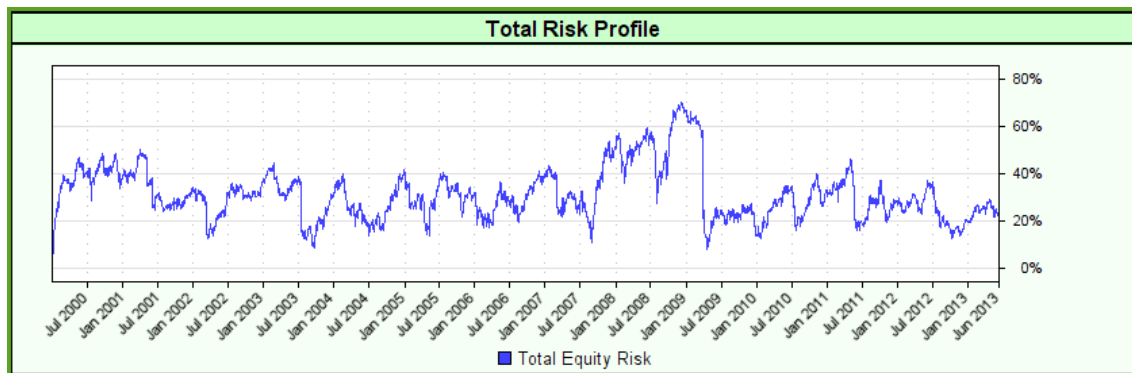


Basic Money Manager 100k Risk Graph Order = 1

When order risk become so varied it is then hard to limit how many positions can be active so as to limit the entry risk exposure to a preferred risk level.

Adding Protective Exit Pricing:

When the protective prices are used with entry orders, the Total Risk Profile of the system drops down to where it might be expected. This next graph shows an example of what we might expect when we add our protective prices to the orders at entry, and then keep them in place over the duration of each position:

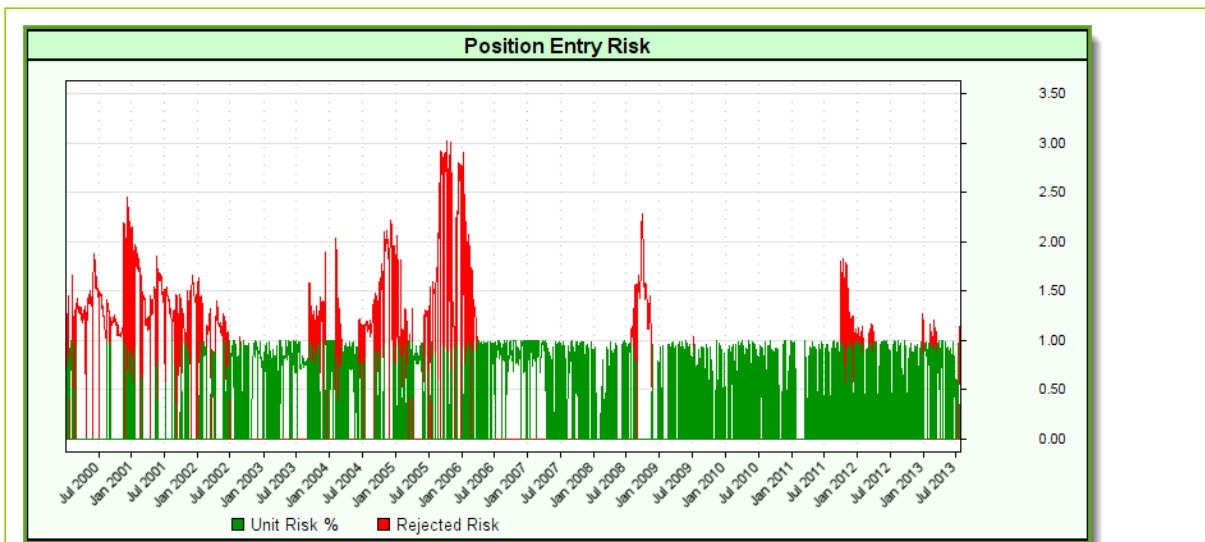


Basic Money Manager Size = 1 with Protective Exit Orders

Total Risk profile has now moved away from finding all the active positions carrying an infinite amount of risk, to a profile of how the probable risk of the system would report if the active positions exited by their protective prices. Total risk is an estimate of the probable loss that would happen and the amount is based upon the position's quantity times to trade's loss amount if it were to exit at its protective price.

Risk Based Sizing

Risk based sizing determine the entry order quantity based upon how much risk is allowed. Risk allowed is calculated by multiplying the system's equity amount by user's fixed percentage rate. This means that a for an account valued at 100,000, a 1% risk rate will allow risk a allocation of 1,000. With a risk allocation of 1,000 and an example entry risk of 500, the risk allocation will allow a quantity of two contracts to be assigned to the entry order.



When the entry risk amount is greater than the allocated amount, a quantity size that is less than the minimum round-lot size will result. Quantities less than the round-lot specified for the instrument will cause an entry order to be rejected. Fixed rate sizing is a method for limiting how much risk a position is allowed to apply to a system, it also allows the system to create a fixed leverage rate as the account value changes.

In the graph above the red spikes show the risk rate of the orders that were rejected. Rejections are more frequent at the beginning when the account value is low, or when there is a period where high volatility is dominating the markets. As the account equity grew, the rejections became less frequent, but the period around 2008 showed a lot of market instability which is a reflection of volatility.

Observations of the green area shows the average risk rate shown is less than the user's parameter risk rate of 1%. It is less because it often happens the division by the entry risk amount into the allocated amount won't always be an even number. When the results have a decimal amount, the value is rounded down to the next integer value which creates an average risk rate that is less than the user's established risk rate of 1%.

Links:

[Operator Reference](#)

This completes this topic.

3.8 Money Management

Account risk of each position is best served when position risk is managed to limit the total risk any a position can assume. It is also important to allow enough risk to enable a reasonable order quantity be allowed to help grow the account's balance at a safe rate that doesn't cause draw down periods to risk an account to the point where the trading stops because of insufficient funds, or insufficient confidence. Risk rate boundaries are different for each of us because our beliefs and expectations are all different. However, each of us can discover our thresholds with system historical testing and personal reflection. Ideally the amount of risk each of us allows isn't so large that a position's failure becomes a significant event during a prolonged draw down cycle. Large losses during difficult trading periods will quickly consume an account to the point where there is too little money, or too little courage to keep trading.

Finding this balancing point for each of us is the critical goal for all traders because draw-down periods are going to appear at some point, and long draw-down periods are hard on the account and the trader. Meaning the total level of risk a trader exposes to their system's account must be kept low enough that hard times don't drain the account or destroy the belief in the system, but still provide a practical level of controlled risk of the account's value.

Total Account Risk:

Total account risk is the sum total of each active position's risk. How many active positions to allow at the same time determines the percentage of account risk. A major portion of how much risk a position contributes to the total risk exposure is influenced by how far prices can be allowed to move against a trade before that trade is terminated. Order risk is determined by measuring the current close price to the On-Stop Exit price to determine the amount of risk points. A single contract or share risk points converted to a currency value is the basis for determining how much risk a position will be allowed to assume when the order is given a quantity size. In simple terms risk is based upon the cost applied to a single contract or share when prices move against a trade's position. This adverse point difference is converted to a monetary value so that risk as amount of loss for a single contract can be used to estimate how many contracts or shares can be assigned as a quantity for a new order. Determining how much money to allow a position is determined by the system's allowed risk rate for sizing orders.

When an order is created and sized to have only 1-contract, the risk of the position is the risk of that single contract. When an order is sized with more contracts the number of contracts times the risk amount of a single contract determines the position risk. Contracts that use risk based sizing are designed to limit total entry position risk to the system's position allowed risk sizing rate. Multiple positions sized and constrained to the system's risk rate can be summed to determine the account's total risk rate.

When an order is generated with a risk amount for a single contract that is larger than allowed, the fixed quantity method of sizing will allow the order to reach the brokerage because there is no risk filtering logic in that order sizing module. While this might sound risky, fixed quantity sizing is the best way to check on how the software handled the transactions. By understanding the transactions the cost of slippage, and commissions, when allowed during a test, can be seen in how the position is settled at position termination.

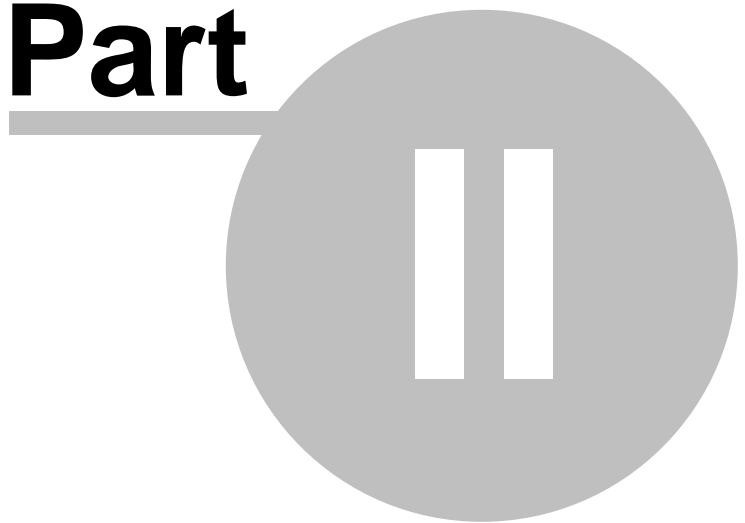
When a system wants to have better risk control, the process of sizing should use logic that will limit the position allocation amount to the trader's risk rate so an order with excessive single share or contract risk levels are rejected, and those with small levels of risk will be allowed to have more than a single contract or share. In Trading Blox the "**Fixed Fractional Money Manager**" and "**Multi-Money Manager**" blox modules have risk filtering logic and allow the user to establish the risk rate for each new order.

Under Construction!

This completes this topic.

Trading Blox Architecture

Part



Part 2 – Trading Blox Architecture

Now that we've built a simple system lets look a little deeper into the architecture of Trading Blox.

Trading Blox has many different parts. This can be a bit overwhelming for a new user. Fortunately, if you examine each of the pieces one by one they are not hard to comprehend.

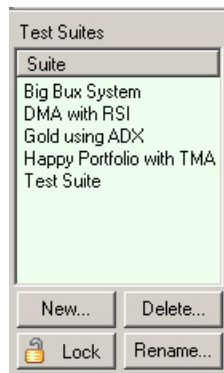
If you followed the previous examples, you've already used each of the following components:

Suites

A test suite is a collection of specific settings for a particular set of systems. Let's say you were testing a Dual Moving Average system with an RSI system and you had a certain portfolio and certain parameter settings that you liked. Your settings for these two systems including the system allocations for the Dual MA and RSI system are stored in the current Test Suite. To save these settings you can create a new Test Suite, calling it something like "Dual MA and RSI".

Suppose you wanted to work on a new system without disturbing your existing settings. You could create a new Test Suite for working on the new system. Later by simply switching to the "Dual MA and RSI" suite you can get back the exact same settings.

Suites can always be seen in the upper left of the Trading Blox Builder screen. You can create new suites or delete unused ones. You can also lock your suite when you are satisfied with it to keep from accidentally changing your settings. You must lock a suite in order to generate orders.



Systems

A system is a collection of formulas and rules that define entries and exits for a set of markets. In Trading Blox Builder, systems are a collection of smaller components called Blox. If you own the Turtle version of Trading Blox Builder, you can use the built in systems that are provided. If you own the Pro version, you can assemble your own systems using the build-in Blox or ones that you download or purchase from others. If you own Trading Blox Builder Builder, you can create your own Blox.

Trading Blox Systems consist of the following components:

System Component	Corresponding Block Type
What to Trade	Portfolio Manager
When to Trade	Entry Blox
Whether to Trade	Risk Manager

How Much to Trade
When to get out

Money Manager
Exit Blox

Blox Modules

Blox are system modules that encapsulate trading ideas. Most of the Blox are self-contained parts of a trading system designed to be connected with other Blox as a component part of a trading system method. Some Blox can access data outside of their module, and outside of their system when their scope settings are set for external access.

The basic components of a trading idea are:

- **Parameters** - used by indicators to determine their specific computation result, for example: the number of days in a moving average. Not all Blox require parameters when the numerical basis for internal calculations is coded into the scripted code.
- **Indicators** - used by the rules as indicators of market conditions, moving averages, RSI, ADX, etc. Many indicators are available within the Indicator section of a Blox. Those not available in that area can be created by entering the source code required for an indicator calculation result.
- **Rules** - used to determine when to enter or exit; how much to buy or sell, or how much risk is too much, buy on moving average crossover, what instruments to allow and other ideas. A rule can be as simple as Buy If **RSI** > 55 etc.

By encapsulating trading ideas into a stand-alone Blox module, a package is created that can easily be linked to one or more systems that need the trading idea contained within the Blox. Blox are trading objects, and while these objects only need to be created once, they can be many times by other systems to simplify the creation of different system methods.

Blox can also be used in multiple systems at once. This is one of the most powerful features of Trading Blox Builder Builder.

Scripts

Just like a director and actors in a movie use scripts to coordinate action, Trading Blox Builder uses scripts to coordinate trading and to implement a system's rules. Scripts are more powerful than simple rules and they can even be used to implement sophisticated risk and portfolio management algorithms.

Trading Blox Builder defines script types which are run at specific times during the simulation which correspond with specific times during the test and trading day. Some scripts have a specific function (such as adjusting stops for the day) while others are simply place holders for tasks that need to be performed regularly like end of day calculations, keeping track of risk, etc.

Trading Blox Builder is quite smart about when it executes scripts in system. For instance, the "Entry Order Filled" script in an Entry Block only gets run when a trade is entered because an entry order's conditions were satisfied by the market. This is one of the reasons that Trading Blox Builder is so fast.

For more information on the Scripts available in Trading Blox Builder, see the [Script Reference](#) section.

Trading Objects

Since Trading Blox Builder simulates real trading as closely as possible to enable you to implement

trading systems that are as realistic as possible, we use concepts called Trading Objects in our scripts with correspond with the real world trading things (or objects) like brokers, instruments, etc.

The most important Trading Objects in Trading Blox Builder are the "Instrument" and "Broker" objects.

Scripts use the instrument object to get information about the current stock or futures market (i.e. instrument). So a script might access the current stock's close using the following code fragment:

```
instrument.close
```

This shows a **property** of the "Instrument" trading object called "close". Properties are used to access data associated with a trading object.

A script might also tell the broker to enter a stop order using a code fragment like this:

```
broker.EnterLongOnStop( entryPrice, protectStopPrice )
```

this tells the broker to enter a Buy Stop to initiate a long position at the price represented by "entryPrice" with an exit stop to be entered at the price represented by "protectStop" in the event that the entry stop is filled.

"EnterLongOnStop" is an example of a **function** of the "Broker" trading object. Functions change the way a simulation behaves or change the state of test data. Functions affect the outcome of a test directly.

For more information on the Trading Objects used in Trading Blox Builder, see the [Trading Object Reference](#) section towards the end of this manual.

Variables

The last example used two Script constructs known by programmers the world over as "variables".

Variables are simply a name which represents a value or series of values. If you have used a spreadsheet then you have used variables. For example, in a spreadsheet column B row 4 might be the total sales for the month. In Excel you could name this cell to something like "monthlySales" then in other cells you could refer to that variable (cell) as either B4 or "monthlySales".

In Blox Basic you can create variables which have names and can hold values just like a spreadsheet cell can.

So the name "entryPrice" in the above script fragment is a place holder for the value which corresponds with the entry price, while the name "protectStopPrice" is a place holder for the value which corresponds with the stop price which should be used to exit the position.

For more information on using Variables in Trading Blox Builder, see the [Variables Reference](#) section.

Parameters

Parameters are a special type of variable which can be stepped using the Trading Blox Builder parameter stepping features. You should define a parameter instead of using a fixed constant value in the trading rules for a system.

Parameters are also often used to define indicators.

Indicators

Indicators are another special type of variable which can be displayed on the trade chart. Trading Blox

Builder includes most of the common indicators, Moving Averages, MACD, ATR, RSI, ADX, etc. Many trading systems are built using indicators.

Units

The concept of units is used throughout this manual. Units refer to concurrent positions taken in the same instrument as part of the same trade. When you enter a position direction for the first time, for example you enter long when you were previously short or out, this new position is the first unit. If you then enter long again, that would be the second unit. In this way you can pyramid your positions, by entering multiple units at different prices and different quantities. You can also exit these positions separately, or all at once.

Section 1 – Working with Systems, Blox & Scripts

Trading Blox are the individual components of a system, so building a system is creating and/or combining Blox together.

When you define a system to contain certain Blox, the functionality contained in those Blox is what defines what the system will do. Each Block has a particular purpose. A system can have multiple Blox of some types (like Entry and Exit) and only one Block of certain other types (Portfolio Manager, Risk Manager, and Money Manager).

This section will show you how to create systems from Blox and Scripts.

1.1 Working with Systems

You can use the systems that come with Trading Blox, you can get systems from other people and import them, or if you have the Pro or Builder versions of Trading Blox Builder, you can create your own. This section of the manual describes how to create your own systems by assembling Blox. You should also be familiar with this section if you want to modify an included system, or a system you have purchased from another source.

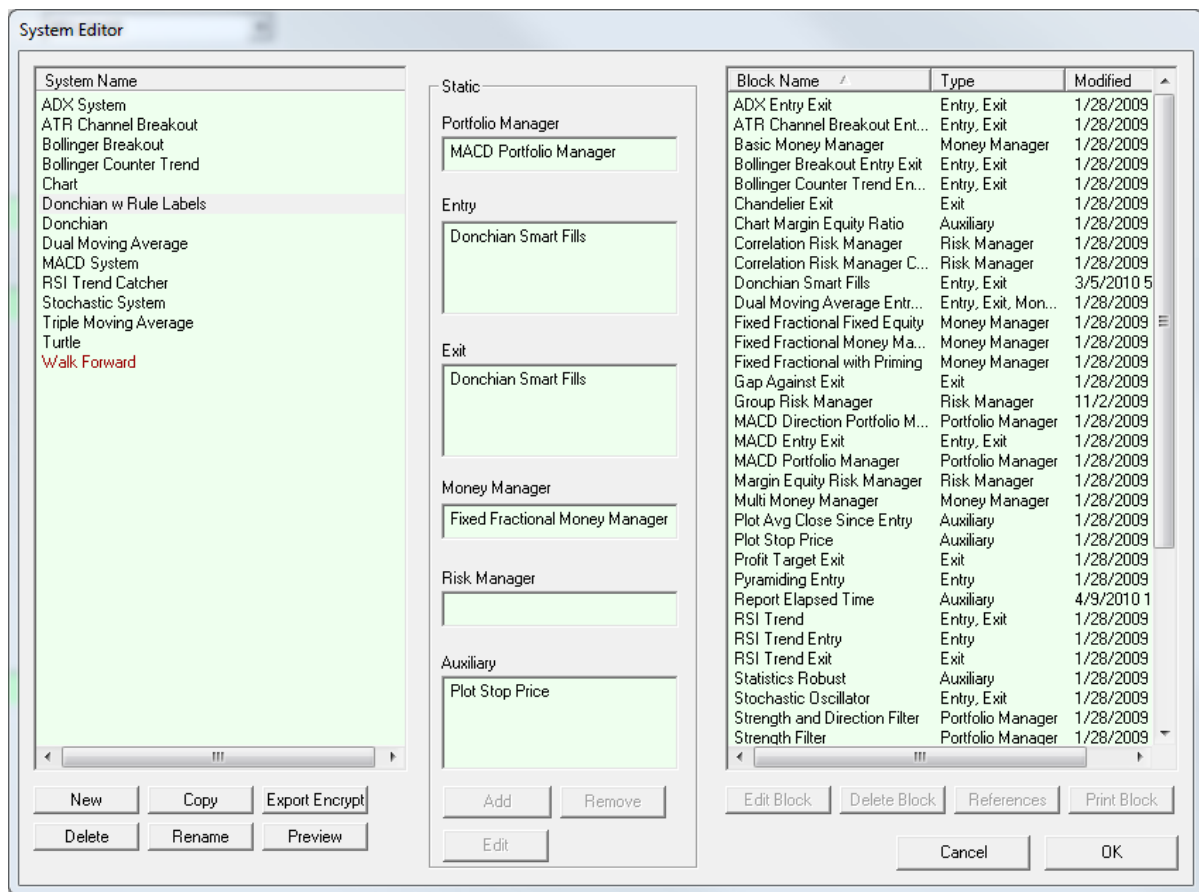
What is a system?

Systems are a collection of Blox. There are five Block types that we use, and each has a particular purpose in a trading methodology. As described earlier, the basic components of a trading system are:

System Component	Corresponding Block Type
What to Trade	Portfolio Manager
When to Trade	Entry Blox
Whether to Trade	Risk Manager
How Much to Trade	Money Manager
When to get out	Exit Blox

The System Editor

Selecting "Edit Systems" from the System Menu will bring up the System Editor window.



New System

You can create new systems and delete systems using the system editor.

You build systems by selecting a system on the left, and adding Blox from the available blox list on the right.

Some of the lists accept multiple Blox and some lists can accept only one Block. The Portfolio Manager, the Risk Manager, and the Money Manager can accept only one Block per system. The Entry Block and Exit Block can accept multiple Blox per system. The reason for this is that you may want to have multiple entry/exit ideas executing at the same time.

Copy System

Select the system to copy, press the copy system button, and enter a new name.

Rename System

Select the system to rename, press the rename system button, and enter a new name.

Delete Systems

Caution: If you delete a systems, you cannot recover the system other than to recreate it. The good news is that since a system is only a collection of Blox, recreating the system is as simple as creating a new system and dragging the required Blox into the system. It is still good practice to back up your systems on a regular basis. The systems are stored in a folder called "Systems" in your Trading Blox folder.

If a System contains a Block of type that can only have one Block, you must remove the Block before

adding a new one. A system can have multiple Entry and Exit Blocks.

To delete a Block from the system, select the System, select the Block within the System, and click "Remove Block from System" You will only be deleting this block from the selected system. You will not be deleting this block from your list of available blocks.

The Blox required for a system to be able to trade are Entry Signals, Exit Signals, and Money Management. The Entry and Exit Block need to call the Broker object to enter and exit trades, and the Money Manager Block needs to set the trade quantity for each trade. You can use the [Basic Money Manager](#) to get started quickly.

After you have modified a System (changed, added, or deleted the Blox contained in the System) click the OK button to save and exit, or click Cancel to cancel all changes. To edit a Block or view the code, double click on the block name. This will bring you directly to the editor with that Block selected.

Preview

This button will open a printable listing of the system, included blox, scripts, parameters, indicators, etc.

Export and Encrypt System

This button will export the system and attached blox to a special encrypted file. It will be put in the Export folder, which will be opened. You can then send this file to another Trading Blox user. They will be able to use and test with the system, change parameters, etc, but will not be able to view or edit the system or blox.

To use one of these exported systems, put the .tbz file in your Import folder before starting up Trading Blox.

Note: Be sure that the name of the System and the name of all Blox in the system are unique. We recommend that you use your name, or some other unique identifier, in the blox and system names. In this way, they will not conflict with other blox or systems that may already be in an environment prior to importing the encrypted system.

Import System

If someone sends you an encrypted system, a .tbz file, you place that in your Import folder. When you startup Trading Blox, this system will be listed and available for testing, but you will not be able to view or edit the system or the blox.

Add

Use this button, or a right click, to add a block to a system.

Remove

Use this button, or right click, to remove a block from a system.

Edit

Use this button to edit a block in the Blox Editor.

Edit Block

Select a block, and click on this button to edit the block in the Blox Editor. Same function as the Edit button in the middle.

Delete Block

Select a block, and click on this button to delete the block from the system and delete the file as well. Blox cannot be delete if they are in a system. Use with caution as blox cannot be recovered once deleted.

References

Select a block, and click on this button to see a list of system references. All the systems the block is in will be listed.

Global Suite Systems

If the system name is the same as a suite, it will be a global suite system. This system allows blox to be attached directly to the suite, and have access to data from all systems. The global suite system scripts run after all the system scripts of same name run.

Scripts available for use in a global suite system:

Before Simulation, Before Test, After Trading Day (access to final test equity), After Test, After Simulation. These scripts have no system object context.

Entry Order Filled, Exit Order Filled, Can Add Unit, Can Fill Order -- note that these order scripts run for all orders placed or filled regardless of originating block or system. The scripts also have access to the system, instrument, and order object from the block in which the order was placed or filled.

1.2 Working with Blox

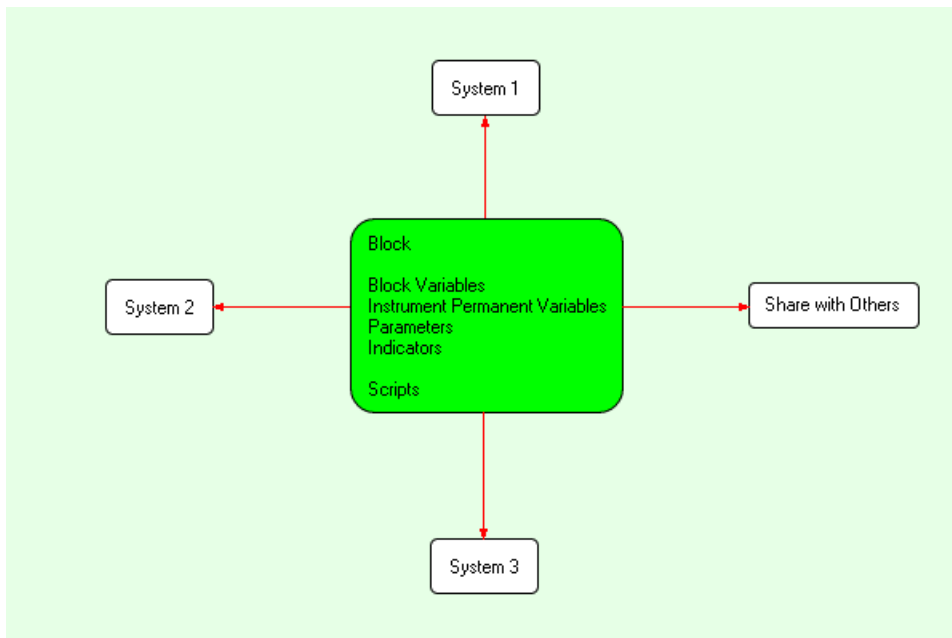
You can create, delete, and edit Blox from the Blox menu. If you edit a Block, that will change the behavior of all the systems that use that Block. This is a very powerful feature since you can have multiple systems that use a particular style Money Manager. And when you update or improve that Money Manager it will improve all the systems that use that Block.

Each Block is entirely independent. The reason for this is so you can mix and match Blox in other systems, and trade or sell them to others. You can create a Risk Manager Block, and drag and drop it into any one of your systems to see what sort of difference it makes. You can also share or sell these Blox with others without any modifications required.

Each Block is made up of multiple Scripts. These Scripts contain the actual code that tells Trading Blox what to do. Each Script can have its own local variables, procedures, and functions. Local variables are only available to that Script. In addition, local variables are undefined at the beginning of a script, and should be reset to some known value at the top of the script before being used.

In addition to local variables, there are Block variables available to scripts in a Block. These include: Block Global's, Instrument Global's, Parameters, and Indicators.

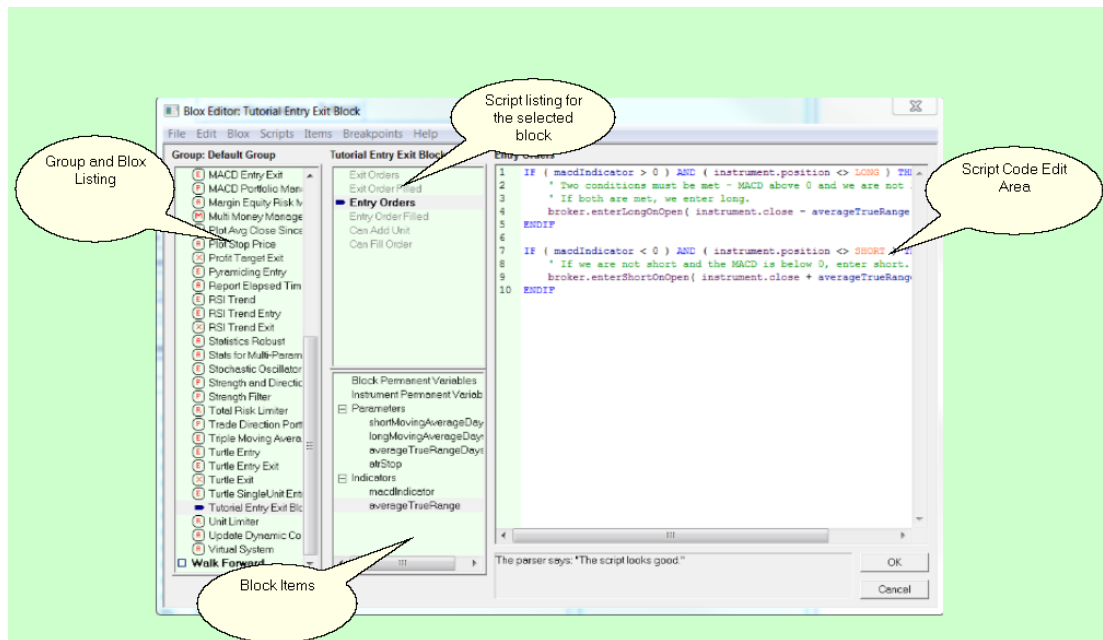
The Scope determines if these variables, parameters and indicators are available only to the Block, System, or Test. Simulation scope is Test Scope but retains values throughout the simulation.



You can view, create, or edit Blox by clicking on the Blox menu item from the Edit menu:

Undo	Ctrl+Z
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Systems	F3
Blox	F4
Futures Dictionary	
Stock Dictionary	
Forex Dictionary	
Portfolios	F2
Preferences	

This will bring up the Blox Editor:



For the new and experienced user alike, it is prudent to make a copy of a built-in blox before you edit them. This serves several purposes:

WWW.TRADING-SOFTWARE-DOWNLOAD.COM

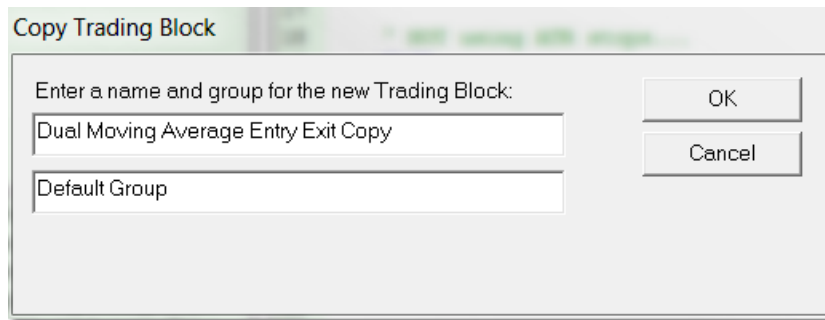
- A new installation or update/upgrade (new version, etc.) will replace all Blox and Systems that came with the product, replacing your editing if the names are the same.
- It is always good to have a backup in case you want to return to the original.

Select a block on the left and right click to select copy or new.

If you click "New" this dialog comes up:

The types are explained in the following sections.

If you click "Copy" will copy the Block and bring up a dialog for the new name and new group.



Once you have your new Block, you can edit the different scripts by clicking on them. You can copy code from other systems by using Control - c and Control - v. Starting from scratch as a beginner if you have no programming experience can seem daunting. We suggest you look at other simple Blox (such as the MACD or Dual Moving Average) to get a feel for the syntax.

Scripts and block items such as parameters variables, and indicators can be copied, pasted, deleted, or moved up and down. Right click on an item to see the menu selection.

You can add new scripts to a block, such as adding a Can Fill Orders script to an Entry Block. You can also add new custom scripts which can act as [custom functions](#).

1.3 Working with Scripts

The following is a chart showing the most basic Blox and associated scripts. Each script is called only under certain circumstances. For each script a mark shows whether the script is called every day, for each instrument, or for each position. For more detail, see the [Simulation Loop](#).

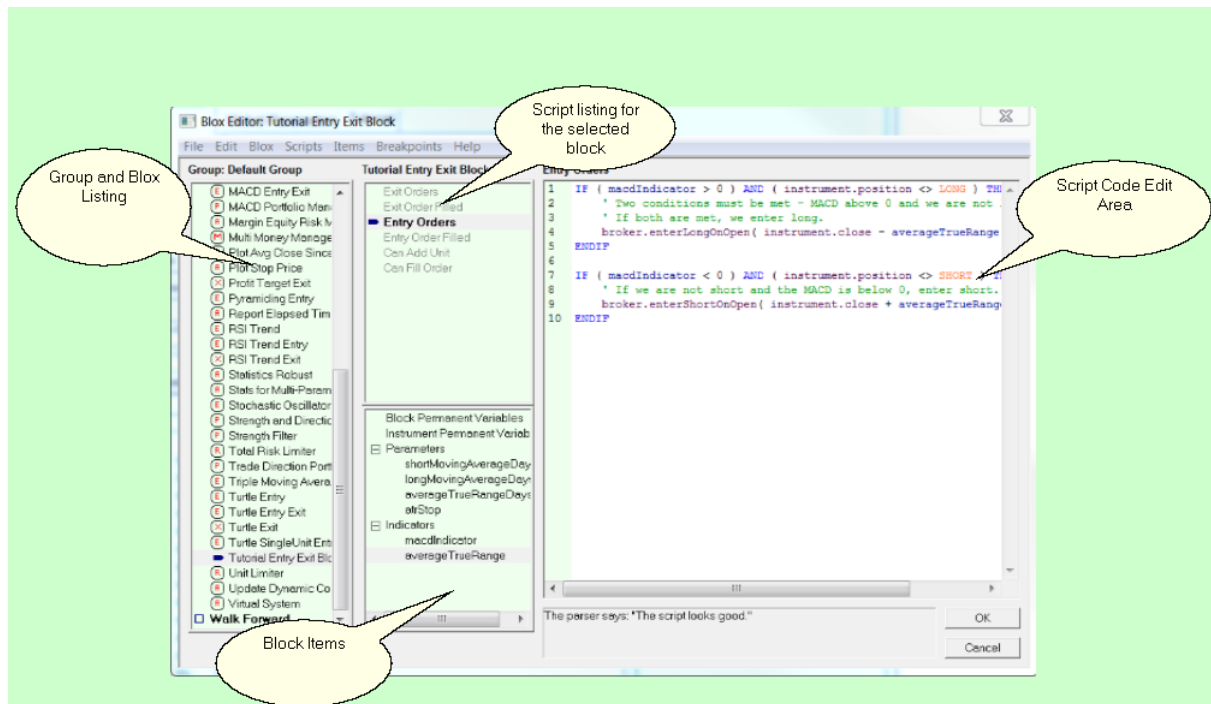
Block Type	Script Type	Day	Instrument	Position	Called When
Entry					
	Entry Orders		•		start of day
Exit					
	Exit Orders			•	start of day
Money Manager					
	Unit Size		called by broker only when an order is filled		

You can build most systems using only these Block Types and this limited very set of scripts. After you have some experience with Trading Blox Builder you may want to experiment with some of the Intermediate Scripts.

Scripts are created and edited in the Block Editor. You access the Block Editor by clicking on the Blox menu item in the Edit menu:

Undo	Ctrl+Z
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Systems	F3
Blox	F4
Futures Dictionary	
Stock Dictionary	
Forex Dictionary	
Portfolios	F2
Preferences	

This will bring up the Block Editor:



The *Blox and Groups* area lists all the Trading Blox available. The Script area shows all the scripts currently in the selected block. If there is code associated with a particular Block's script, that script will be drawn in Black text and Bold. If a script is empty it will be dark gray. If you are examining a new system you can easily tell which scripts have been used by the Blox in that system by looking at the color of the scripts in the list.

The Blox Items area shows all the variables, parameters and indicators used by a particular block. For more information on [Block Permanent Variables](#), and [Instrument Permanent Variables](#), see the [Variables Reference](#) section. For more information on Parameters and Indicators see their respective reference sections: [Parameter Reference](#) and [Indicator Reference](#).

You can create a new variable, parameter, or indicator by selecting the appropriate type and selecting

new from the menu or right click, or by double-clicking on the type itself in the list.

To change the values associated with a variable, parameter, or indicator you can double-click that item directly or select edit from the menu.

The order of items in the list determines their order in the User Interface that gets generated as well as the order of processing for calculated indicators. To change a script's or item's position, select that item and then use the Move Up or Move Down menu item.

Basic Scripts

The following table shows an intermediate view of the scripts available to the most common Block Types:

Block Type	Script Type	Day	Instrument	Position	Called When
Entry					
	Before Simulation				start of simulation
	Before Test				start of test
	Before Trading Day	•			start of day
	Entry Orders		•		start of day
	After Trading Day	•			end of day
	After Instrument Day		•		end of day
	After Test				end of test
	After Simulation				end of simulation
Exit					
	Before Simulation				start of simulation
	Before Test				start of test
	Before Trading Day	•			start of day
	Exit Orders			•	start of day
	Adjust Stops			•	end of day
	After Trading Day	•			end of day
	After Instrument Day		•		end of day
	After Test				end of test
	After Simulation				end of simulation
Money Manager					
	Unit Size	called by broker when placing new order			

You can build very complex systems using only these Block Types and this limited set of scripts.

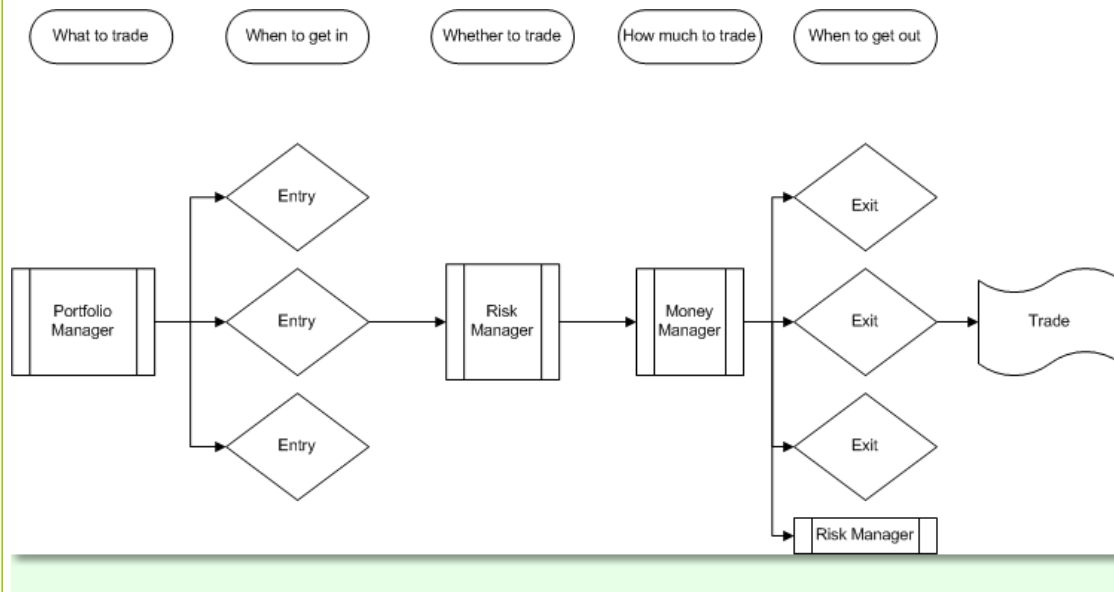
After you have explored these scripts and their use in building new Trading Blox, you can explore the full set of scripts described in the [Script Reference](#).

Section 2 – Process Flow

It helps to think of Trading Blox in terms of a process flow where the individual Trading Blox are part of a larger process which determines which markets to enter and when to exit positions, the system.

Starting with the entire portfolio of available markets, the Portfolio Manager Block filters those markets to determine which ones are available for trading on a given day, the Entry Blox create entry orders when the systems entry conditions have been met. Once an entry order has been created, the Risk Manager Block determines whether or not a particular order should be taken by assessing its affect on overall risk. If the Risk Manager Block determines the trade should be taken, the Money Manager Block looks at entry risk and other market factors to determine the size of the order (i.e. number of shares or contracts).

Orders that result in fills based on the subsequent market pricing will result in simulated positions. For each market that has a position on a given day, Trading Blox calls the Exit Blox to create potential exit orders. The Risk Manager Block is also given a chance to reduce position size or change stops each day in response to overall market risk.



Section 3 – Simulation Loop

A Trading Blox Builder simulation emulates actual trading as closely as possible. In order to create positions, you must enter orders with the broker object. Trading Blox Builder will determine if those orders would have been filled based on the data for the instruments for those orders.

One thing to remember is that Trading Blox Builder helps keep you out of trouble by only allowing access to data you would really have for making trading decisions. For example, before the markets open when entering orders, the instrument's current data is for the previous trading day. This is what happens in real life, you don't have access to today's data until the end of the day.

This has implications for the current dates of the instrument and test objects.

Each day before trading begins the test object date is set to the current date while the instrument's date is still set to the previous trading day's date. For example, on a Monday the test date might be 2006-04-10 while the instrument's date might be 2006-04-07 (the previous Friday).

The [comprehensive simulation loop](#) is described in the script reference section.

Section 4 – Comprehensive Simulation Loop

If you find yourself asking, "When do scripts get executed?" the following rather complicated section shows exactly the algorithm used by Trading Blox Builder during a simulation. This is what we refer to as the "Simulation Loop".

When running a test, Trading Blox Builder will call scripts according to the algorithm defined below.

One thing to remember is that Trading Blox Builder helps keep you out of trouble by only allowing access to data you would really have for making trading decisions. For example, before the markets open when entering orders, the instrument's current data is for the previous trading day. This is what happens in real life, you don't have access to today's data until the end of the day.

This has implications for the current dates of the instrument and test objects.

Each day before trading begins the test object date is set to the current date while the instrument's date is still set to the previous trading day's date. For example, on a Monday the test date might be 2006-04-10 while the instrument's date might be 2006-04-07 (the previous Friday).

After all the orders for the day have been entered and just before Trading Blox Builder starts to process orders to see if they have been filled, Trading Blox Builder moves the instrument's date to match the current date if there is data in the instrument for this day.

The test runs from the first real trading day after the start date of the test (`test.currentDay = 1`) to the end date of the test, for all weekdays.

Scripts specific to instruments like entry and exit are not run for an instrument on holidays or other days without data

Scripts that are not instrument specific, or require input from all instruments, run on all weekdays

Lines listed in **red** describe actions that Trading Blox Builder performs that either affect or rely on actions performed by scripts.

Multiple scripts of the same type, in different blox, will run in **Alphabetical Order** based on the block name (case sensitive) always. So if there are 10 blox each with a Before Trading Day script, the scripts will run in alphabetical order according to the block name.

Simulation Loop

```

for ( each block in all systems )
  call Before Simulation script
next ( block )

for ( each test (parameter step) in the simulation )

  setup parameters and reset variables to default

  for ( each block in all systems )
    call Before Test script
  next ( block )

  for ( each day in the test )

```

Set `test.currentdate` = test date and `test.currenttime` = first testing time
 Set `instrument.date` and `instrument.time` = the date/time of the bar prior to the test date/time

call Before Trading Day script for the Global Suite System, if available.

for (each **system**)

for (each **instrument** in the portfolio that is primed)
 call Rank Instruments script

Sort the instruments by long and short ranking

for (each **instrument** in the portfolio that is primed)
 call Filter Portfolio script

for (each **block** in system)
 call Before Trading Day script
 for (each **instrument** in the portfolio that is primed)
 call Before Instrument Day

next (**system**)

Intra-day loop start

for (each **system**)
 call Before Bar script
 next (**system**)

for (each **system** place all the orders with the broker)
 for (each **instrument** in the portfolio that is primed and has trade data on the trading date/
 time)
 for (each **entry/exit** block in system)
 call Exit Orders script only if there is a position
 call Entry Orders script every time
 call Unit Size script when order is created by broker object
 call Can Add Unit script to check if trade is allowed
 next (**system**)

call Before Order Execution script for the Global Suite System, if available.

for (each **system**)
 call Before Order Execution script
 next (**system**)

Set `instrument.date` = test date and `instrument.time` = test.time for all instruments in system
 portfolio

After this point there is full access to instrument data for test date/time

for (each **system** process the orders and fill based on actual market activity)
 for (each **instrument** in the portfolio that is primed and has trade data on the trading date/
 time)
 call Update Indicators scripts

for (each **on-open exit order** that has been created by the broker object)
 if order is filled based on price bar data
 call Can Fill Order to see if the order can be filled

if filled call Exit Order Filled

Insert Actual Broker Positions for "Open" execution type if required

for (each **on-open entry order** that has been created by the broker object.)

if order is filled based on price bar data

call Can Fill Order to see if the order can be filled

if filled call Entry Order Filled

for (each **Entry Block** in the system)

for (each **instrument** in the portfolio that is primed and has trade data on the trading date)

call After Instrument Open script if present

for (each **stop or limit exit order** that has been created by the broker object)

if order is filled based on price bar data

call Can Fill Order to see if the order can be filled

if filled call Exit Order Filled

Insert Actual Broker Positions for "Bar" execution type if required

for (each **stop or limit entry order** that has been created by the broker object.)

if order is filled based on price bar data

call Can Fill Order to see if the order can be filled

if filled call Entry Order Filled

for (each **on-close exit order** that has been created by the broker object)

if order is filled based on price bar data

call Can Fill Order to see if the order can be filled

if filled call Exit Order Filled

Insert Actual Broker Positions for "Close" execution type if required

for (each **on-close entry order** that has been created by the broker object.)

if order is filled based on price bar data

call Can Fill Order to see if the order can be filled

if filled call Entry Order Filled

Update Equity and Risk Statistics for system

next (**system** fill process)

for (each **system**)

call After Bar script

next (**system**)

Intra-day loop end -- increment date/time by test.timeIncrement and loop until day is finished

for (each **system** do after daily trading)

for (each **instrument** in the portfolio with an open position)

for (each **block** in system)

call Adjust Stops

call Initialize Risk Management

```
for ( each instrument in the portfolio with an open position )  
    call Compute Instrument Risk
```

```
call Compute Risk Adjustments
```

```
for ( each instrument in the portfolio with an open position )  
    call Adjust Instrument Risk
```

```
Update Equity and Risk Statistics for system again
```

```
for ( each block in the system )  
    for ( each instrument in the portfolio that is primed )  
        call After Instrument Day  
        call After Trading Day
```

```
next ( system )
```

```
call After Trading Day for the Global Suite System
```

```
Update Equity and Risk Statistics for test  
Compute end-of-day, month, and year statistics as appropriate for all systems and test
```

```
next ( day in test )
```

```
Closeout open positions on the close of the last day if not generating orders
```

```
for ( each block in all systems )  
    call After Test script
```

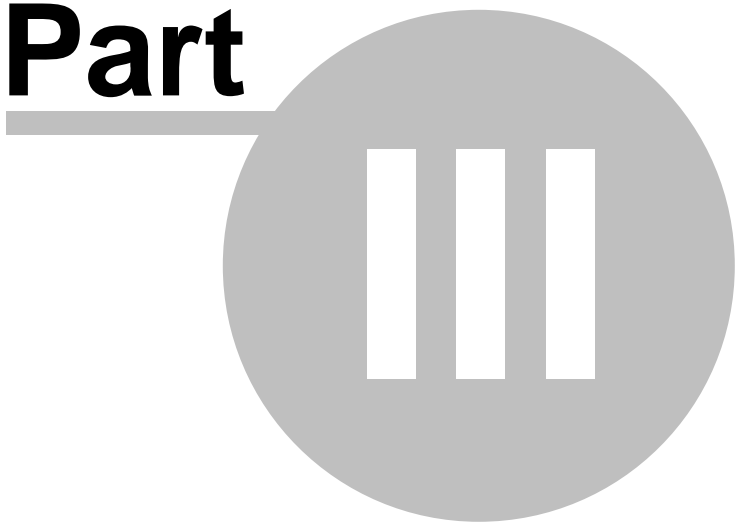
```
if generating orders  
    Generate Orders
```

```
next ( test in simulation )
```

```
for ( each block in all systems )  
    call After Simulation script
```

Blox Module Reference

Part



Part 3 – Blox Module Reference

Blox Modules

Blox are system modules that encapsulate trading ideas. Most of the Blox are self-contained parts of a trading system designed to be connected with other Blox as a component part of a trading system method. Some Blox can access data outside of their module, and outside of their system when their scope settings are set for external access.

The basic components of a trading idea are:

- **Parameters** - used by indicators to determine their specific computation result, for example: the number of days in a moving average. Not all Blox require parameters when the numerical basis for internal calculations is coded into the scripted code.
- **Indicators** - used by the rules as indicators of market conditions, moving averages, RSI, ADX, etc. Many indicators are available within the Indicator section of a Blox. Those not available in that area can be created by entering the source code required for an indicator calculation result.
- **Rules** - used to determine when to enter or exit; how much to buy or sell, or how much risk is too much, buy on moving average crossover, what instruments to allow and other ideas. A rule can be as simple as Buy If **RSI** > 55 etc.

By encapsulating trading ideas into a stand-alone Blox module, a package is created that can easily be linked to one or more systems that need the trading idea contained within the Blox. Blox are trading objects, and while these objects only need to be created once, they can be many times by other systems to simplify the creation of different system methods.

Trading Blox:

Trading Blox Builder includes the following Blox:

Script Name:	Description:
Entry	responsible for creating entry orders
Exit	responsible for creating orders to exit existing positions
Portfolio Manager	used to filter the instruments available to the system
Money Manager	used to set the size of a trade for position sizing
Risk Manager	used for filtering entry trades based on risk thresholds, adjusting stops, and reducing or exiting positions if necessary to reduce overall portfolio risk
Auxiliary	used to create custom indicators and statistics

Section 1 – Blox Types

Enter topic text here.

1.1 Portfolio Manager

The *Portfolio Manager* is used to filter the instruments available to the system. This is analogous to a screen for stock trading. The purpose of the *Portfolio Manager* block is to indicate to the system which instruments are to be traded on a particular day. If you want all instruments included, then don't use a *Portfolio Manager* block. You can also use this to set whether an instrument is allowed to trade just long, just short, or both. The built-in Trade Direction Portfolio Manager does just this.

First the *Rank Instruments* script is called once for each instrument. This allows you to use the [Ranking Functions](#) of the instrument Trading Object (i.e. `instrument.SetLongRankingValue` and `instrument.SetShortRankingValue`) to set the value which will be used by Trading Blox to rank the various instruments.

Then Trading Blox sorts and ranks each instrument based on the ranking value provided by the Rank Instruments script. The long ranking value is sorted highest to lowest, while the short ranking value is sorted lowest to highest. The respective rank is put in the [Ranking Properties](#).

Finally *Filter Portfolio* is called once for each instrument so you can use appropriate Instrument [Trade Control Functions](#) like `instrument.AllowAllTrades` or `instrument.DenyAllTrades` as necessary based on the current instrument rank.

Note that if there is no scripting code in the Filter Portfolio script, then the ranking will not take place. Some code in the Filter Portfolio script is required in order for the ranking to take place, and the rank to be determined and defined.

See Also

[Ranking Properties](#)

[Ranking Functions](#)

[Trade Control Properties](#)

[Trade Control Functions](#)

1.2 Entry

The Entry Block is responsible for creating entry orders or orders to enter a new position.

NOTE: Scripts shown as instrument scripts are not run on holidays or other days where there is no instrument data.

1.3 Exit

The Exit Block is responsible for creating orders to exit existing positions. It runs only when the instrument has a position of LONG or SHORT.

1.4 Money Manager

The Money Manager Block is used to set the size of a trade for position sizing. It includes the Unit Size script.

Note that the Unit Size script can be Added to any block if an integrated money manager is desired.

1.5 Risk Manager

A Risk Manager block is used for filtering entry trades based on risk thresholds, adjusting stops, and reducing or exiting positions if necessary to reduce overall portfolio risk. It includes the following scripts:

Block Type	Script Type	Day	Instrument	Position	Called When
Risk Manager					
	Before Test				start of test
	Initialize Risk Management	•			end of day
	Compute Instrument Risk			•	end of day
	Compute Risk Adjustment	•			end of day
	Adjust Instrument Risk			•	end of day
	Can Add Unit		called by broker when for entry orders		
	Can Fill Order		called as order is about to be filled		

NOTE: As indicated above, the instrument-specific scripts associated with the Risk Manager loop over instruments with *existing positions*. They do not loop over instruments that are out of the market.

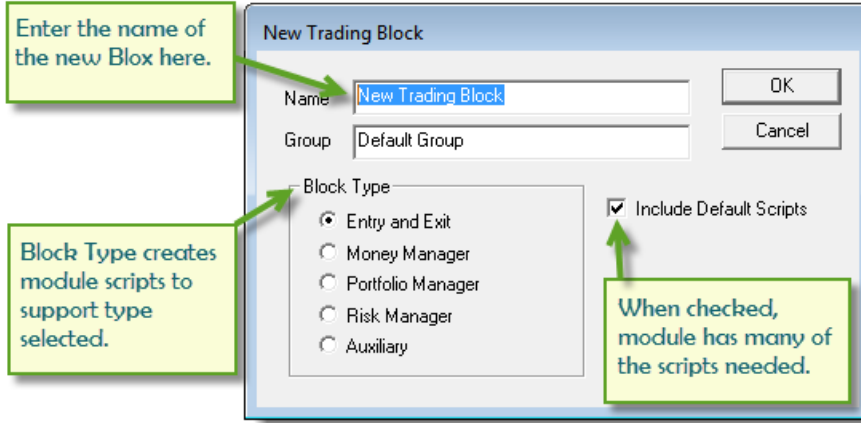
The Can Add Unit and Can Fill Orders scripts can be added to any block. In this way multiple blocks can process these scripts to determine if an order can be placed or filled.

1.6 Auxiliary

Auxiliary Blox can be used to create custom indicators or statistics. An Auxiliary block might have the Update Indicators script, or perhaps other scripts like the Can Add scripts or Before/After trading scripts.

Section 2 – Blox Script Access


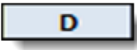
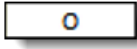

When a new block module is created there is an option to include additional default scripts in addition to the required script sections that are required in the Block-Type selected:



This next table provides the matrix of the required block modules, and the script names of added default scripts. Adding more scripts to a module is easy and often is required. However, if you add a defining block script to a module it is likely to have additional affects not intended if the added script is a defining script for the Portfolio Manager, Money Manager and Risk Manager.

Order	Script Section Name:	Portfolio Manager	Entry	Entry, Exit	Entry, Exit, Money Manager	Money Manager	Risk Manager	Auxiliary
1	Before Simulation	D	D	D	D	D	D	D
2	Before Test	D	D	D	D	D	D	D
3	Rank Instrument	D						
4	Filter Portfolio	D						
5	Before Trading Day	D	D	D	D	D	D	D
6	Before Instrument Day	D	D	D	D	D	D	D
7	Before Bar	I	I	I	I	I	I	I
8	Exit Orders			D	D			
9	Entry Orders		D	D	D			
10	Unit Size				D	D		
11	Can Add Unit		D	O	D	O	D	O
12	Before Order Execution		O	O	O	O	O	O
13	Update Indicators	O	O	O	O	O	O	D
14	Can Fill Order		D	D	D	O	O	O
15	Exit Order Filled			D	D	O	O	O
16	Entry Order Filled		D		D	O	O	O
17	After Instrument Open		O	O	O			O
18	After Bar	I	I	I	I	I	I	I
19	Adjust Stops			O	O		O	O
20	Initialize Risk Management						D	
21	Compute Instrument Risk						D	
22	Compute Risk Adjustments						D	
23	Adjust Instrument Risk						D	
24	After Instrument Day	D	D	D	D	D	D	D
25	After Trading Day	D	D	D	D	D	D	D
26	After Test	D	D	D	D	D	D	D
27	After Simulation	D	D	D	D	D	D	D
28	<Custom Name Script>	O	O	O	O	O	O	O

Legend: Description:

	Cells where this is displayed locate the script names that define the type of module.
	Script name section added when the "Include Default Scripts" option is enabled.
	Can Fill Order scripts only execute when either an entry has been filled in the market, and when an exit order has closed an active position, or any of the position's unit segments. Custom script execute when their names are called using the Script Object's Execute function
	Intraday script section executes for each intraday record.

Section 3 – Blox Script Timing

Script Execution Timing Table					
Order	Script Section Name:	Runs Once Each Simulation	Runs Once Each Test	Runs Once Each Bar	Runs Once Each Instrument
1	Before Simulation	Y			
2	Before Test		Y		
3	Rank Instrument				Y
4	Filter Portfolio				Y
5	Before Trading Day			Y	
6	Before Instrument Day				Y
7	Before Bar			Y	
8	Exit Orders				X-1
9	Entry Orders				Y
10	Unit Size				E-1
11	Can Add Unit				E-1
12	Before Order Execution			Y	
13	Update Indicators				Y
14	Can Fill Order				EX-1
15	Exit Order Filled				X-2
16	Entry Order Filled				E-2
17	After Instrument Open				Y
18	After Bar			Y	
19	Adjust Stops				X-3
20	Initialize Risk Management			Y	
21	Compute Instrument Risk				Y
22	Compute Risk Adjustments			Y	
23	Adjust Instrument Risk				Y
24	After Instrument Day				Y
25	After Trading Day			Y	
26	After Test		Y		
27	After Simulation	Y			
28	<Custom Name Script>				

Table on the left shows all the Trading Blox module scripts.

Each column descriptions shows the details of how often each script is executed for each of the script names.

Only script sections with Trading Blox Basic statements will execute. Script that are blank are not executed.

Order number listed to the left of each script name is a relative indication of the order in which the scripts might execute most often.

Scripts with a value other than a blank cell, or a Y have exceptions to when they will execute.

Each exception is describe in the legend at the bottom of the table.

Legend:	Execution Description:
E-1	Unit Size and Can Add Unit scripts will only execute after a Broker Object function call that generates an order to enter the market with a new position or a new unit. Only new entry orders pass through these script sections. See order.isEntry
E-2	Entry Order Filled scripts only execute after the Can Fill Order script section completed its execution and allowed the order to continue, and when an entry order has filled in the market. See order.continueProcessing
EX-1	Can Fill Order scripts only execute when either an entry has been filled in the market, or when an exit order has closed an active position or any of a position's unit segments.
X-1	Exit Orders only execute when there is an active instrument position, and it only executes when the instrument with the active position is sequenced into context.
X-2	Exit Order Filled scripts only execute after the Can Fill Order script section has completed its execution and allowed the order to continue, and when an

	active position or any of its units have been closed.
X-3	Adjust Stops scripts executes when there is an open position ahead of the risk scripts so that risk information used can be collected and changes can be implemented.
Custom Scripts	<Custom-name-scripts> only execute when called. They can be called from any another script, standard or custom, and when they complete their execution they return to line in the script from which they were called.

Links:
[order.continueProcessing](#), [order.isEntry](#), [System Object](#)

See Also:
[Script Section Type Details](#), [Global Script Timing Table](#)

Section 4 – Global Script Timing

Trading Blox Suites support simultaneous testing of multiple systems so the overall strategy can include multiple trading methods on the same or different classes of portfolios with the same or different instruments. Suites with one or more Systems will each contain a collection of Blox modules. Each block in a system has a purpose, like Portfolio Management, Entry and Exit signals, Money Management, Risk Management, and Auxiliary blox used to support a special indicator, custom function, reporting method or any other idea useful to a system's operation.

Within each blox are one or more script section, and some scripts can have some of the same script names as other modules in the system. Scripts are executed in a controlled sequence across all the systems. Scripts within a system with the same name are executed in the order in which they are listed in the System Editor blox list in the [Script Section Type Details](#) topic. Script types in the others systems in a suite will execute the same script names in the order in which the system was added to the suite.

Suites can also contain a Global System Scripts (GSS) when the name of the system is an identical match to the Suite's name. Global System won't be allowed to execute all the scripts that any of the blox scripts can execute. Instead, a Global System will only execute the scripts that are useful for managing or adjusting the operation of the standard scripts in the suite. Table shown below list the types of scripts that can be used with a Global System.

Parameters for each system in a suite are displayed in a separate system name tab. Each selected tab only shows the parameters for the selected system. Parameters for Global Systems are display at the bottom of the Global Settings tab just after the Equity Manager parameters.

Script Sequencing:

System index values determine the order in which the same name scripts in each system are executed. For example, the Before Test script in the system indexed at 1 will always execute ahead of the Before Test of the system indexed at 2. This logic will hold for the remaining systems in the suite until there are no more system index values.

Suites that include GSS system modules will align execution with their matching system script names. GSS scripts will either execute ahead or after the system scripts according to the execution placement shown in the following table. Knowing which GSS script names execute ahead or after the system scripts is necessary to understand how they can be used to gather or send information to or between systems.

Global Suite					
Common Blox Script Execution Sequence					
TB: v4.2.4.x 2-July-2013					
Script Execution Order	System Index #: 1	System Index #: 2	System Index #: x	Global System Index #: 0	Global Scripts Execute
1 2 3 x	Before Simulation	Before Simulation	Before Simulation	Before Simulation	Before
1 2 3 x	Before Tests	Before Tests	Before Tests	Before Tests	Before
1 2 3 x	Before Trading Day	Before Trading Day	Before Trading Day	Before Trading Day	Before
1 2 3 x	Before Bar	Before Bar	Before Bar	Before Bar	Before
1 2 3 x	Can Add Unit	Can Add Unit	Can Add Unit	Can Add Unit	After
1 2 3 x	Before Order Execution	Before Order Execution	Before Order Execution	Before Order Execution	After
1 2 3 x	Can Fill Order	Can Fill Order	Can Fill Order	Can Fill Order	After
1 2 3 x	Entry Order Filled	Entry Order Filled	Entry Order Filled	Entry Order Filled	After
1 2 3 x	Exit Order Filled	Exit Order Filled	Exit Order Filled	Exit Order Filled	After
1 2 3 x	After Bar	After Bar	After Bar	After Bar	After
1 2 3 x	After Trading Day	After Trading Day	After Trading Day	After Trading Day	After
1 2 3 x	After Test	After Test	After Test	After Test	After
1 2 3 x	After Simulation	After Simulation	After Simulation	After Simulation	After

Global & System Script Execution Timing

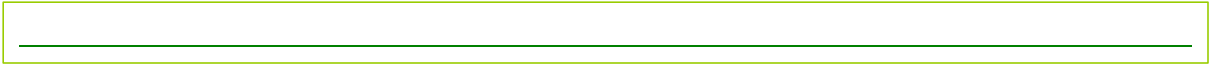
All the system in suite can access other systems using the [test.SetAlternateSystem](#) function. When a module's script has access to another system it can access information in that systems to obtain and provide information.

Links:

[System Object](#), [Test Miscellaneous](#)

See Also:

[Blox Script Access](#), [Blox Script Timing](#), [Script Section Type Details](#)



Section 5 – Script Section Type Details

Trading Blox assigns the blox Module-Type name based upon the name of some scripts contained in the blox. Type-Classifying scripts are used because those script names are recommended for supporting how various methods are placed to support how the system operates. Some script names are only executed when position is active, or when a broker object statement is executed.

Static

Portfolio Manager

Trade Direction Portfolio Mana...

Entry

Turtle Entry Exit R-Labels

Exit

Chandelier Exit
Turtle Entry Exit R-Labels

Money Manager

Fixed Fractional Money Manager

Risk Manager

Correlation Risk Manager

Auxiliary

Plot Stop Price
Statistics Robust

System Blox Type:	Blox Allowed:	Type Class Scripts (at least one required):
Portfolio Manager	1	Rank Instruments Filter Portfolio
Entry	No Limit	Entry Orders
Exit	No Limit	Exit Orders
Money Manager	1	Unit Size
Risk Manager	1	Initialize Risk Manager Compute Instrument Risk Compute Risk Adjustments Adjust Instrument Risk
Auxiliary	No Limit	Any Script Not Listed Above

Some modules can contain some of the same scripts as other modules in the system. Scripts with the same name will be execute at the same time but the order in which they execute will be based upon the order in which the blox is listed in the System Editor blox listing.

In the Blox Type sections shown on the left some of the modules shown in the sections have some of the same type script sections. When these scripts are executed their name position in the type section window will determine the order in which these same type name scripts are executed.

For example, in the Exit section shown in the image on the left, there are two blox names listed. Chandelier Exit, a progressive protective exit price module, and Turtle Entry Exit R-Labels, the Turtle Entry Exit standard Exit Order section for protecting and exiting positions. When this system execute, the order of Exit Orders in each of the two blox modules shown will happen so the Chandelier Exit Exit Orders script section will execute ahead of the Turtle Entry Exit Exit Orders section.

To change the order in which a script section with the same names is executed, the name of the blox modules must be changed so as to change how its ascending alphabetically sorted position appears in the type section.

Modules in all of the script section in which more than module of that type are allowed to be placed within a system will follow the same alphabetical blox name order.



Section 6 – Scripts Common to Many Blox

Block Type	Script Type	Bar	Day	Instrument	Position	Called When
Entry						
	Before Simulation					start of simulation
	Before Test					start of test
	Before Trading Day		•			start of day
	Before Instrument Day			•		before each instrument day
	Before Bar	•				before each intraday bar
	After Instrument Open			•		after open
	Before Order Execution					before the orders are executed in the market and filled or killed
	After Bar	•				after each intraday bar
	After Instrument Day			•		after each instrument day
	Adjust Stops			•	•	for each position
	After Trading Day		•			end of day
	After Test					end of test
	After Simulation					end of simulation

NOTE: Scripts shown as instrument scripts above are not run on holidays or other days where there is no instrument data. Scripts that are instrument independent are run every weekday regardless of holidays.

Section 7 – Script Section Descriptions

All the Trading Blox script sections are listed in this table. Each script section provides execution that is timed to provide access to data at a specific phase of the [Simulation Loop](#) processing. Some script sections are only executed under specific circumstances and when called by other scripts or functions.

For example, [Exit Orders](#), and [Adjust Stops](#) are only executed when there the `instrument.position` property shows a **Long** or a **Short** position is active. [Before Instrument Day](#) and After Instrument Day scripts are only executed when there is a data record available for the date being processed. See [Blox Script Timing](#) for more information about script timing.

Trading Blox Script Sections:

Script Name:	Description:
Before Simulation	
Before Test	
Rank Instruments	
Filter Portfolio	
Before Trading Day	
Before Instrument Day	
Before Bar	
Exit Orders	
Entry Orders	
Unit Size	
Can Add Unit	
Before Order Execution	
Update Indicators	
Can Fill order	
Exit Order Filled	
Entry Order Filled	
After Instrument Open	
After Bar	
Adjust Stops	
Initialize Risk Management	
Compute Instrument Risk	
Compute Risk Adjustment	
Adjust Instrument Risk	
After Instrument Day	
After Trading Day	
After Test	
After Simulation	

For further information on a specific block type or block script click on block type or script type in the lists below. Some of the scripts are [Scripts Common to Many Blocks](#). A general description of each of these [Common Scripts](#) follows, however in some cases the descriptions in the [Blox Reference](#) section for each block type will have information specific to that block type if applicable.

7.1 Before Simulation

The Before Simulation Script is run once for a simulation, even a simulation that includes many different parameter stepping tests. This script is often used to load external data which will be used for every test in a multi-parameter-step simulation:

The following is an example of a Before Simulation script:

```
VARIABLES: instrumentCount TYPE: Integer
VARIABLES: externalFileName TYPE: String

' Get the instrument count.
instrumentCount = system.totalInstruments

' Loop initializing each instrument.
FOR index = 1 TO instrumentCount STEP 1

    ' Set the portfolio instrument.
    portfolioInstrument.LoadSymbol( index )

    ' Get the symbol for the instrument.
    externalFileName = portfolioInstrument.symbol + "_ExternalData.csv"

    ' Print out the file name.
    PRINT "Loading External File: ", externalFileName

    ' Load the external data.
    IF NOT portfolioInstrument.LoadExternalData( externalFileName
        "BetaDate", "Beta1", "Beta2" ) THEN
        PRINT "Could not Load External Data for ", externalFileName
    ENDIF
NEXT
```

This example script loads external data by looking over the system's portfolio and then loading external files using the instrument's symbol.

This script does not have access to the test parameters, since it does not represent any one test. It is run before the first test is initialized.

7.2 Before Test

The Before Test Script is called once at the beginning of each test in a simulation. It can be used to initialize variables used during the simulation.

This script has access to the current parameter settings for the test.

7.3 Rank Instruments

The *Rank Instruments* script is used to set the long and short ranking value using the [Ranking Functions](#) of the Instrument Trading Object. If your strategy requires a screen or 'trade only the top x strongest' or some other type of logic that requires ranking the instruments, this is where you set the value to be used in the sort.

Ranking Values and Instrument Ranks

There are separate long and short ranking values. Ranking values are used by Trading Blox Builder to determine the relative ranking of each instrument. After the Rank Instruments script has been called for each instrument in the current Portfolio, Trading Blox Builder sorts the instruments highest to lowest using the long ranking value and then lowest to highest using the short ranking value. It then sets the rank for each instrument based on its position after being sorted by the ranking value.

For long ranking values, the highest values will result in the lowest rank. So an instrument with a rank of 1 represents the instrument with the highest long ranking value for that day.

For short ranking values, the lowest values will result in the lowest rank. So an instrument with a rank of 1 represents the instrument with the lowest short ranking value for that day.

This approach lets you use a single measure for both long and short trades. For example, a strength measure would result in higher strength instrument's rated lower for long trades and lower strength instrument's rated lower for short trades.

Example code for the Rank Instruments script:

```
' Set the long ranking value for this instrument
instrument.SetLongRankingValue( rsi )
```

7.4 Filter Portfolio

This script allows you to indicate whether an instrument should be included for testing or not using the [Trade Control Functions](#) functions of the instrument trading object. Note that if there is no scripting code in the Filter Portfolio script, then the ranking will not take place. Some code in the Filter Portfolio script is required in order for the ranking to take place, and the rank to be determined and defined.

You can check the longRank or shortRank in relation to a fixed rankThreshold type parameter (ie x number of instrument).

You can also check the rank vs. the total number of instruments in the portfolio to trade only the top x % of the instruments. This value is the [system.totalInstruments](#) property.

You can also check the rank vs. the total number of trading instruments. This value is the [system.tradingInstruments](#) property.

Once this script is finished, it sets the [system.canTradeInstruments](#) to the total number of trading instruments that can trade today based on this script logic.

Example

```
' If this instrument is in the top rankings...
IF instrument.longRank <= rankThreshold THEN

    instrument.AllowLongTrades
ENDIF
```

7.5 Before Trading Day

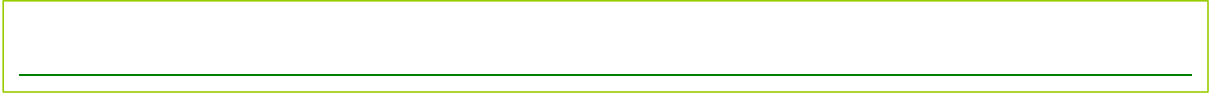
The Before Trading Day Script is called once at the beginning of each trading day in a simulation. It can be used to reset Block Permanent variables values each day or for debugging. This script does not have access to Instrument properties or Instrument Permanent variables.

7.6 Before Instrument Day

This script is called once per day for each trade record in its instrument file, before the entry and exit orders are placed into the market. It runs for all instruments and has instrument and IPV and Indicator access.

Use [instrument.tradesOnTradeBar](#) property when you need to know if the instrument has a trade record for this date.

7.7 Before Bar



7.8 Exit Orders

The *Exit Orders* script is called once each day for each instrument that has a position. The *Exit Orders* script is responsible for creating orders to exit existing positions.

The RSI Trend Exit block uses the following code in the *Exit Orders* script.

```
' -----
' Exit Position if RSI crosses Threshold
' -----

IF instrument.position = LONG AND
    relativeStrengthIndex <= exitThreshold THEN

    ' Exit the position.
    broker.ExitAllUnitsOnOpen
ENDIF

IF instrument.position = SHORT AND
    relativeStrengthIndex >= (100 - exitThreshold) THEN

    ' Exit the position.
    broker.ExitAllUnitsOnOpen
ENDIF

' -----
' Enter stop if "holdstops" is true
' -----

IF holdStops THEN

    broker.ExitAllUnitsOnStop( instrument.unitExitStop )
ENDIF
```

This sample script has two common features. First, a check against the open position because of the differing logic for LONG and SHORT positions.

Second, the placing of stop orders using the `instrument.unitExitStop` price. You will need logic like this if you wish to have stops which are in effect for the duration of a trade. Trading Blox Builder requires that you place new orders for stops each day.

7.9 Entry Orders

Called each day for each instrument at the beginning of the day. The *Entry Orders* script is the recommended place for creating orders to enter new positions.

The following example Entry Orders script was taken from the [Creating a New System](#) tutorial at the beginning of this manual:

```
IF ( macdIndicator > 0 ) AND ( instrument.position <> LONG ) THEN
    ' Two conditions must be met - MACD above 0 and we are not long.
    ' If both are met, we enter long.
    broker.enterLongOnOpen
ENDIF

IF ( macdIndicator < 0 ) AND ( instrument.position <> SHORT ) THEN
    ' If we are not short and the MACD is below 0, enter short.
    broker.enterShortOnOpen
ENDIF
```

This script has the broker object enter new long and short positions depending if the `macdIndicator` is positive or negative.

The script has a common feature of checking for existing positions before entering orders, the check for:

```
instrument.position <> LONG
```

and

```
instrument.position <> SHORT
```

which is part of the `IF` statement.

Unless you wish to add to a position and thereby create additional units for your position, you should check if there is already a position before entering orders.

7.10 Unit Size

This script is only called when a Broker Entry Function is executed. During the execution of an Entry-Function the Broker object will call the Unit Size script so that the logic in the attached Money Manager module can process the Entry information.

How the order information is handled in the **UNIT SIZE** script can vary because different systems require different order sizing methods. At its simplest form of Unit Sizing a quantity of contracts or shares are added to the order's quantity property. Fixed Fractional Sizing, Account Fixed Allocation, or any other method used to determine sizing will usually have a qualifying process in the logic to determine if the allocation or risk expense will allow a quantity that is larger than the `instrument.roundlot` value. When the quantity calculation falls below this minimum instrument quantity for establishing an order, the `order.Reject("Reason")` function is executed. When orders are rejected their `instrument.symbol`, `instrument.date` and reason for the rejection are placed into the Trading Blox **Filter.Log** file. In addition the `order.continueProcessing` property will be changed from its default value of **TRUE** to **FALSE**.

Orders completing their processing in **UNIT SIZE** are then processed in the **CAN ADD UNIT** script section so that orders that are not rejected can be managed by that script section so as to determine if the order can be sent to the brokerage for processing. Order can be filtered, adjusted and rejected in the **CAN ADD UNIT** script. Order rejections in this script are handled and reported in the same way as they are rejected in the **UNIT SIZE** script.

For more information on **UNIT SIZE** scripts review the information shown in [Order Sizing](#).

Note:

`order.continueProcessing` and `order.processingMessage` properties will be set when the `order.SetQuantity` function is called. When a quantity is set, the equity, volume, Portfolio Manager, and Risk Manager filters are called and checked. If any of these fail, then the status will be available at this time.

Links:

[Broker](#), [Data Properties](#), [Entry Order Functions](#), [Order Object](#), [Order Sizing](#)

See Also:

7.11 Can Add Unit

Called by Trading Blox Builder as the result of a call to the broker statement. This script can be used to determine if the order will be placed or not. The *Can Add Unit* script can be used to implement risk limits. For example, the Turtle System's Correlation Limiter uses the *Can Add Unit* script to limit the number of units which can be taken based on market correlation.

The *Can Add Unit* script is only called for entry orders created by calls to the Broker object in scripting. It is not called for entries due to Actual Broker Positions.

The entire order object is available. It contains information about this potential order to be placed. See the [Order](#) object for available properties and functions.

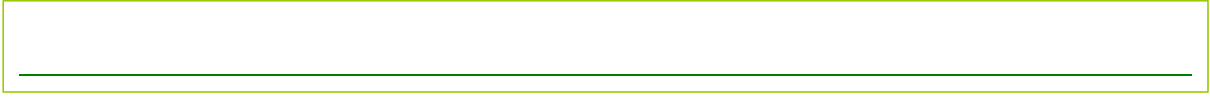
If multiple Can Add scripts are in a system, the order will be rejected if ANY Can Add script rejects the order.

The following is an example of a *Can Add Unit* script:

```
IF instrument.closelyCorrelatedLongUnits >=
maxCloselyCorrelatedUnits OR
    instrument.looselyCorrelatedLongUnits >=
maxLooselyCorrelatedUnits THEN

    order.Reject( "Too Many correlated units" )
ENDIF
```

7.12 Before Order Execution



7.13 Update Indicators

The *Update Indicators* script is run for each instrument, as it is moved from one bar to the next. For daily data this means at the beginning of the instrument day (not the test day) and for intraday data this means at the beginning of the instrument date/time (not the test date/time).

If you update your custom indicators here the values will be available to use in all other scripts and blox, like the Adjust Stops script, the After Instrument Day script, and of course the Entry Orders and Exit Orders scripts.

7.14 Can Fill Order

Called as part of the fill process, the *Can Fill Order* script lets you determine by examining the trade day's price data, whether this order would be filled or not. This only gets called if Trading Blox has determined that this order should be filled based on a bar's price data. You can call `order.Reject` to override this determination. In addition, you can override the fill price, quantity, and stop in certain circumstances.

The entire order object is available properties and functions. See the [Order](#) object for more information.

If multiple Can Fill scripts are in a system, the order will be rejected if ANY Can Fill script rejects the order.

NOTE: The following order types **CANNOT** be canceled:

- **Exit Orders** placed by Trading Blox Builder to exit all open positions at the end of the test
- **Entries or Exits** placed for Actual Broker Positions
- **Exit Orders** placed automatically as a result of a reversal - i.e. the position is long and a new short entry order triggers

Example

The following is a Can Fill Order script that checks for max margin.

```
IF order.IsEntry THEN
    IF instrument.IsFuture THEN
        newMarginValue = order.quantity * instrument.margin
    ENDIF
    IF instrument.IsStock THEN
        newMarginValue = order.Quantity * instrument.close *
instrument.conversionRate * instrument.stockSplitRatio
    ENDIF
    IF test.totalMargin + newMarginValue > test.totalEquity THEN
        order.Reject( "Over the margin limit" )
    ENDIF
ENDIF
```

7.15 Exit Order Filled

The *Exit Order Filled* script is called each time an exit order is filled. This script lets you perform any calculations or take actions that depend on the fill price or fill dates for an order.

This script has full access to the [Order](#) object properties.

All scripts of this type in a System will be called each time an exit order is filled. To check if the current block is the same as the block originating the order, use the following:

```
IF block.name = order.blockName THEN
```

7.16 Entry Order Filled

The *Entry Order Filled* script is called each time an entry order is filled. This script lets you perform any calculations or take actions that depend on the fill price or fill dates for an order.

The Turtle System uses this script to adjust stops for existing positions based on the slippage of an actual fill.

This script has full access to the [Order](#) object properties.

All scripts of this type in a System will be called each time an entry order is filled. To check if the current block is the same as the block originating the order, use the following:

```
IF block.name = order.blockName THEN
```

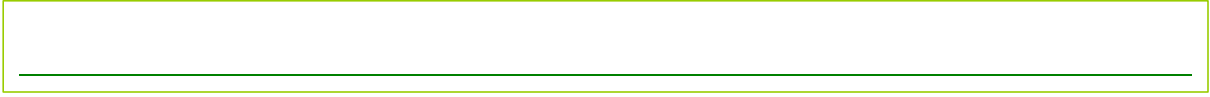
7.17 After Instrument Open

WARNING: The *After Instrument Open* script is an advanced script that is not recommended for most users. It is present to enable experienced system developers to write systems that have special order processing logic based on the relationships between the open, highs and lows.

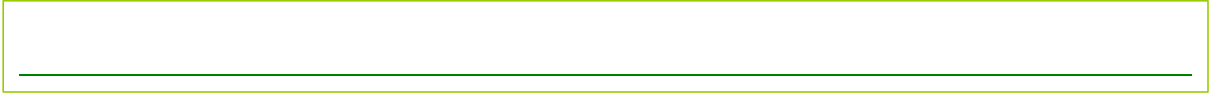
Since this script is called after the current bar is set for each instrument (see [Comprehensive Simulation Loop](#)), the entire bar's price data is available. For this reason, it is possible to write systems with postdictive errors using this script.

If you want to create orders which are based on a bar's open, we recommend using the `tradeDayOpen` instrument property in the *Entry Orders* script.

7.18 After Bar



7.19 Adjust Stops



7.20 Initialize Risk Management

Called once each day at the end of the day before the other Risk Manager scripts. The *Initialize Risk Management* script can be used to initialize portfolio-level risk settings each day.

The *Total Risk Limiter* block uses this script to initialize its `totalRisk` variable each day:

```
totalRisk = 0
```

7.21 Compute Instrument Risk

Called once for each instrument which has an existing position after the *Initialize Risk Management* script has been called, the *Compute Instrument Risk* script can be used to compute per-instrument risk and to total the risk at the portfolio level.

The *Total Risk Limiter* block uses this script to add up the risk for each instrument:

```
' Add the instrument risk to the total risk.  
totalRisk = totalRisk + instrument.currentPositionRisk
```

7.22 Compute Risk Adjustment

Called once each day after the *Compute Instrument Risk* script has been called for each instrument. The *Compute Risk Adjustment* script can be used to calculate portfolio-level adjustments which can be applied on a per-instrument basis in the *Adjust Instrument Risk* script which follows.

The *Total Risk Limiter* block uses this script to determine the amount that stops need to be moved or positions need to be reduced.

```
VARIABLES: riskPercent TYPE: Percent

IF system.tradingEquity > 0 THEN

    ' Compute the current risk.
    riskPercent = totalRisk / system.tradingEquity

    ' If the risk is above our threshold...
    IF riskPercent > maximumRiskThreshold THEN

        reductionPercent = (riskPercent - maximumRiskThreshold) /
riskPercent

    ELSE

        reductionPercent = 0.0

    ENDIF

ELSE

    reductionPercent = 0.0

ENDIF
```

In this block, the `maximumRiskThreshold` is a parameter which defines the maximum percentage risk for all open positions.

The `reductionPercent` will be used in the [Adjust Instrument Risk](#) script to reduce the position size or move the stops.

7.23 Adjust Instrument Risk

Called once for each instrument which has an existing position, the *Adjust Instrument Risk* script can be used to adjust stops and reduce or exit positions based on portfolio-level risk as computed in the *Compute Risk Adjustment* script.

The *Total Risk Limiter* block uses this script to move stops or reduce positions based on the `reductionPercent` computed in the [Compute Risk Adjustment script](#):

```
VARIABLES: quantity, reductionQuantity TYPE: Integer
VARIABLES: risk TYPE: Floating
VARIABLES: newStop TYPE: Price

' If we need to reduce risk
IF reductionPercent > 0.0 THEN

    IF reductionAlgorithm = REDUCE_POSITIONS THEN

        ' Reduce the position by this amount.
        broker.AdjustPositionOnOpen( 1.0 - reductionPercent )

    ENDIF

    IF reductionAlgorithm = MOVE_STOPS THEN

        IF instrument.position = LONG THEN

            ' Adjust the stops for each unit.
            FOR index = 1 to instrument.currentPositionUnits

                ' Determine the current risk.
                risk = instrument.close -
                    instrument.unitExitStop[index]

                ' Determine the stop that corresponds with
                ' the reduced risk.
                newStop = instrument.close -
                    ((1 - reductionPercent) * risk)

                ' Set the new stop.
                instrument.SetExitStop( index, newStop )
                broker.ExitUnitOnStop( index, newStop )
            NEXT

        ENDIF ' Long

        IF instrument.position = SHORT THEN

            ' Adjust the stops for each unit.
            FOR index = 1 to instrument.currentPositionUnits

                ' Determine the current risk.
                risk = instrument.unitExitStop[index] -
```



```
instrument.close

' Determine the stop that corresponds with
' the reduced risk.
newStop = instrument.close +
          ((1 - reductionPercent) * risk)

' Set the new stop.
instrument.SetExitStop( index, newStop )
broker.ExitUnitOnStop( index, newStop )
NEXT

ENDIF ' Short

ENDIF ' Algorithm Move Stops

ENDIF ' There is a reduction required
```

7.24 After Instrument Day

The *After Instrument Day* script is called once each day per instrument. This script can be used to calculate instrument-specific variables or custom indicators at the end of each day.

7.25 After Trading Day

The *After Trading Day* script is called once each day. This script can be used to reset values at the end of each day. This script does not have access to instrument properties or Instrument Permanent variables.

7.26 After Test

The *After Test* script is called once at the end of each test in a simulation. It is often used to PRINT values or to write to files at the end of a test.

7.27 After Simulation

The *After Simulation* Script is called once at the end of an entire simulation. It is often used to PRINT values or to write to files at the end of a test.

Blox Basic Language Reference

Part



Part 4 – Blox Basic Language Reference

Blox Basic is a full-featured scripting language that lets you control a historical trading simulation using powerful language constructs.

The following sections are included in the Blox Basic Reference:

Reference Areas:	Descriptions:
Comments	Methods available to annotate your scripts so they will be understandable later.
Debugger	A powerful tool to verifying your scripts are doing what you intended, and a way to understand what must be changed when they are not working well.
Function Reference	A comprehensive list of all the functions built into Blox Basic
Operator Reference	Shows the types of operators and expressions you can write in a Blox Basic script
Statement Reference	shows the types of statements you can write in Blox Basic

Section 1 – Basic Keywords

Trading Blox Basic's language is based upon the use of Objects. Object are data structures that provide property values, and methods for changing values. It also provides Sub-Routine and Functions for modifying information in various ways.

When viewing a Basic reference you will find that Properties use a lower case character as its first letter name reference.

Object Methods show their first letter as a upper case character for naming its method.

Sub-Routine and Functions are used without any prefix reference, and all will have a upper case first character in their name.

Functions that return a value must be assigned to a variable, or used as a value in Print, or data building process.

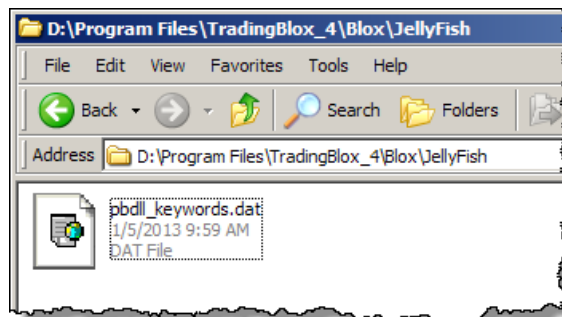
Sub-Routines create or change something, but don't return a value and be used as a stand-alone statement.

Basic KeyWords Listing:

A complete list of all of the Trading Blox Basic Keywords can be found in the file named:
pbdll_keywords.dat

This file is a simple text file that will open in any text editor like Windows NotePad. File is located in the Trading Blox installation location shown in the image on the right. This example is just a sample that has Trading Blox installed on the "D" drive. In most cases the drive location will be on the "C" drive.

File is created automatically by Trading Blox so it is always available with all recent versions of Trading Blox.



Trading Blox sub-directory example as seen in Windows

Section 2 – Colors

Selecting, applying and changing colors used on a chart, or graph is under the users control methods described in the section below. Chart area colors can be changed so displays of indicators, bars, trade details can be customized to meet changes to color backgrounds and grids. All plotted items can be colored by the user, and some chart display items can change for each bar on the chart.

Plotting Instrument Permanent Variables (IPV) Series on Charts:

Instrument Permanent Variables designated as an IPV Series-Type, have the option of being plotted on a chart.

Plotting Block Permanent Variables (BPV) Series Custom Graphs:

Block Permanent Variables that use the selection of System, Test, or Simulation Scope feature can be plotted on a **Summary Custom Graph** that appears in the **Trading Blox Summary Test Results** report.

Changing Colors:

Colors can be assigned and changed using the variable declaration and editing dialogs, or they can be changed by using the [ColorRGB](#) and [SetSeriesColorStyle](#) functions.

Coloring the plotting series for both of these series is handled by accepting the color displayed in the Plotting area on the right-side of the Variable Selection and Editing dialog, or by clicking on the the Colored button control.

When the Colored-button is clicked a matrix table of common colors will display where any of the colors can be selected by clicking on the colored square. An option to use a color not shown is available by clicking on the "**More Colors...**" button at the bottom of the color-table. When that button is clicked a "**Custom Color Selection Dialog**" appears where almost any color can be selected or created. New colors can be assigned to the white colored areas so they are made available later. **Summary Custom Charts** require the reporting option being used show it has enabled the **Custom Graph** option for the simulation method being used.

Colors assigned using the series dialog options typically use the same color across the entire plotted area. When the need to vary the color of plotting item in an **IPV** series, the [SetSeriesColorStyle](#) function can be applied anytime using scripting references during testing.

Selecting Trade Chart Preference Colors:

These preset Preferences can be used in scripting, and they can also be adjusted by users.

Trade chart Preference Default Colors:

Color Property Name:	Default Color Number:
ColorBackground	14745599
ColorUpBar	37632
ColorDownBar	213
ColorUpCandle	37632
ColorDownCandle	213
ColorCrossHair	12632256

ColorGrid	16443110
ColorLongTrade	37632
ColorShortTrade	213
ColorTradeEntry	16764057
ColorTradeExit	16751001
ColorTradeStop	14527197
ColorCustom1	0
ColorCustom2	139
ColorCustom3	16711680
ColorCustom4	16777215

User Series Array Color Control:

BEFORE TEST Script

```

' -----
'           ' B , G , R
ColorItem[ 1 ] = ColorRGB( 0, 0,255 ) ' Red
ColorItem[ 2 ] = ColorRGB( 255, 0, 0 ) ' Blue
ColorItem[ 3 ] = ColorRGB( 0,255, 0 ) ' Green
ColorItem[ 4 ] = ColorRGB( 0, 0, 0 ) ' Black
ColorItem[ 5 ] = ColorRGB( 155,133, 49 ) ' Aqua
ColorItem[ 6 ] = ColorRGB( 0, 0,192 ) ' Dark Red
ColorItem[ 7 ] = ColorRGB( 212,141, 84 ) ' Light Blue
ColorItem[ 8 ] = ColorRGB( 9,108,227 ) ' Orange
ColorItem[ 9 ] = ColorRGB( 123,117,235 ) ' Pink
ColorItem[10] = ColorRGB( 38, 162, 94 ) ' Dark Green
ColorItem[11] = ColorRGB( 122, 73, 95 ) ' Purple
ColorItem[12] = ColorRGB( 151, 72, 6 ) ' Brown
ColorItem[13] = ColorRGB( 0,255,255 ) ' Yellow
' -----

```

To use above series, consider this approach shown in [AddLineSeries](#) example:
Multi-Line Chart Example:

```

' Assign each system net rate change to a specific color
plotColor = ColorItem[ systemIndex ]

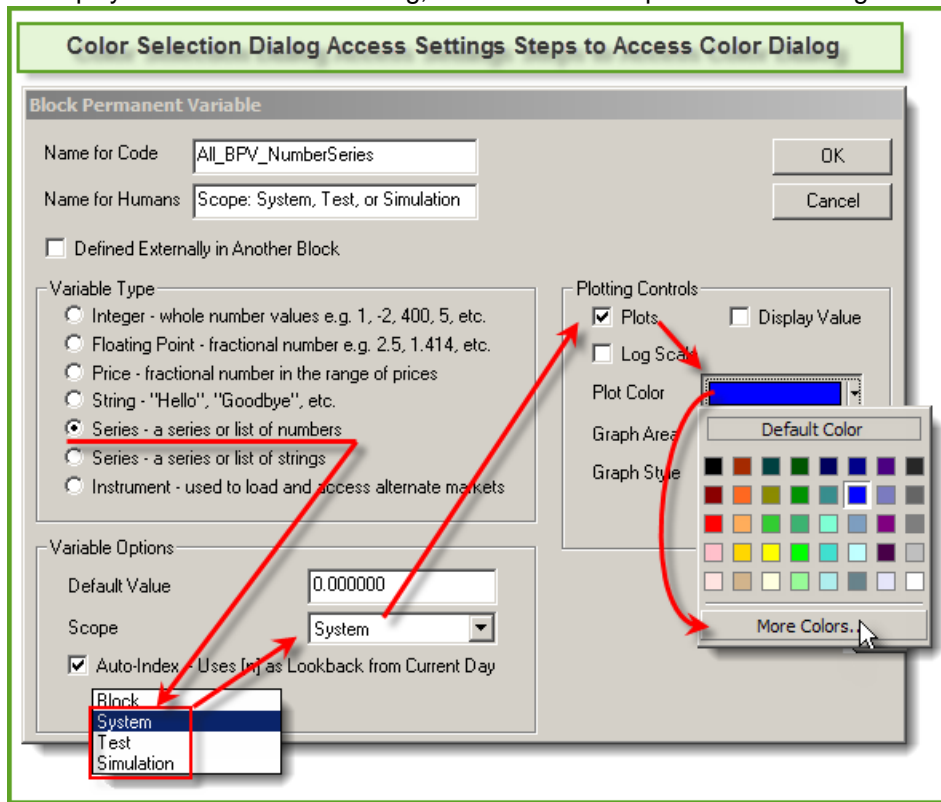
' Place this system's test-date total equity percentage net chang
' value in the chart space using the new color
chart.AddLineSeries( AsSeries(systemEquity), elementCount, _
                    alternateSystem.name, plotColor )

```

Trading Blox Color Selection Dialog:

All BPV Series will provide access to the Trading Blox Color Selection Dialog when the BPV series uses a System, Test or Simulation Scope setting with a BPV numeric series.

To display the color selection dialog, follow the click steps in this next image:



When the "More Colors..." button is clicked the dialog in this next image will appear:

Pick a color from color table, or select a color from the rainbow

This slider Arrow adjust color shade

These RGB values can be used in ColorRGB function

```

ColorValue = ColorRGB( 255, 0, 0 ) ' Blue = 255
ColorValue = ColorRGB( 255, 0, 255 ) ' Magenta = 16711935
ColorValue = ColorRGB( 255, 0, 255 ) ' Red = 16711680
ColorValue = ColorRGB( 255, 0, 255 ) ' Green = 65280
ColorValue = ColorRGB( 0, 0, 0 ) ' Black = 0

```

Just about any color's RGB value can be discovered using this dialog. However, if the chart image where this color is to be used will appear in a report generated with a HTML Browser process that is used to create Trading Blox reports, picking a color from the Basic Color Matrix Table will keep the colors used within the Safe-Color range that are easily reproduced using a HTML process.

Applying the RGB, (Red, Green, Blue) values to Trading Blox's ColorRGB function, place the color numbers using Blue, Green and Red as the first, second and third parameter locations

Script Color Assignment Examples:

```

'           Blue   Green   Red
PlotColor1 = ColorRGB( 255, 0, 0 ) ' Plot Blue Color
PlotColor2 = ColorRGB( 0, 255, 0 ) ' Plot Green Color
PlotColor3 = ColorRGB( 0, 0, 255 ) ' Plot Red Color

' Trade Color Preference Settings Color Numbers values
PlotColor1 = ColorCustom1 ' Use Preference ColorCustom1 Value
PlotColor2 = ColorCustom2 ' Use Preference ColorCustom2 Value
PlotColor3 = ColorCustom3 ' Use Preference ColorCustom3 Value

```

Free Color RGB Identification Software:
[ColorPic - Free Download](#) (External Web Site Link)

Link will open the default browser to the web page where this free program **ColorPic** software will provide the color number of any pixel displayed on computer screen.

Links:

[AddLineSeries](#), [ColorRGB](#), [SetSeriesColorStyle](#), [Preference Items](#)

Section 3 – Constants Reference

Trading Blox Builder contains several built-in constants that can be used in scripts:

Constant Name:	Constant Value:
PI	3.141592653589 79321
TRUE	1
FALSE	0
LONG	1
SHORT	-1
OUT	0
SUNDAY	0
MONDAY	1
TUESDAY	2
WEDNESDAY	3
THURSDAY	4
FRIDAY	5
SATURDAY	6
UNDEFINED	n/a

Using constants help to make your code easier to understand.

The **UNDEFINED** can be assigned to plotting series, so that the particular series value does not plot.

Contrast this code:

```
IF instrument.position = 1 THEN
    ' Do some Long stuff here.
ENDIF
```

with this code:

```
IF instrument.position = LONG THEN
    ' Do some Long stuff here.
ENDIF
```

In the following code, is it obvious what day we are referring to:

```
IF DayOfWeek( instrument.date ) = 1 THEN
    ' Do our weekly tasks here.
ENDIF
```

How about in this code:

```
IF DayOfWeek( instrument.date ) = MONDAY THEN
    ' Do our weekly tasks here.
ENDIF
```

Section 4 – Data Groups and Types

Trading Blox Basic supports four variable Groups:

Groups:	Use:
Block Permanent (BPV)	<p>Block Permanent Variables are common to all the script section, and are accessible anywhere in the block module when the variables Scope is set to Block. They can be accessible anywhere in the System, Test, or Simulation if one of those wider Scope ranges is selected.</p> <p>Various BPV Variable Types can be selected (see table below). None except the Instrument Type is connected to a specific instrument as Instrument Permanent variables are designed. Instead, they can accept a value from an instrument, but the value will be accessible to all the other instruments as the Instrument loop processes each symbol in a portfolio.</p> <p>When deciding on which kind of variable to use, consider if the value in the variable can be common to all instruments. If it can, the a BPV will be a good choice. If it needs to be specific to a specific instrument, then select an IPV.</p>
Instrument Permanent (IPV)	<p>Instrument Permanent variables are designed to add information to the property information available to each portfolio symbol. In a sense they create an additional property that can easily be assigned and accessed through simple scripting statements.</p> <p>IPV group supports a variety of variable types (see table below), and they follow the same rules instruments observe in the various scripts were they are given context automatically, or are required to be brought into context through a manual process.</p> <p>Select an IPV as a group choice when the information to be contained within the variable will only pertain to that specific instrument. For example, if you want to set a value for a signal for that instrument alone, then an IPV is a good choice. However, if the value in a variable pertains to all of the instruments, then a BPV is the better choice.</p>
Parameters	<p>Parameter variables become visible in the user's menu area. They provide methods for changing the length of a calculation, the value to be applied in a calculation, controls that can influence which areas of a module are allowed to function, and they can be stepped in an attempt to discover how their variation in value influences how a system performs.</p> <p>Parameters are accessible in all script sections.</p>
Local	<p>Local variables are created in a specific script section. Their scope is isolated to the script section in which they are declared, and are accessible anywhere in that same script section.</p> <p>When naming a Local variable ensure the same name being declared has not been used elsewhere in the script, or in the System, Test, or Simulation when those scope are employed in the Suite of modules.</p> <p>In simple terms, only use local variable names that are unique and are only needed in that script section.</p>

Local variables are not cleared when the script section is executed. This means to clear the value in a Local variable that can affect a statement before the script section has assigned it a value, the variable in most cases should be cleared.

Within these groups there are six standard variable types. One of which is a special Instrument type that can be created as a BPV and used to access an instrument. Not all of the variable types are available in all of the groups. To know which is available where, review the table here and review the links associated with of the variable types and groups.

Variables available by Group and Type:

TYPE	Available:	Description:
Boolean	P	This parameter variable type can only be True or False. When True the value is equal to 1, and it is zero when False.
Floating	B, I, P	Decimal numbers like 1.24 or 3.14159 which are not whole numbers
Instrument - BPV	B	Direct access to instruments is only available in the scripts that allow the instrument object context. To access an instrument in a script where it doesn't have context it is necessary to use this BPV type so an instrument can be moved into context and made accessible.
Integer	B, I, P	Whole numbers like 1, 200, 582, -5
Money	S	Variables which hold money. Internally Money variables are stored in the same way as a floating point variables. Money variables are printed differently and show in the debugger with different formatting.
Percent	P	Default value entered must be as a decimal where 0.10. When this parameter appears on the main screen page it will be displayed as 10%. In calculations it will be applied as a decimal where 10% is applied as 0.10.
Price	B, I	Variables which hold price information. Internally Price variables are stored in the same way as a floating point variables. Price variables are printed according the current instrument's formatting and show in the debugger using that format. Price variables are also unadjusted for any negative value adjustment that may be present because of a negative price series in the instrument's data.
Selector	P	This is a special type of parameter because it can be assigned a list of words that will act as value to allow the user to modify how the scripts operate. When the list is created, the first word phrase entered will be assigned the value of zero. Second word phrase will be assigned the value of 2, and each remaining word phrase will be assigned the next available integer value until all have been assigned a value. In operation, the use will select one of the option displayed in a drop-down list of word phrases, or they can select the Step All option enabled when it is enabled.
Series	B, I	All series or arrays store numbers and text value in a series elements. Each element contains the value it was assigned until it is cleared. Each element in a series is accessible, or index, by using its referenced location. In simple terms, a series is a list or an array of elements that contain the values of Floating numbers or String Alpha-Numeric characters.

		<p>Auto-Indexed series are accessed are reference by the specifying the offset value from the current bar to where that element will be located. For example, to access yesterday's series element, use a value of 1 as the offset amount.</p> <p>Manually sized series are access an element by its count position within the series.</p> <p>Smallest value of a series is 1, and 1 is always the position location of first element in the series. Use above information to determine if it is accessed by offset reference, or direct location reference.</p>
String	B, I, P	Characters, or combinations of characters like "Hello", "A", and "November Soybeans."
<p>Legend:</p> <p>B = BPV I = IPV P = Parameter S = Script</p> <p>Note: All Parameters, except String Parameters, can be assigned a default value. All parameters, except String types, can step their values, and can be assigned a stepping priority value to arrange their sequence in how they are to stepped during a test simulation. Integer Parameters can be enable during program data priming Look-back amount calculations to prevent out of range errors..</p>		

Notes:

Variables declared using the **VARIABLES** statement in any of the script section are local to that script except when used in a Custom Script Function. Local means that script defined variables are only accessible during the script in which they are defined. This also means you cannot define a variable in an Entry script and then believe you can use it in the Exit script. When you need cross script access to common variables use an IPV, or a BPV declared variable.

When script declared variables are used in a Custom Script Function, the values in the Custom Script Function will be exposed to the values in the script that called the Custom Script Function. This ability makes Custom Script Function flexible to use and to gain access to various script section, but it can cause errors when the name of a script declared variable is the same as a variable in the script that is calling the Custom Script Function.

Declaring Local variables can be handle in ways similar to what is shown in the examples that follow:

Syntax:

```
VARIABLES: varname1 [ TYPE: type ], varname2 [ TYPE: type ] ...
```

varname Name of the variable.

type **Type** of the variable - see table below.

Various examples to illustrate how to use a **VARIABLES statement:**

```
VARIABLES: someValue TYPE: Integer
```

```
' SomeValue was defined and can be only integer
```

```
someValue = 10            ' SomeValue is integer and contains 10
```

```
someValue = 3.15        ' SomeValue is integer and contains 3
```

```
VARIABLES: a TYPE: Floating
' Single VARIABLES statement
```

```
VARIABLES: a, b, c TYPE: Integer
' Three variables in single VARIABLE statement
```

```
VARIABLES: str1 TYPE: String, int1 TYPE: Integer
' Multiple variables of different types
```

The variable `someValue` should be declared with a TYPE of a Price when value displays need price formatting:

```
someValue = instrument.close
someValue = instrument.high - averageTrueRange
someValue = instrument.low * 1.2
someValue = longMovingAverage (where longMovingAverage is a moving average)
```

The variable `someValue` should be a Floating TYPE in the following situations:

```
someValue = instrument.close - instrument.close[1]
someValue = instrument.high - instrument.low
someValue = averageTrueRange
```

4.1 Boolean

TYPE:	Description:
Boolean	This parameter variable type can only be True or False. When True the value is equal to 1, and it is zero when False.

Notes:

A Boolean variable is restricted to a binary state. This means it can only be one of two values. In numerical terms it is either a 1 or a 0. Two Trading Blox constants are available as variable terms to denote each of the binary state values allowed a Boolean.

Only a parameter can be declare as a Boolean type variable, but an Integer is often used to denote a True or False condition, so an Integer can be used as a scripting value to accept the value of a Boolean parameter, or provide a binary state process.

When a parameter is declared as a Boolean, the user will see a parameter on the Main screen displayed with selection list option that allows the user to select either True or False. If the user checked the "Stepping Enabled" option for the Boolean parameter, the Boolean parameter variable will still only have two options, but Trading Blox display a third option "Step True to False" that will enable Trading Blox to automatically step the value between True and False during a test.

Dialog Example:

Boolean Parameter Dialog Example

Parameter Menu Example:

Boolean Parameter Menu Selection Example

Script Example:

```

' See Dialog Parameter Name
If TrueOrFalse = True Then
  ' When True, Do Stuff Here
Else
  ' When False, Do Stuff Here
EndIf

```

Links:

[Constants Reference](#)

See Also:

[Data Group and Types](#)

4.2 Floating

TYPE:	Description:
Floating	Decimal numbers like 1.24 or 3.14159 which are not whole numbers
<p>Example:</p> <pre>' Floating TYPE example VARIABLES: NumberIs TYPE: Floating ' Calculate two numbers NumberIs = 3.364 * 1.108 ' Send number results to Log Window PRINT "NumberIs = ", NumberIs</pre>	
<p>Returns:</p> <pre>' Log window shows NumberIs = 3.727312000</pre>	
<p>Links:</p> <p>PRINT</p> <p>See Also:</p> <p>Data Group and Types</p>	

4.3 Instrument - BPV

TYPE:	Description:
Instrument - BPV	Direct access to instruments is only available in the scripts that allow the instrument object context. To access an instrument in a script where it doesn't have context it is necessary to use this BPV type so an instrument can be moved into context and made accessible.
<p>Notes:</p> <p>Data instrument access to its properties and functions is automatic when the instrument has context, and when the symbol is active in a script where instruments have context. Context means the information is automatically made available when that script section is executing and the portfolio looping brings that instrument into the script.</p> <p>When other script sections are executing, Trading Blox doesn't loop instruments, but instead will on execute scripts without instrument context once for each bar being tested. To access an instrument the script will need to bring the instrument into context in order to gain access to its properties and functions. To bring an instrument into context, use a BPV Type Instrument variable as the structure where the instruments data can be copied so it can be accessed.</p> <p>This table is a current list of the script section that automatically bring instruments into context automatically, and the scripts sections where scripting code will be required to gain access to an</p>	

instrument:

Script Instrument Access	
Automatic Access:	Manual Access:
Rank Instruments	Before Simulation
Filter Portfolio	Before Test
Before Instrument Day	Before Trading Day
Exit Orders	Before Bar
Entry Orders	Before Order
Unit Size	Execution
Can Add Unit	After Bar
After Instrument Open	Initialize Risk
Update Indicators	Management
Can Fill Order	Compute Risk
Entry Order Filled	Adjustments
Can Fill Order	After Trading Day
Exit Order Filled	After Test
Adjust Stops	After Simulation
Adjust Instrument Risk	
After Instrument Day	

When access to instrument information isn't automatic it can be loaded into the BPV Instrument Variable Type using the `LoadSymbol` function. This next example shows how all of the instruments in the portfolio can be loaded and accessed:

Example:

```

' ~~~~~
' Example of how to bring instruments into context
' ~~~~~
' When Portfolio has selected instruments,...
If System.TotalInstruments > 0 Then
  ' Loop through all the instruments in the portfolio
  For x = 1 to System.TotalInstruments Step 1
    ' Load instrument at portfolio's index position of ' x '
    ' and place the information into the BPV instrument ' Mkt '
    ' When function executes, it will assign a 1 if the
    ' instrument access is successful, or a 0 when access fails
    iLoadOK = Mkt.LoadSymbol( x )

    ' if iLoadOK is TRUE ( greater than 1 )
    If iLoadOK Then
      ' Display the instruments symbol in the Log Window
      Print "Mkt.Symbol      ", Mkt.Symbol
    EndIf
  Next ' x
EndIf ' System.TotalInstruments > 0

```

Returns:

Mkt.Symbol CL2

In the above example a generic name of 'Mkt' was assigned to the BPV instrument. When an instrument access needs to be available through out a simulation, a unique name can be given to a

BPV instrument so it cancelled using that unique name and thus be available without having to bring it into context with a loading script. Consider this simple Grain market example:

Example:

```
' ~~~~~
' Load three grain instruments into grain named BPV Instruments.
' ~~~~~
' Load Corn into BPV Instrument Corn
If Corn.LoadSymbol( "C2" ) = TRUE THEN Print "Corn Market Loaded"

' Load Soybeans into BPV Instrument Soybeans
If Soybeans.LoadSymbol( "S2" ) = TRUE THEN Print "Soybean Market Loaded"

' Load Wheat into BPV Instrument Wheat
If Wheat.LoadSymbol( "W" ) = TRUE THEN Print "Wheat Market Loaded"
```

Returns:

```
Corn Market Loaded
Soybean Market Loaded
Wheat Market Loaded
```

In the above example each of the grain markets can be called in any of the script sections without having to load them where they are not normally in the context of that script. Calling them in any of the scripts would look something like this:

Example:

```
' Script line using BPV Instrument Corn placed in each of
' Blox script sections so that each could report what it
' found accessible.
Print Block.ScriptName, Corn.Symbol, Corn.Date, Corn.Close
```

Returns:

```
' Details shown are text copies of some of the
' data records generated by the above when it
' was placed in each of the script section in
' the blox
Before Test C2 2007-12-27 508.250000000
Update Indicators C2 2007-12-27 508.250000000
Before Trading Day C2 2007-12-27 508.250000000
Before Instrument Day C2 2007-12-27 508.250000000
Update Indicators C2 2007-12-28 505.500000000
After Instrument Day C2 2007-12-28 505.500000000
After Trading Day C2 2007-12-28 505.500000000
Before Trading Day C2 2007-12-28 505.500000000
Update Indicators C2 2007-12-31 509.000000000
After Instrument Day C2 2007-12-31 509.000000000
After Trading Day C2 2007-12-31 509.000000000
After Test C2 2007-12-31 509.000000000
After Simulation C2 2007-12-31 509.000000000
```

Links:

[LoadSymbol](#), [Print](#), [TotalInstruments](#)

See Also:

[Data Group and Types](#), [Block](#)

4.4 Integer

TYPE:	Description:
Integer	Whole numbers like 1, 200, 582, -5

Example:

```
' Integer TYPE example
VARIABLES: NumberIs TYPE: Integer

' Calculate two numbers
NumberIs = 3.364 * 1.108

' Send number results to Log Window
PRINT "NumberIs = ", NumberIs
```

Returns:

```
' Log window shows
NumberIs = 3
```

Links:

[PRINT](#)

See Also:

[Data Group and Types](#)

4.5 Money

TYPE:	Description:
Money	Variables which hold money. Internally Money variables are stored in the same way as a floating point variables. Money variables are printed differently and show in the debugger with different formatting.

Example:


```
' Money TYPE example
VARIABLES: NumberIs TYPE: Money

' Calculate two numbers
NumberIs = 3.364 * 1.108

' Send number results to Log Window
PRINT "NumberIs = ", NumberIs
```

Returns:

```
' Log window shows
NumberIs = 3.727312000
```

Links:**See Also:**

[Data Group and Types](#)

4.6 Percent

TYPE:	Description:
Percent	Default value entered must be as a decimal where 0.10. When this parameter appears on the main screen page it will be displayed as 10%. In calculations it will be applied as a decimal where 10% is applied as 0.10.

Notes:

Percentage Screen display values show the default value entered into the parameter dialog's Default Value field, unless the Blox is attached to a Suite and the user has changed the Main screen displayed value. When entering a value into the dialog's Default Value field use the decimal equivalent for the percentage display required.

When changing the value of the Main screen's displayed percentage display, a percentage number should be used. For example, if you enter a 20 into the Main screen parameter display, it will display 20%, and it will be equivalent to 0.20 decimal when applied as a script value. If you enter a value of 0.20 it will be displayed as 0.20% and be equivalent to 0.02 decimal in a code statement.

Dialog Example:

Edit Parameter

Name for Code:

Name for Humans:

Parameter Type:

- Integer - whole number values e.g. 1, 400, 5, etc.
- Floating Point - fractional numeric values e.g. 1.25, 2.5, etc.
- Percent - fractional numeric percentage e.g. 1.5%, 10%, etc.
- Boolean - values that are either TRUE or FALSE
- String - text e.g. "hello"
- Selector - values that are selected from a list of values

Selector Entries:

Entry	Basic Constant

Default Value:

Scope:

Used for Lookback Stepping Enabled

Stepping Priority:

Enter Percentage's Decimal value. Example: 0.10 for 10%

Parameter Stepping Enabled

Parameter Menu Example:

Auxiliary

Percent Parameters Display: Step

Default Value:

Scope:

Used for Lookback Stepping Enabled

Entry Value Shows

Entry Value Shows

Example:

Variables: AmountIs, NumberIs **Type:** Floating

' Assign a value

AmountIs = 1000

' Using the Dialog variable name above,...

NumberIs = AmountIs * PercentValue

' Send number results to Log Window

PRINT "NumberIs = ", NumberIs

Returns:

NumberIs = 100.00000000

Links:[Constants Reference](#)**See Also:**[Data Group and Types](#)

4.7 Price

TYPE:	Description:
Price	Variables which hold price information. Internally Price variables are stored in the same way as a floating point variables. Price variables are printed according the current instrument's formatting and show in the debugger using that format. Price variables are also unadjusted for any negative value adjustment that may be present because of a negative price series in the instrument's data.
Example:	
<pre>' Price TYPE example VARIABLES: NumberIs TYPE: Price ' Calculate two numbers NumberIs = 3.364 * 1.108 ' Send number results to Log Window PRINT "NumberIs = ", NumberIs</pre>	
Returns:	
<pre>' Log window shows NumberIs = 3.727312000</pre>	
Links:	
PRINT	
See Also:	
Data Group and Types	

4.8 Selector

TYPE:	Description:
Selector	This is a special type of parameter because it can be assigned a list of words that will act as value to allow the user to modify how the scripts operate. When the list is created, the first word phrase entered will be assigned the value of zero. Second word phrase will be assigned the value of 2, and each remaining word phrase will be assigned the next available

integer value until all have been assigned a value.

In operation, the user will select one of the option displayed in a drop-down list of word phrases, or they can select the Step All option enabled when it is enabled.

Notes:

Selector parameter provides multiple options for controlling various sections in a Blox to be operational, or disabled.

Each option text is created by the user and each of the options are available in the parameter section of the Main screen. Text can be any name that begins with a character, but that option text-name can only exist once in the Blox, but it can exist in other Blox as long the scope is set to Blox.

Dialog Example:

New Parameter

Name for Code: SelectedItem

Name for Humans: Select A List Option:

Parameter Type:

- Integer - whole number values e.g. 1, 400, 5, etc.
- Floating Point - fractional numeric values e.g. 1.25, 2.5, etc.
- Percent - fractional numeric percentage e.g. 1.5%, 10%, etc.
- Boolean - values that are either TRUE or FALSE
- String - text e.g. "hello"
- Selector - values that are selected from a list of values

Selector Entries:

Entry	Basic Constant
Option 1	OPTION_1
Option 2	OPTION_2
Option 3	OPTION_3

Default Value: Option 1

Scope: Block

Used for Lookback Stepping Enabled

Stepping Priority: 0

Buttons: Add Entry, Delete Entry, Move Entry Up, Move Entry Down

Parameter Menu Example:

Auxiliary

Select A List Option: Option 1

Option 1
Option 2
Option 3
Step All Values

Example:

```

' Condition Statement Follows Menu's Selected option
If SelectedItem = OPTION_1 Then
  ' Do Option_1 Stuff...
Else
  If SelectedItem = OPTION_2 Then
    ' Do Option_2 Stuff...
  Else
    If SelectedItem = OPTION_3 Then
      ' Do Option_3 Stuff...
    EndIf
  EndIf
EndIf

```

Links:**See Also:**

[Data Group and Types](#)

4.9 Series

BPV and **IPV** variables support numeric and text based series arrays.

Series can be declared to Auto-Index or Manually index, and the location to access and assign information is determined by the user selected option.

Index Method	Series Type	Description:
Auto-Index (Default)	BPV	Auto-Index option controls the size of the series, and increments each element to align with the test position. Series element position index value, and the number of elements in the series is the value of the <code>test.currentDay</code> property. BPV Series values can be plotted and displayed in a Custom Chart when the BPV series is set to a Scope of System, Test, or simulation. When the scope is not set to Block, the Plotting section of the dialog will appear where the graphing options can be accessed.
	IPV	Auto-Index option controls the size of the series, and increments each element to align with the instrument's test position. Series element position index value, and the number of elements in the series is the value returned by <code>instrument.bar</code> property of each instrument. Auto-Index series Plot and Display options allow the data from the series to be displayed on the price chart where the instrument's price information is displayed.
Manual-Index	BPV & IPV	Manual indexing series size can be determined when the series is created, and they can be changed in the script with the <code>SetSeriesSize</code> series function.

	<p>Minimum series size and index value is 1, and Maximum series size and index value is the size, or the number of elements in the series. Index size can be retrieved using the GetSeriesSize series function.</p> <p>Script index values must always be in the range starting at 1 and can be up to size of the series.</p> <p>Manual series can assign the same value to all the elements in the series at the same time using SetSeriesValues series function.</p> <p>Manual series can be sorted in an ascending or descending direction using the SortSeries series function. When sorting String values in a series require the understanding of how character values are ordered when sorted.</p>
BPV	<p>Numeric BPV series can also be declared in the script window:</p> <pre>Variables: < series-name > TYPE: Series</pre> <p>When using script to declaring a series it must be sized using the SetSeriesSize series function before it can be index for value assignment.</p>
IPV	
	<p>GetSeriesSize SetSeriesSize SetSeriesValues SortSeries SortSeriesDual</p>
<p>Links:</p> <p>instrument.bar, test.currentDay GetSeriesSize, SetSeriesSize, SetSeriesValues, SortSeries, SortSeriesDual</p> <p>See Also:</p> <p>Data Group and Types, Instrument Data Access Properties, Test Object General Properties</p>	

Numeric Series

TYPE:	Description:
Number Series	All series, or arrays store numbers and text values in a series of individual variable elements. Each element stores the value it was assigned until it is cleared, or given a different value. Each element in a series is accessible by using its referenced location. Usually this referencing is to assign the location to an integer value that acts as a location reference, or an index to that elements location.

In simple terms a series is a list of elements that contain the values of Floating numbers in numeric arrays, or a series of String Alpha-Numeric characters in the elements in a String array.

Auto-Indexed series are accessed are reference by the specifying the offset value from the current bar to where that element will be located. For example, to access yesterday's series element, use a value of 1 as the offset amount.

Manually sized series are access an element by its count position within the series.

Smallest value of a series is 1, and 1 is always the position location of first element in the series. Use above information to determine if it is accessed by offset reference, or direct location reference.

Dialog Example:

Series data types can be Auto-Indexed, or they can be manually indexed. If there needs to be a series element for a named series to match the count of an instrument's data records, then use the default setting of Auto-Index.

A series that doesn't need to align to the instrument's data records, but might contain less or more information, use the manual index option and control the indexing and the sizing of the series with your program's source code.

Links:

See Also:

[Data Group and Types](#)

String Series

TYPE:	Description:
String Series	<p>All series, or arrays store numbers and text values in a series of individual variable elements. Each element stores the value it was assigned until it is cleared, or given a different value. Each element in a series is accessible by using its referenced location. Usually this referencing is to assign the location to an integer value that acts as a location reference, or an index to that elements location.</p> <p>In simple terms a series is a list of elements that contain the values of Floating numbers in numeric arrays, or a series of String Alpha-Numeric characters in the elements in a String array.</p> <p>Auto-Indexed series are accessed are reference by the specifying the offset value from the current bar to where that element will be located. For example, to access yesterday's series element, use a value of 1 as the offset amount.</p> <p>Manually sized series are access an element by its count position within the series.</p> <p>Smallest value of a series is 1, and 1 is always the position location of first element in the series. Use above information to determine if it is accessed by offset reference, or direct location reference.</p>

Dialog Example:

Series data types can be Auto-Indexed, or they can be manually indexed. If there needs to be a series element for a named series to match the count of an instrument's data records, then use the default setting of Auto-Index.

A series that doesn't need to align to the instrument's data records, but might contain less or more information, use the manual index option and control the indexing and the sizing of the series with your program's source code.

Links:

See Also:

[Data Group and Types](#)

4.10 String

String variables, which have a maximum character length of 512 are used to store text characters so they can be accessed later in the module's operation.

TYPE:	Description:
String	Any keyboard character, or combinations of characters like "Hello", "A", and "November Soybeans." A String character, or word can be assigned to a String variable, or passed to a function that requires a String assignment. When passed to a variable, or to a parameter position the character group must be contained within a pair of apostrophe characters:

Example:

```
' String TYPE Example
VARIABLES: TextItemIs TYPE: String

' Assign text to string variable
TextItemIs = "November Soybeans."

' Send String results to Log Window
PRINT "TextItemIs = ", TextItemIs
```

Returns:

```
' Log window shows
TextItemIs = November Soybeans.
```

Links:

[PRINT](#)

See Also:

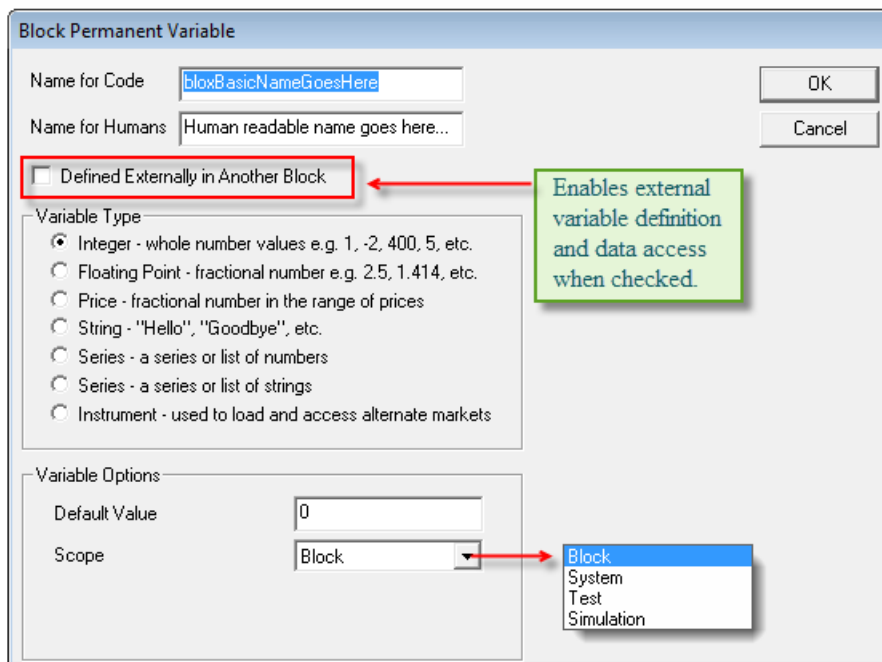
[Data Group and Types](#)



Section 5 – Data Scope Reference

Variables, indicators and parameters can have different scopes. Scope in programming determines the range, or reach from which a data item can be referenced. In Trading Blox different types of variables can have different levels of scope. Scope is established by selecting the various scope references provided in the Scope's drop-down list, which is always set to Block unless the programmer changes the scope assignment. When scope is set to Block, that is most narrow or limited reach that the data in that data element is allowed. Local variables, those declared using the keywords Variables & Type entered into the script section code space, and not used in a custom function, are scoped the script section in which they are declared. Local variables declared and scoped within a custom function can also be accessible to the variables in the script section that called the custom function.

Setting the variable scope to something other than to **Block** scope allows the variable to be used in other blox in the system. However, the other blox needs to have a variable with the same name, and it should be defined as external.



Only **IPV** and **BPV** can be defined as **External**.

IPV -- Instrument Permanent Variables:

Block	You can only use this variable in the scripts that are in the block.
System	You can use this variable in any block in the System by declaring the variable as External in the other blocks.
Simulation	Same as System scoped except the value is not reset for every test (parameter run).

Notes:

To use a System or Simulation scoped IPV in another block, define an external IPV of the same name in the other block.

BPV -- Block Permanent Variables:

Block	You can only use this variable in the scripts that are in the block.
System	You can use this variable in any block in the System.
Test	You can use this variable in any block in the Test.
Simulation	Same as Test scoped except the value is not reset for every test (parameter run).
Notes: To use a System, Test, or Simulation scoped BPV in another block, define an external BPV of the same name in the other block.	

Parameters:	
Block	You can only use this parameter in the scripts that are in the block.
System	You can use this parameter in any block in the System
Simulation	You can use this parameter in any block in the Test.

Indicators:	
Block	You can only use this indicator in the scripts that are in the block.
System	You can use this indicator in any block in the System.
Notes: To use a System scoped Indicator in another block, define an external IPV Series of the same name in the other block.	

All variables, parameters and indicators are reset between every test run during a multi-stepped parameter test, and it happens just before the **Before Test** script. An exception to this data refreshing is allowed when the variable is set to the Simulation scope reference.

IPV and **BPV** variables are reset to their default value, the parameters are set to the next stepped value, and all indicators are recomputed.

Setting **IPV** or **BPV** variables using **Simulation** scope will prevent the values from being reset to the default value. This can be useful when keeping track of a value over the course of many stepped tests, or when loading external data in the **Before Simulation** script.

Because all values are reset just before the **Before Test** script, the **Before Simulation** script has no access to indicators or parameters, as they are not set yet. Any **IPV** or **BPV** that are set or loaded in the **Before Simulation** script will be reset to their default value unless they are defined as **Simulation** scope.

Section 6 – Data Script Comments

Comments are sections of code that are intended for human reading, and are ignored by Blox Basic script interpreter.

Comments help you document as you are creating it, and provide an aid in understanding the code later on. Comments are also useful for others to understand your intentions when you created your code.

Blox Basic comments start with the `'` character and continues until the end of the line. The Trading Blox Builder interpreter ignores any characters or code that follows a `'` character.

Example:

```
' ~~~~~  
' The Amazing Code to Add Two Variables Together!!!  
' Copyright 2005 By TradingBlox LLC. All Rights Reserved.  
' Steal this code at your own peril!  
' ~~~~~  
  
' Add a and b  
SumAB = ( a + b )  
  
' That was great!
```

Section 7 – Data Variable Names

Variable names must begin with a letter and must be no longer than 64 characters and can only contain the following characters:

- `_` The underscore character
- A-Z or a-z Any letters
- 0-9 Any combination of digits

The following are all valid variable names:

```
aVariable
aVariable2000
a_variable
a_variable_2000
```

Variable names that begin with a numeric characters are **NOT** valid:

```
2000variable
12_variable
```

This list of words should not be used as variable names, as they are reserved by the program for specific purposes.

All words in this list are **case insensitive** - neither the variable "UNITSIZE" nor "unitsize" can be used..

abs	long
AbsoluteValue	loop
acos	lowerCase
and	ltrim
ArcCosine	max
ArcSine	mid
ArcTangent	middleCharacters
ArcTangentXY	min
asc	mod
ascii	monthNumber
asciiToCharacters	newPosition
asFloating	next
asin	not
asInteger	or
asString	orde
atan	r
atan2	out
beep	pi
block	print
broker	RadiansToDegrees
CanAddUnit	random
chr	right
clearscreen	rightCharacters
cos	rtrim
cosine	short
dayOfMonth	sin
DegreesToRadians	sine
degtorad	sqr
do	sqr

else	squareRoot
endfunction	step
endif	stringLength
endsub	sub
endwhile	system
entryPrice	tan
exp	tangent
exponent	test
false	then
findString	to
findString	toJulian
for	totalRisk
function	trim
goto	trimLeftSpaces
hypot	trimRightCharacters
hypot	trimRightSpaces
hypotenuse	trimSpaces
if	true
instrument	type
isFloating	type
isInteger	ucase
isString	unitSize
lcase	until
left	upperCase
leftCharacters	variables
len	variables
log	while
log10	xor

Section 8 – Data Variables

What is a variable?

Variables are simply a name which represents a value or series of values. If you have used a spreadsheet then you have used variables. For example, in a spreadsheet column B row 4 might be the total sales for the month. In Excel you could name this cell to something like "monthlySales" then in other cells you could refer to that variable (cell) as either B4 or "monthlySales".

In Blox Basic you can create variables which have names and can hold values just like a spreadsheet cell can.

Why would you do this? Suppose you want to compute the stop price for a buy and it will be 3 ATR less than the long moving average. You could use the expression:

```
longMovingAverage - (3 * averageTrueRange)
```

everywhere you needed to use the stop like:

```
broker.EnterLongOnOpen( longMovingAverage - (3 * averageTrueRange) )
```

Or you could simply define a variable and set its value using that expression:

```
VARIABLES: stopPrice TYPE: Price

' Compute the stop.
stopPrice = longMovingAverage - (3 * averageTrueRange)

' Enter the order to buy
broker.EnterLongOnOpen( stopPrice )
```

The method provides several benefits. First, it is easier to understand. Second, when you read the code later, you can easily see where the stop price is being calculated.

Variables are essential for dealing with user input, calculations, simplifying code, and output in a program. For instance, to print out the multiplication tables up to 10 x 10 manually:

```
PRINT "1 x 1 = 1"
PRINT "1 x 2 = 2"
PRINT "1 x 3 = 3"
PRINT "1 x 4 = 4"
```

It would take a long time! Using variables and a simple program you can do this more quickly:

```
columnOne = 1
columnTwo = 1

DO
    PRINT columnOne, " x ", columnTwo, " = ", columnOne * columnTwo

    columnTwo = columnTwo + 1

    IF columnTwo = 11 THEN
        columnOne = columnOne + 1
        columnTwo = 1
    ENDIF

LOOP UNTIL columnOne = 11
```


As you can see using variables can make life easier for repetitive tasks.

How do I make a variable?

Trading Blox Builder provides several ways of "declaring" variables.

- 1) Use the [VARIABLES](#) statement
- 2) Create an [Instrument Permanent Variable](#)
- 3) Create a [Block Permanent Variable](#)

How do I know which kind of variable to create?

Each of these types of variables have different lifetimes and qualities:

- 1) Variables declared with the VARIABLES statement only maintain their value in and during the script in which they are declared. They are undefined at the start of the script. These variables are *temporary* and *not permanent*. So you cannot assume the variable will hold its value from one instrument to the next, or one day to the next.
- 2) Variables you create as an Instrument Permanent Variable retain their value from one script to the next but their value is specific to the current instrument. In other words, each instrument has its own copy of the variable. For instance, an Instrument Permanent Variable named `channelWidth` could be 20 for Gold and 10 for Corn. Instrument Permanent Variables get their name because they are instrument-specific and permanent (i.e. they maintain their value across script invocations).
- 3) Block Permanent Variables are not instrument-specific, there is only one variable for the entire block. For example, if you increment a Block Permanent Variable named `totalUnits` each time a new position is added and you put on three units in three different instruments/markets the value of `totalUnits` will be 3 in every script that accesses it.

Variable Data Types:

When you use one of these methods to declare a variable, you must choose a "type". The variable's type determines what kind of information that variable can contain. A variable can be one of the following.

Variable Type Names:	Descriptions:
Floating	numbers like 1.24 or 3.14159 which are not whole numbers
Instrument	An instrument which can be accessed like the global 'instrument' trading object.
Integer	whole numbers like 1, 200, 582, -5
Money	Variables which hold money. Internally Money variables are stored in the same way as a floating point variables. Money variables are printed differently and show in the debugger with different formatting.
Price	Variables which hold price information. Internally Price variables are stored in the same way as a floating point variables. Price variables are printed according the current instrument's formatting and show in the debugger using that format. Price variables are also unadjusted for any negative value adjustment that

	may be present because of a negative price series in the instrument's data.
<u>Series</u>	A list, or an array of Floating numbers or characters (only Block Permanent and Instrument Permanent variables support number and character series).
String Series	A list or array of strings.
String	characters or combinations of characters like "Hello", "A", and "November Soybeans"

After a variable is declared, it cannot change its type. If you make it a string it must stay a string. Values of one type can be converted to another type using the conversion functions.

Links:

[Type Conversion Functions](#)

Section 9 – FunctionReference

Trading Blox Builder includes many built-in functions which can be used in scripts. These functions are similar to the functions used in spreadsheet formulas. Trading Blox Builder includes the following function types:

Function Section:	Function Type:
Date	date functions
File	file functions
General	general functions
Mathematical	math functions
String	string functions
Type Conversion	type conversion functions

Each function section contains a table showing the names of the functions topics in that section.

9.1 Custom Functions

Please See: [Script](#) Object

Custom User Functions

Creating & Calling User Functions

User Created Functions expand the extensibility of the Trading Blox Basic language and facilitate a user's ability to create a unique functions or methods. This is made possible by using Trading Blox's Script Object that are explained later in this page.

All user functions can have any combination of numeric ([Integer](#), or [Floating](#)), [String](#), series, or no parameters. They can return a numeric, string, series, or no value. They work just like inherent Trading Blox Functions, but user functions must be loaded or present in one of the Blox in the system being tested. If the user functions are not loaded into the system an error will result.

Any other Blox script can call a user created function. These user created scripts are not processed like other standard Blox script sections. Normal Blox scripts are executed automatically in a predetermined location in the sequence controlled by data processing loop. Instead user functions are only processed when the Blox containing the user function calling code request them.

This ability to call methods that perform specific operations by providing numeric or string data as parameters removes the need for ordering the sequence in which a Blox section is processed. User Functions also make the process of re-using code modules easier because they can be created as a generic script section that will be available to any Blox. User Functions called by a standard Blox script will be able to have access the same data that the Blox section calling the user function has at the time the user function is called.

A good practice is to create an Auxiliary block with all the custom functions, and include in a Global Suite System. In this way, all the custom functions will be available to all systems in the suite.

Creating User Functions

User Functions are created in the Blox Editor by using the Script menu New Custom menu item. Start by creating an Auxiliary Block with no default scripts, and add custom functions as needed. The name of the script will be used to call the script using the script.execute function.

Script sections can be removed or added using the Blox Editor script menu item by selecting the **Delete** option when the script section you want removed is highlighted. With all the script sections you don't want removed, click on the Scripts Menu item and then select **New Custom**. This selection brings up a dialog where you can enter the name of your new User Function. The name you give to the user functions will be displayed as a script section in the same way that Trading Blox regular script sections show names.

Calling User Functions

Once you've created the script section, it will open a coding area where you can code the script you want. This script code window will be no different than any of the other script coding windows.

Start by entering the code you will need to achieve the user function's goal. With your code entered, and assuming there is no parsing errors being reported at the bottom of the code window, you can then call this script from anywhere in the system by using the following process:

```
lResult = Script.Execute( "User_Function_Name" , [parameterlist...] )
```

[parameterlist...] This is where you pass values to your new function.

To obtain the value passed back from your user function to the variable `lResult`, you can use Blox Basic's PRINT function, or use the value contained in the `lResult` variable in another calculation

```
Print lResult
```

```
Any_Var = lResult
```

When you execute a user function on the right side of an equals sign '=', the user function will assign the function's result to the variable on the left side of the equal sign. In this case the value contained in `lResult` will be placed in the variable `Any_Var`.

You can call a User Function to print directly to a PRINT statement:

```
Print Script.Execute( "User_Function_Name", [parameterlist...] )
```

In this method the function's result will print directly to TBB's default **Print Output.csv** file and **Log Window**.

When you writing the code for a User Function script, you will need to assign the user function's calculation results to a script's return property. Placing one of the following methods in the user function code will assign the output value you select in the user function so that it is returned to the calling Blox:

```
Script.SetReturnValue( Any_Num or Any_String ) ' Sets the string or number return value.
```

If the `SetReturnValue` function is called more than once, the last call will determine the return value.

You can call a User Function without assigning a value to capture its return result. To do it that way, the calling statement would look like this:

```
Script.Execute( "User_Function_Name", [parameterlist...] )
```

In this case you would need to use one of the following properties to access the user function's results in the calling Blox section where you call the user function:

```
lResult = Script.ReturnValue ' Use for INTEGER OR FLOAT Returns  
Or  
Any_Text = Script.StringReturnValue ' Use for STRING Return
```

Script Methods & Properties

In Trading Blox Builder's standard Blox scripts, the code within the Blox is self contained for the most part. Access to information used within a Blox is determined by location that Blox is called within the code processing loop. For the most part, data is passed to each Blox through the use of **BPV**, or **IPV** variables that have obtained data elsewhere, and in the case of **BPV** variables a Blox can assign values to **BPV** and variables declared within the Blox.

Data can also be assigned and obtained from **BPV** series created using the functions:

```
Any_Name.LoadSymbol
```

Or

```
Another_Name.LoadExternalData
```

Parameter values are primarily passed to a user function using the the following properties:

```

' Variable Containers of Passed Values to User Created Functions
Script.ParameterList[]           ' Use For Integer & FLOAT values
Script.StringParameterList[]     ' Use For String values

' Quantity Count of Passed Parameter Variables in User Function
Script.ParameterCount            ' Count of Integer & FLOAT Variables
Script.StringParameterCount     ' Count of String Variables

' Return Variable Container Of Last User Function Result
Script.ReturnValue               ' Use For Integer Or FLOAT Returns
Script.StringReturnValue         ' Use For String Return

```

User scripts have access to any data that the calling Blox has access to at the time the user function is called. This means that if all the data you need to use in the user function is contained of the IPV or BPV variables available to the calling Blox, then you might not need to pass any new information to the user function.

```

' Subroutine Processes for Setting a Specific Parameter Value in Active
Function
Script.SetParameter( parameter_number, _
                    value )           ' Use For Integer & FLOAT
Parameters
Script.SetStringParameter( parameter number, _
                          String value ) ' Use For String Parameters

' These are Subroutine Processes for Setting the User Function's
RETURN value
Script.SetReturnValue( value )       ' Use For a Integer & FLOAT
RETURN
Script.SetStringReturnValue( String value ) ' Use For a String RETURN

' This is Subroutine Calling Process to Execute a User Created Function
Script.Execute( scriptName, [parameterlist...] )

```

NOTE:

```

' Each Parameter List TYPE IS parsed into each OF the following
variable
' containers based upon the LEFT TO RIGHT sequence IN which the
parameter
' value IS listed when it IS called, AND also the TYPE OF variable
being
' passed:
Script.ParameterList[]           ' Use For Integer Or FLOAT parameters
Or
Script.StringParameterList[]    ' Use For String parameters

```

To pass in a Series, use the GetReference function

```

script.Execute( "MyCustomFunction", 10, "hello", 20, "world",
GetReference( instrument.averageTrueRange ) )

```

To then access values from the series, use the `GetSeriesValue` function:

```
PRINT script.GetSeriesValue( 1, index )
```

9.2 Date Time Functions

For the following functions, "date" must be in the format of **YYYYMMDD**. This can include variables or properties such as [test.currentDate](#) or [instrument.date](#). To set dates into variables, use the Integer type.

Time is in the format HHMM and is also an integer.

Function Name:	Description:
ChartTime	Creates a Chart object compatible date and a time value..
DateToJulian	Returns the number of days since 1900 for a given date. It is useful for calculating the number of days between two dates.
DayMonthYearToDate	Returns the date based on the day, month, year
DayOfMonth	Returns the day of the month
DayOfWeek	Returns the day of the week for the date
DayOfWeekName	Returns the name of the day of the week for the day of the week index
DaysInMonth	Returns the number of calendar days in a month
Hour	Returns the hour for the specified time
JulianToDate	Returns the date for a Julian number
Minute	Returns the minute for the specified time
Month	Returns the month for the specified date
MonthName	Returns the name of the month, for the given month index
SystemDate	Keyword returns the current system (computer) date in a YYYYMMDD format. Displays as YYYY-MM-DD, but can be compared to other integer dates such as instrument.date and test.currentDate
SystemTime	Keyword returns the current system (computer) time
TimeDiff	Returns the difference between two times, in time format
WeekNumberISO	Returns an International Organization for Standardization (ISO) compatible week number for any date since 1900
Year	Returns the year for the specified date

ChartTime

Creates a Chart object compatible date and a time value.

Syntax:

```
ChartTime( YYYYMMDD, HHMM )
```

Parameter:	Description:
YYYYMMDD	Any valid numeric date in the YYYYMMDD format. Date must be a number without

	any delimiting characters.
HHMM	Any valid 24-hour HHMM formatted time value with no delimited characters between the hour and minute section of the time number. Use a zero value when no time value is needed.

Returns:

Date and time values are converted to the number of seconds that have elapsed since 01-01-0001 00:00:00 to the time when executed. This date and time format is the date & time format used by the chart object.

Example:

```
' -----
' test.currentDate without a time value
PRINT "test.currentDate ", test.currentDate
PRINT "ChartTime          ", ChartTime(test.currentDate, 0)
```

Returns:

```
test.currentDate , 2010-07-12
ChartTime        , 63414536400.000000000
```

```
' -----
' YYYYMMDD date in seconds, without a time value
PRINT "ChartTime ", ChartTime(test.currentDate, test.currentTime)
```

Returns:

```
ChartTime 20130126, 63494755200.000000000
```

```
' -----
' YYYYMMDD date in seconds, without a time value
PRINT "ChartTime 20130126 ", ChartTime(20130126, 0)
```

Returns:

```
ChartTime 20130126, 63494755200.000000000
```

```
' -----
' YYYYMMDD date in seconds, without a time value
PRINT "ChartTime ", ChartTime(test.currentDate, test.currentTime)
```

Returns:

```
ChartTime 20130126, 63494755200.000000000
```

```
' -----
' AFTER TRADING DAY SCRIPT ASSIGNS TEST DATES TO BPV Auto-Index SERIES

' Store Test Dates & ChartTime seconds in two numeric series
TestDateArray = test.currentDate
' Converted Test Dates to ChartTime seconds
DateTimeArray = ChartTime( test.currentDate, 0 )
```

```
' AFTER TEST Print Test Dates as elapsed seconds, and YYYYMMDD dates
PRINT "DateTimeArray[3] ", DateTimeArray[3], TestDateArray[3]
PRINT "DateTimeArray[2] ", DateTimeArray[2], TestDateArray[2]
PRINT "DateTimeArray[1] ", DateTimeArray[1], TestDateArray[1]
PRINT "DateTimeArray[0] ", DateTimeArray[0], TestDateArray[0]
```

Returns:

```
DateTimeArray[3] ,63414230400.000000000,20100709.000000000,
DateTimeArray[2] ,63414316800.000000000,20100710.000000000,
DateTimeArray[1] ,63414403200.000000000,20100711.000000000,
DateTimeArray[0] ,63414489600.000000000,20100712.000000000,
```

Notes:

[SystemDate](#) is a compatible date format, but it will return the same date value for a series, but might not. [SystemTime\(\)](#) function is not a compatible format for [ChartTime](#).

Links:

[PRINT](#), [General Properties](#)

See Also:

[SetxAxisDates](#)

DateToJulian

Returns the number of days since 1900 for a given date. It is useful for calculating the number of days between two dates.

Syntax:

```
value = DateToJulian( expression )
```

Parameter:

expression

Description:

Any expression that resolves to a date in the format YYYYMMDD

Returns:

Number of days since 1900 for the given date

Example:

```
daysBetween = DateToJulian( instrument.tradeExitDate ) -  
               - DateToJulian( instrument.date )
```

Links:

See Also:

[Date Time Functions](#)

DayMonthYearToDate

Returns the date based on the day, month, year

Syntax:

```
theDate = DayMonthYearToDate( d, m, y )
```

Parameter:

d, m, y

Description:

The day number, month number, and year number

Returns:

A date in the **YYYYMMDD** based on the dmy input

Example:

```
PRINT DayMonthYearToDate( 21, 5, 1990 )  
  
' PRINTS 19902105
```

Links:**See Also:**

[Date Time Functions](#)

DayOfMonth

Returns the day of the month

Syntax:

```
value = DayOfMonth( expression )
```

Parameter:

expression

Description:

Any expression that resolves to a date in the format YYYYMMDD

Returns:

Day of the month (1 to 31) for the given date

Example:

```
day = DayOfMonth( 20021215 ) ' returns 15
```

Links:**See Also:**

[Date Time Functions](#)

DayOfWeek

Returns the day of the week for the date. There are built in [Constants](#) to compare against:

[SUNDAY](#) , [MONDAY](#) , [TUESDAY](#) , [WEDNESDAY](#) , [THURSDAY](#) , [FRIDAY](#) , [SATURDAY](#)

Syntax:

```
value = DayOfWeek( expression )
```

Parameter:	Description:
expression	Any expression that resolves to a date in the format YYYYMMDD

Returns:

Day of the week for the given date

Example:

```
IF DayOfWeek( test.currentDate ) = MONDAY THEN  
    ' Do the weekly updating here.  
ENDIF
```

Links:

[Constants](#)

See Also:

[Date Time Functions](#)

DayOfWeekName

Returns the name of the day of the week for the day of the week index.

Syntax:

```
value = DayOfWeekName( expression )
```

Parameter:

expression

Description:

Any expression that resolves to an integer between 0 and 6

Returns:

Name of the day of the week for the given day of week number

Example:

```
PRINT DayOfWeekName( DayOfWeek( test.currentDate ) )
```

Links:**See Also:**

[Date Time Functions](#)

DaysInMonth

Returns the number of calendar days in a month.

Syntax:

```
numDays = DaysInMonth( m, y )
```

Parameter:

m, y

Description:

The month number, and year number

Returns:

Number of days in the specified month.

Example:

```
PRINT DaysInMonth( 5, 1990 )
```

```
' PRINTS 31
```

Links:**See Also:**

[Date Time Functions](#)

Hour

Returns the hour for the specified time.

Syntax:

```
value = Hour( expression )
```

Parameter:

expression

Description:

Any expression that resolves to a time HHMM format

Returns:

Hour of the given time.

Example:

```
theHour = Hour( 1055 )      ' returns 10
```

Links:**See Also:**

[Date Time Functions](#)

JulianToDate

Returns the date for a julian number. The reverse of the [DateToJulian](#) function.

Syntax:

```
date = JulianToDate( expression )
```

Parameter:

expression

Description:

any expression that resolves to a valid julian number

Returns:

date in YYYYMMDD format

Example:

```
theDate = JulianToDate( DateToJulian( instrument.date ) + 5 )
```

Links:**See Also:**

[Date Time Functions](#)

Minute

Returns the minute for the specified time.

Syntax:

```
value = Minute( expression )
```

Parameter:	Description:
expression	Any expression that resolves to a time HHMM format.

Returns:

Minute of the given time.

Example:

```
theMinute = Minute( 1055 ) ' returns 55
```

Links:

See Also:

[Date Time Functions](#)

Month

Returns the month for the specified date.

Syntax:

```
value = Month( expression )
```

Parameter:

expression

Description:

Any expression that resolves to a date in the format YYYYMMDD

Returns:

Month for the given date

Example:

```
tradeMonth = Month( 20021215 ) ' returns 12
```

Links:**See Also:**

[Date Time Functions](#)

MonthName

Returns the name of the month, for the given month index

Syntax:

```
value = MonthName( expression )
```

Parameter:

expression

Description:

Any expression that resolves to an integer between 1 and 12

Returns:

Name of the month

Example:

```
PRINT MonthName( Month( test.currentDate ) )  
' Prints January for a date like 20100101
```

Links:

See Also:

[Date Time Functions](#)

SystemDate

Keyword returns the current system (**computer**) date in a **YYYYMMDD** format. Displays as **YYYY-MM-DD**, but can be compared to other integer dates such as `instrument.date` and `test.currentDate`.

Current Simulation's computer date. In **YYYYMMDD** format.

Syntax:

```
Print "SystemDate ", SystemDate
```

Parameter:	Description:
<none>	

Returns:

SystemDate, 2013-01-25

Example:

`SystemDate` replaces `CurrentDate`

Links:

See Also:

[Date Time Functions](#)

SystemTime

Keyword returns the current system (**computer**) time.

Current simulation time. In **HHMM** format.

There are three return formats depending on the parameter passed into the function. If no parameter is passed in, then type 1 is assumed.

Type 1 returns the time in your local format.

Type 2 returns the time in an **HHMM** format, so it can be compared to `instrument.time` and `test.currentTime`.

Type 3 returns the number of seconds since the start of the current day. Useful for timing application processes.

```
SystemTime          8:46:36 AM
SystemTime( 1 )    8:46:36 AM
SystemTime( 2 )    846
SystemTime( 3 )    31596
```

NOTE:

`SystemTime` replaces `CurrentTime`

Syntax:

```
Print "SystemTime() ", SystemTime()
```

Parameter:

Description:

Returns:

```
SystemTime(), 1:28:03 PM
```

Example:

```
Print "SystemTime() ", SystemTime()
```

Links:

See Also:

[Date Time Functions](#)

TimeDiff

Returns the difference between two times, in time format.

Syntax:

```
value = TimeDiff( expression1, expression2 )
```

Parameter:

expression1,
expression2

Description:

Any expression that resolves to a time HHMM format.

Returns:

Difference between the two times, in HHMM format

Example:

```
thedifference = TimeDiff( 1055, 945 )  
' Return 0110
```

Links:**See Also:**

[Date Time Functions](#)

WeekNumberISO

Returns an International Organization for Standardization (ISO) compatible week number for any date since 1900.

Each week can have a number between 1 and 53 depending upon the year. Week 1 of each year begins on the first week of a new calendar year where the first Thursday in January occurs. This means that any week where 1-January falls on a Monday, Tuesday, Wednesday or a Thursday, that week is Week 1. When 1 January falls on a Friday, Saturday, or a Sunday, that week is either week 52 or week 53.

Syntax:

```
WeekNumber = WeekNumberISO( Expression )
```

Parameter:

Expression

Description:

Any expression that resolves to a date in the format **YYYYMMDD**.

Returns:

The week number for any date since 1900

Example:

```
' -----
' WeekNumber - ISO Compatible
' -----
' ISO says to find the first Thursday of each year and then back
' off to that Week's Monday to determine the first date in a year
' when Week #1 occurs.
'
' This code uses that same logic to find the first Thursday and then
' goes back to identify that week's Monday.
'
' When the first date of a year doesn't fall into Week #1, it says
' to use information from the previous year to determine if the first
' date of the year falls into Week #52 or Week #53. This code also
' uses that logic to fill that requirement.
' ~~~~~

Variables: WeekNumber TYPE: INTEGER
Variables: Day_Offset TYPE: INTEGER
Variables: DayName TYPE: STRING

Day_Offset = 0

WeekNumber = WeekNumberISO( Instrument.Date[Day_Offset] )

' .....
' .....
' Print the Results to the Output.CSV File
' .....
```

```
If Instrument.CurrentBar = 1 Then
    ' Send Results to the PRINT LOG
    Print "Date: ", _
          " DOW-Name ", _
          " Week_# "
EndIf

DayName = DayOfWeekName( DayOfWeek(Instrument.Date[Day_Offset]))

' Send Results to the PRINT LOG
Print Instrument.Date[Day_Offset], _
      DayName, _
      WeekNumber
' ~~~~~
```

Links:**See Also:**[Date Time Functions](#)

Year

Returns the year for the specified date.

Syntax:

```
value = Year( expression )
```

Parameter:

expression

Description:

Any expression that resolves to a date in the format YYYYMMDD

Returns:

Year for the given date

Example:

```
tradeYear = Year( 20021215 ) ' returns 2002
```

Links:**See Also:**

[Date Time Functions](#)

9.3 File & Disk Functions

These functions allow file manipulation at the disk level. You can open a file, move a file, copy a file, or delete a file.

Functions:	Description:
ClearLogWindow	Clear all text from main screen's Log Window area.
CloseLogWindow	Close the main screen's Log Window area.
CopyFile	Copies the referenced file.
CreateDirectory	Creates a directory
DeleteFile	Deletes the referenced file.
EditFile	Edits the referenced file.
Extract	Extracts all indicators and IPV Series variable by date time and by instrument to a file.
FileExists	Confirms the existence of the referenced file or directory folder.
FileSize	Returns the referenced file size.
MoveFile	Moves the referenced file from its current location to the referenced destination location.
OpenFile	Open the referenced file and displays it on the screen.
OpenFileDialog	Displays the Windows Open File Dialog.
OpenLogWindow	Open the main screen's Log Window area.
SaveFileDialog	Displays the Windows Save File Dialog.

ClearLogWindow

Syntax:	
Parameter:	Description:
Example:	
Returns:	
Links:	
See Also:	

CloseLogWindow

Syntax:	
Parameter:	Description:
Example:	
Returns:	
Links:	
See Also:	

CopyFile

Copies a file.

Syntax

```
OpenFile( fileName )
```

Parameters

fileName the file to open, copy, delete, or move

returns n/a

Examples

```
CopyFile( "c:\correlation.htm", "c:\corr2.htm" )
```

```
DeleteFile( "c:\correlation.htm" )
```

```
MoveFile( "c:\corr2.htm", "c:\corr3.htm" )
```

```
OpenFile( "c:\corr3.htm" )
```

CreateDirectory

When a needed directory folder is needed this function can create it.

Syntax:

```
success = CreateDirectory( directoryName )
```

Parameter:

Description:

directoryName

Create a folder using the name specified

returns

True if the directory was created; False if the directory creation failed.

Example:

```
' Create a new directory folder named 'MyOrders'
If CreateDirectory( FileManager.DefaultFolder + "MyOrders\" ) THEN
  PRINT "Directory Created"
ELSE
  PRINT "Directory already exists."
ENDIF
```

Links:

[File Manager](#)

See Also:

DeleteFile

Deletes a file.

Syntax

```
OpenFile( fileName )
```

Parameters

fileName

the file to open, copy, delete, or move

returns

n/a

Examples

```
CopyFile( "c:\correlation.htm", "c:\corr2.htm" )
DeleteFile( "c:\correlation.htm" )
MoveFile( "c:\corr2.htm", "c:\corr3.htm" )
OpenFile( "c:\corr3.htm" )
```


OpenFileDialog

OpenFileDialog returns the string of the path the user has selected using the open file dialog.

Syntax

```
completeFilePath = OpenFileDialog( [filter extension], [file Name],  
[start path] )
```

Parameters

filter extension	extensions to filter, like .exe, .txt or .csv
file Name	the file name to suggest
start path	the path to set the dialog to start with
return value	returns a string containing the complete path the user selected

The filter extensions are a paired string separated by a |. So "Text Files|.txt" would display Text Files and show only files ending with .txt.

For multiple extensions for the same display name use a semi colon as in the example below.

```
"Text Files|.txt;.csv"
```

You can set multiple filters such as:

```
"Text Files|.txt|CSV Files|.csv"
```

Examples

```
PRINT OpenFileDialog( "Text Files|.txt;.csv", "orders.txt",  
fileManager.DefaultFolder )  
PRINT SaveFileDialog( "Text Files|.txt;.csv", "orders.txt",  
fileManager.DefaultFolder )
```

OpenLogWindow

Syntax:	
Parameter:	Description:
Example:	
Returns:	
Links:	
See Also:	

SaveFileDialog

SaveFileDialog returns the string of the path the user has selected using the save file dialog.

Syntax

```
completeFilePath = SaveFileDialog( [filter extension], [file Name],
[start path] )
```

Parameters

filter extension	extensions to filter, like .exe, .txt or .csv
file Name	the file name to suggest
start path	the path to set the dialog to start with
return value	returns a string containing the complete path the user selected

The filter extensions are a paired string separated by a |. So "Text Files|.txt" would display Text Files and show only files ending with .txt.

For multiple extensions for the same display name use a semi colon as in the example below.

```
"Text Files|*.txt;*.csv"
```

You can set multiple filters such as:

```
"Text Files|*.txt|CSV Files|*.csv"
```

Examples

```
PRINT OpenFileDialog( "Text Files|*.txt;*.csv", "orders.txt",  
fileManager.DefaultFolder )  
PRINT SaveFileDialog( "Text Files|*.txt;*.csv", "orders.txt",  
fileManager.DefaultFolder )
```

9.4 General

These are general program functions intended as optional references in a block module.

Function:	Description:
BuildDividendFiles	This special function will kick off the Build Dividend Files process. The test then needs to be aborted and run again with this new data.
ColorRGB	Function assigns its return color value to another variable, or it can be used as value in a color parameter field.
FileVersion	Returns the file version such as 3.4.1.12
FileVersionNumerical	Returns the numerical such as 03040112
GetRegistryKey	Gets a value from the registry
LicenseName	Returns the Trading Blox License Name currently in use. Used in encrypted systems to lock a system to a particular Trading Blox user.
LineNumber	Returns the current line number of the script, for PRINT debugging purposes.
MessageBox	Presents a message box
PlaySound	Plays a sound
PreferenceItems	
ProductVersion	Returns the product version such as 3.4
ProductVersionNumerical	Returns the numerical such as 03040000
SetRegistryKey	Sets a value into the registry

BuildDividendFiles

Function creates dividend files using the CSI Data Service Unfair Advantage Stock subscription service.

Syntax:

`BuildDividendFiles`

Parameter:

< none >

Description:

Process builds files that are not made available for test simulation during dividend file creation. This means the first pass is used to create the files, and the second pass will make the files available for testing.

If the Main Menu screen open Log window reports errors, the Stock subscription has expired, or Trading Blox preference setting for the location of the Unfair Advantage software installation needs to be changed.

Example:

```
' Begin Dividend File Creation using CSI Stock Subscription  
BuildDividendFiles
```

Results:**Links:****See Also:**

ColorRGB

Function assigns its return color value to another variable, or it can be used as value in a color parameter field.

Use to change the color of any function that has a color parameter.

Function requires values in all three of its parameter fields.

Syntax:

```
' Create a color based upon the values of the three parameters
ColorValue = ColorRGB( BlueValue, GreenValue, RedValue
```

Parameter:	Description:
BlueValue GreenValue RedValue	All three parameter fields and their values can be any integer that begins at 0 and ends at 255. Each value entered into each of these three parameters determines the color-number this function creates. Number created is then the value needed to duplicate that color in a chart, or other process where a color value is needed.

Examples:

```
' Assign ColorRGB value to another variable
' Generate Blue Color Number
ColorValue1 = ColorRGB( 255, 0, 0 )
' Generate Green Color Number
ColorValue2 = ColorRGB( 0, 255, 0 )
' Generate Red Color Number
ColorValue3 = ColorRGB( 0, 0, 255 )
```

OR

```
' Assign ColorRGB values to chart object function
' Use Blue Color value for data series Line1
chart.AddLineSeries( AsSeries( Line1 ), 100, "Line1", ColorRGB( 255, 0,
' Use Green Color value for data series Line1
chart.AddLineSeries( AsSeries( Line2 ), 100, "Line2", ColorRGB( 0, 255,
' Use Red Color value for data series Line1
chart.AddLineSeries( AsSeries( Line3 ), 100, "Line3", ColorRGB( 0, 0, 255 )
```

OR

```
' Assign ColorRGB values to SerSeriesColorStyle function
' Use Blue Color value for PlotLine1 series Line1
SetSeriesColorStyle( PlotLine1, ColorRGB( 255, 0, 0 ) )
' Use Green Color value for PlotLine2 series Line1
SetSeriesColorStyle( PlotLine2, ColorRGB( 0, 255, 0 ) )
' Use Red Color value for PlotLine3 series Line1
SetSeriesColorStyle( PlotLine3, ColorRGB( 0, 0, 255 ) )
```

OR

```

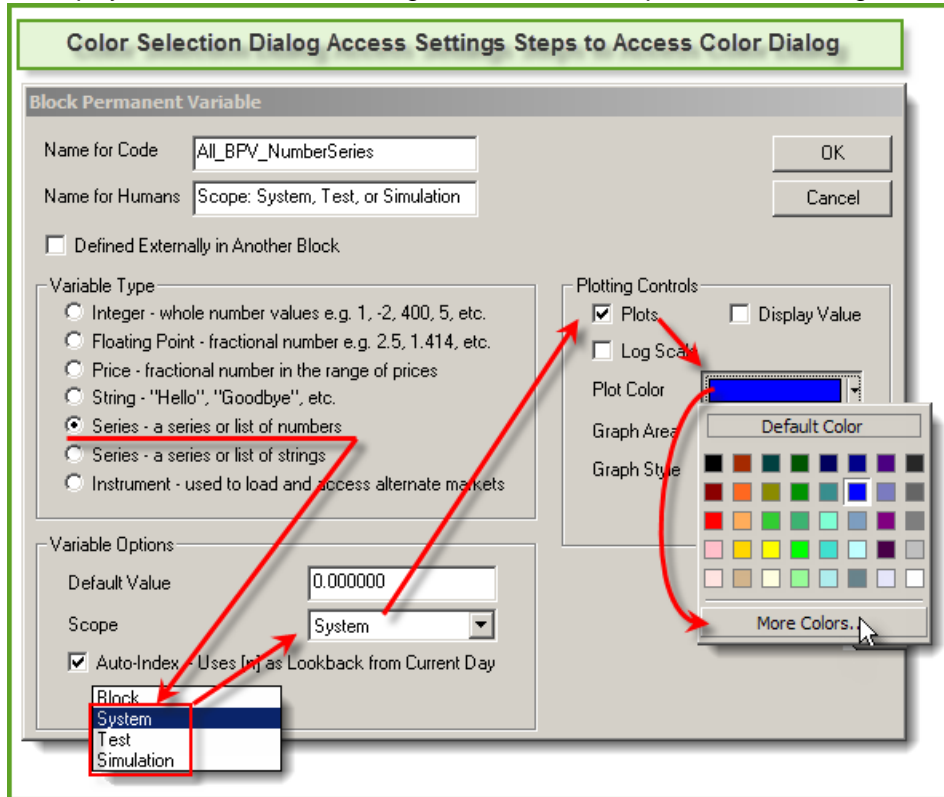
' Consider a Random number color assignment
'
'                               BLUE           GREEN           RED
plotColor = ColorRGB( Random(255), Random(255), Random(255) )

```

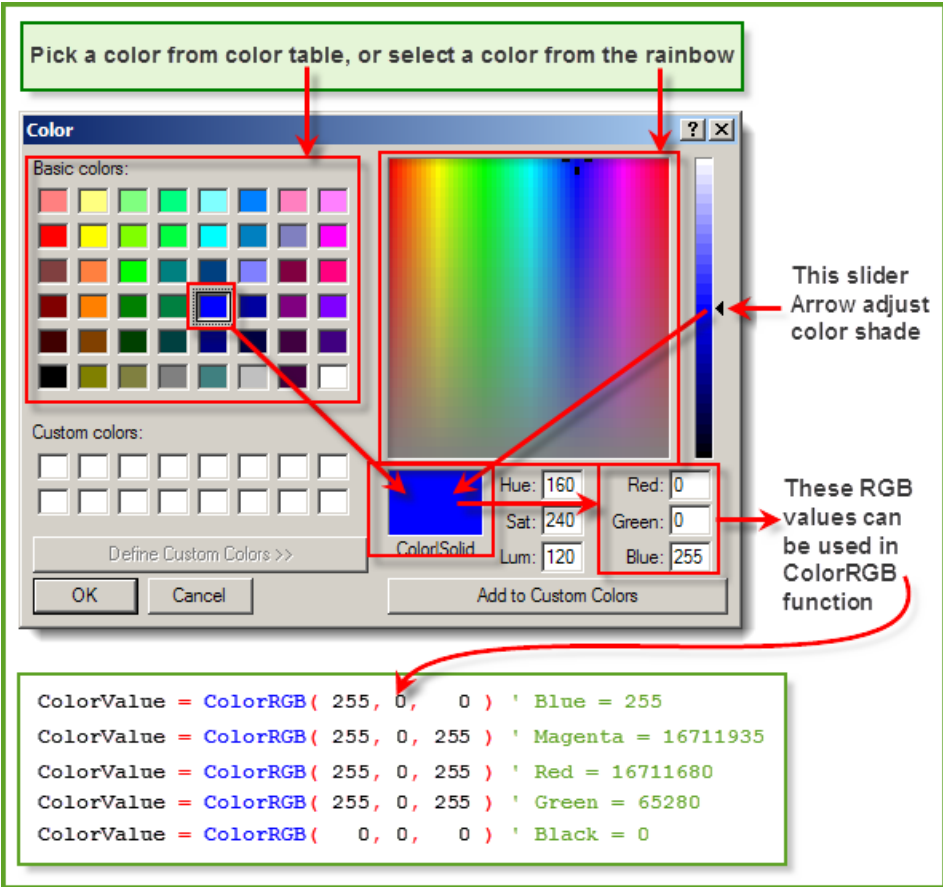
Trading Blox Color Selection Dialog:

All BPV Series will provide access to the Trading Blox Color Selection Dialog when the BPV series uses a System, Test or Simulation Scope setting with a BPV numeric series.

To display the color selection dialog, follow the click steps in this next image:



When the "More Colors..." button is clicked the dialog in this next image will appear:



Pick a color from color table, or select a color from the rainbow

Color

Basic colors:

Custom colors:

Define Custom Colors >>

OK Cancel

Add to Custom Colors

Hue: 160 Red: 0
Sat: 240 Green: 0
Lum: 120 Blue: 255

ColorSolid

This slider Arrow adjust color shade

These RGB values can be used in ColorRGB function

```

ColorValue = ColorRGB( 255, 0, 0 ) ' Blue = 255
ColorValue = ColorRGB( 255, 0, 255 ) ' Magenta = 16711935
ColorValue = ColorRGB( 255, 0, 255 ) ' Red = 16711680
ColorValue = ColorRGB( 255, 0, 255 ) ' Green = 65280
ColorValue = ColorRGB( 0, 0, 0 ) ' Black = 0

```

Just about any color's RGB value can be discovered using this dialog. However, if the chart image where this color is to be used will appear in a report generated with a HTML Browser process that is used to create Trading Blox reports, picking a color from the Basic Color Matrix Table will keep the colors used within the Safe-Color range that are easily reproduced using a HTML process.

Applying the RGB, (Red, Green, Blue) values to Trading Blox's ColorRGB function, place the color numbers using Blue, Green and Red as the first, second and third parameter locations

Script Color Assignment Examples:

```

'           Blue   Green   Red
PlotColor1 = ColorRGB( 255, 0, 0 ) ' Plot Blue Color
PlotColor2 = ColorRGB( 0, 255, 0 ) ' Plot Green Color
PlotColor3 = ColorRGB( 0, 0, 255 ) ' Plot Red Color

' Trade Color Preference Settings Color Numbers values
PlotColor1 = ColorCustom1 ' Use Preference ColorCustom1 Value
PlotColor2 = ColorCustom2 ' Use Preference ColorCustom2 Value
PlotColor3 = ColorCustom3 ' Use Preference ColorCustom3 Value

```

Links:

[AddLineSeries](#), [Colors](#), [SetSeriesColorStyle](#)

FileVersion

Syntax:

Parameter:

Description:

Example:

Results:

Links:

WWW.TRADING-SOFTWARE-DOWNLOAD.COM

See Also:

FileVersionNumerical

Syntax:	
Parameter:	Description:
Example:	
Results:	
Links:	
See Also:	

GetRegistryKey

Use this function to get a registry key value, that was set with the SetRegistryKey function

Syntax

```
keyValue = GetRegistryKey( keyName, [subKeyName] )
```

Parameters

keyName	the name of the key, used by SetRegistryKey
subKeyName	the name of the subKey

Examples

```
SetRegistryKey( "HelloWorld", "What a wonderful day." )
PRINT GetRegistryKey( "HelloWorld" )

SetRegistryKey( "Count", 0 )

count = GetRegistryKey( "Count" )
count = count + 1
SetRegistryKey( "Count", count )
PRINT GetRegistryKey( "Count" )
```



Syntax:	
Parameter:	Description:
Example:	
Results:	
Links:	
See Also:	
<hr/>	

LicenseName

Returns the Trading Blox License Name currently in use. Used in encrypted systems to lock a system to a particular Trading Blox user.

Syntax:

Parameter:

Description:

Example:

Results:

Links:

See Also:

LineNumber

Use this function for debugging purposes using along with the PRINT function to get the line number of the script section where this function is placed.

Syntax:

`Print LineNumber`

Parameter:

none

Description:**Returns:**

Returns the current line number of the script where this function is placed.

Example:**Links:**

[Block Object](#)

See Also:

Message Box

MessageBox function presents the user with a message box and stops processing of the test.

Syntax:

```
returnValue = MessageBox( message, [Button Options], [Icon and Sound] )
```

Parameter:	Description:
message	String message to display
[Button Options]	Optional decimal number from the type1 list
[Icon and Sound]	Optional decimal number from the type2 list
returnValue	Decimal number from the return value list

Dialog Options:

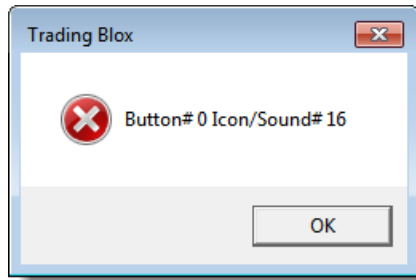
Button Options:	Icon and Sound Options:	Button Return Values:
0 -- OK	16 -- Icon X, Critical Stop Sound	1 -- OK
1 -- OK/Cancel	32 -- Icon Question, Ding Sound	2 -- Cancel
2 -- Abort/Retry/Ignore	48 -- Icon Exclamation, Exclamation Sound	3 -- Abort
3 -- Yes/No/Cancel	64 -- Icon Information, Error Sound	4 -- Retry
4 -- Yes/No	80 -- No Icon, Ding Sound	5 -- Ignore
5 -- Retry/Cancel	128 -- No Icon, No Sound	6 -- Yes
6 -- Cancel/Try Again/Continue		7 -- No
		10 -- Try Again
		11 -- Continue

Example:

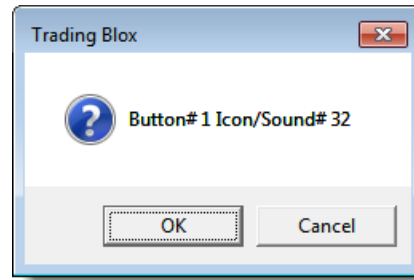
```
' Create Message Box Parameter Details:
sMsg = "Button# " + AsString(iButton, 0) _
      + " Icon/Sound# " + AsString(iIconSound, 0)

' Ask User if it is OK to continue Processing Data
iResult = MessageBox( sMsg, iButton , iIconSound )
```

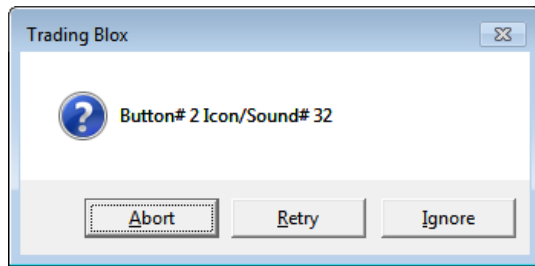
Dialog Examples:



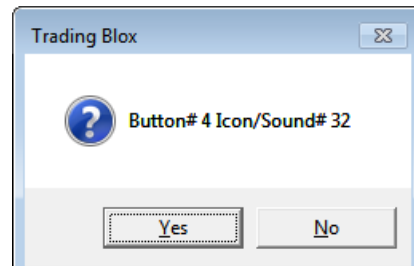
Button - OK & Icon X, Critical Stop Sound



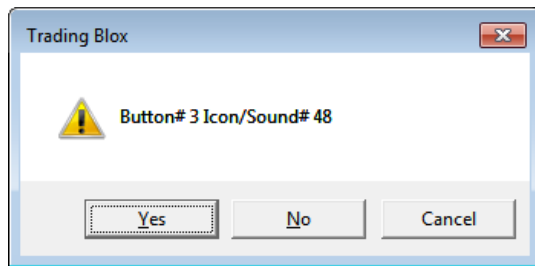
Button - OK/Cancel & Icon Question, Ding Sound



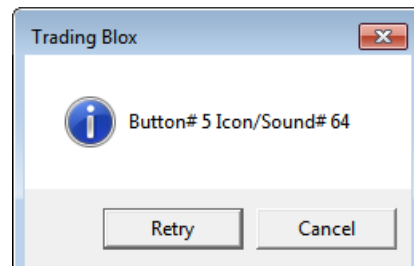
Button -Abort/Retry/Ignore & Icon Question, Ding Sound



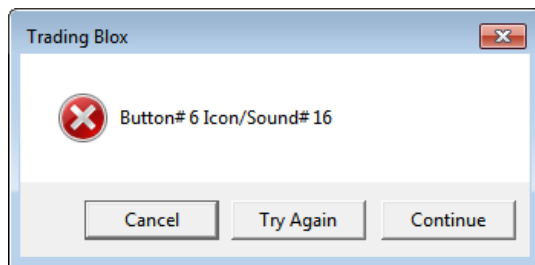
Button -Yes/No & Icon Question, Ding Sound



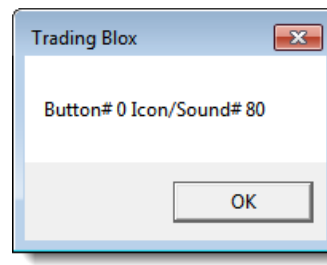
Button -Yes/No/Cancel & Icon Exclamation, Exclamation Sound



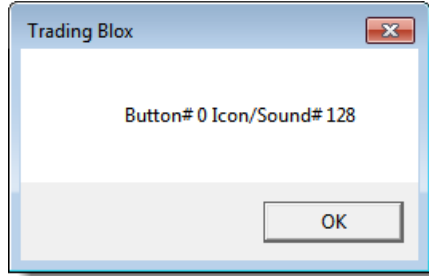
Button -Retry/Cancel & Icon Information, ERROR Sound



Button -Cancel/Try Again/Continue & Icon X, Critical Stop Sound



Button -OK & Icon X, No Icon, Ding Sound



Button -OK & Icon X, No Icon, No Sound

Example:

```
' Ask User if it is OK to continue Processing Data
result = MessageBox( "Is it OK to process data for " + instrument.date

' If the message box indicates the "No" button was pressed,...
If result = 7 THEN
  ' Send User Warning Message Trading Blox is Aborting Test
  test.AbortSimulation( "Your Finished!" )
ELSE
  ' Send to Print Output & Log Window Processing Success
  PRINT "Processing trades for ", instrument.symbol, _
        + " on ", instrument.date
ENDIF
```

Example:

```
' ~~~~~
' 0 -- OK
' 1 -- OK/Cancel
' 2 -- Abort/Retry/Ignore
' 3 -- Yes/No/Cancel
' 4 -- Yes/No
' 5 -- Retry/Cancel
' 6 -- Cancel/Try Again/Continue

' Assign Button Option:
iButton = iButtonOption ' iButtonOption is Integer Type Parameter
' ~~~~~
' Button -OK & Icon X, No Icon, No Sound
' 16 -- Icon X, Critical Stop Sound
' 32 -- Icon Question, Ding Sound
' 48 -- Icon Exclamation, Exclamation Sound
' 64 -- Icon Information, ERROR Sound
' 80 -- No Icon, Ding Sound
' 120 -- No Icon, No Sound
```

```

' BPV Number Series - Manual Index
SoundOption[1] = 16 ' Critical Stop
SoundOption[2] = 32 ' Question Mark
SoundOption[3] = 48 ' Exclamation Sound
SoundOption[4] = 64 ' ERROR Sound
SoundOption[5] = 80 ' No Icon Ding
SoundOption[6] =128 ' No Icon Sound ?

' Assign Selection Sound option - iSoundOption is a Selector Type Parameter
iIconSound = SoundOption[iIconSoundOption + 1]
' ~~~~~
' 1 -- OK
' 2 -- Cancel
' 3 -- Abort
' 4 -- Retry
' 5 -- Ignore
' 6 -- Yes
' 7 -- No
' 10 -- Try Again
' 11 -- Continue
' BPV String Series - Manual Index
ReturnMeaning[1] = "1 -- OK"
ReturnMeaning[2] = "2 -- Cancel"
ReturnMeaning[3] = "3 -- Abort"
ReturnMeaning[4] = "4 -- Retry"
ReturnMeaning[5] = "5 -- Ignore"
ReturnMeaning[6] = "6 -- Yes"
ReturnMeaning[7] = "7 -- No"
ReturnMeaning[8] = "8 -- Error"
ReturnMeaning[9] = "9 -- Error"
ReturnMeaning[10] = "10 -- Try Again"
ReturnMeaning[11] = "11 -- Continue"
ReturnMeaning[12] = "12 -- Error"

' ~~~~~
' Create Message Box Parameter Details:
sMsg = "Button# " + AsString(iButton, 0) _
      + " Icon/Sound# " + AsString(iIconSound, 0)

' Ask User if it is OK to continue Processing Data
iResult = MessageBox( sMsg, iButton , iIconSound )

' Display Message Box Return Value
PRINT "Message Box Returned: " + ReturnMeaning[iResult]
' ~~~~~

```

Button Returns:

```

Message Box Returned: 1 -- OK
Message Box Returned: 2 -- Cancel
Message Box Returned: 4 -- Retry
Message Box Returned: 4 -- Retry

```

Message Box Returned: 7 -- No
Message Box Returned: 6 -- Yes
Message Box Returned: 2 -- Cancel
Message Box Returned: 10 -- Try Again

Links:

[AbortSimulation](#), [General](#)

See Also:**PlaySound**

Plays a sound file from the Sounds folder.

The Sounds folder is located in the main Trading Blox folder.

Example:

```
PlaySound( "Test Done.wav" )
```

Syntax:

Parameter:	Description:

Example:

Results:

Links:

See Also:

Preference Items

Preferences:

NumberOfExtraDataFields
LoadVolume
LoadUnadjustedClose
ProcessDailyBars
ProcessWeeklyBars
ProcessMonthlyBars
ProcessWeekends
RaiseNegativeDataSeries
YearsOfPrimingData

Colors set in preferences that can be used in scripting:

ColorBackground
ColorUpBar
ColorDownBar
ColorUpCandle
ColorDownCandle
ColorCrossHair
ColorGrid
ColorLongTrade
ColorShortTrade
ColorTradeEntry
ColorTradeExit
ColorTradeStop
ColorCustom1
ColorCustom2
ColorCustom3
ColorCustom4

ProductVersion

Syntax:

Parameter:	Description:

Example:

Results:

Links:

See Also:

ProductVersionNumerical

Syntax:	
Parameter:	Description:
Example:	
Results:	
Links:	
See Also:	

SetRegistryKey

Use this function to set a registry key.

Syntax

```
SetRegistryKey( keyName, keyValue, [subKeyName] )
```

Parameters

keyName	the name of the key, used by SetRegistryKey
keyValue	the string value of the key. numbers will be converted to strings
subKeyName	the name of the subKey

Examples

```
SetRegistryKey( "HelloWorld", "What a wonderful day." )
PRINT GetRegistryKey( "HelloWorld" )

SetRegistryKey( "Count", 0 )

count = GetRegistryKey( "Count" )
```

```
count = count + 1  
SetRegistryKey( "Count", count )  
PRINT GetRegistryKey( "Count" )
```

Syntax:	
Parameter:	Description:
Example:	
Results:	
Links:	
See Also:	

9.5 Mathematical Functions

Table description will be added soon. Until then, click on function link to see description in the function's topic page.

Function:	Description:
AbsoluteValue	
ArcCosine	
ArcSine	
ArcTangent	
ArcTangentXY	
Average	
CAGR	
Ceiling	
Correlation	
CorrelationLog	
Cosine	
DegreesToRadians	
EMA	


```
value = AbsoluteValue( -5.6 )      ' Returns 5.6  
value = AbsoluteValue( "Hello" )  ' Returns 0.
```

ArcCosine

Returns the arc cosine of an angle specified in radians. The range of the result is 0 to PI radians. To convert angle from radians to degrees use [RadiansToDegrees](#) function.

Short form: "acos".

Syntax

```
value = ArcCosine( expression )
```

iParameters

expression	any expression that resolves to a valid cosine range -1 to 1
returns	the arc cosine of expression

Examples

value = ArcCosine(0)	' Returns 1.570795327
value = ArcCosine(0.5)	' Returns 1.047197551
value = RadiansToDegrees(ArcCosine(0.5))	' Returns 60

ArcSine

Returns the arc sine of an angle specified in radians. The range of the result is 0 to PI radians. To convert angle from radians to degrees use [RadiansToDegrees](#) function.

Short form: "asin".

Syntax

```
value = ArcSine( expression )
```

iParameters

expression	any expression that resolves to an angle in radians
returns	the arc sine of expression

Examples

value = ArcSine(PI)	' Returns 5.6.
value = ArcSine(PI / 2)	' Returns 5.6.

ArcTangent

Returns the arc tangent of an angle specified in radians. The range of the result is 0 to PI radians. To convert angle from radians to degrees use [RadiansToDegrees](#) function.

Short form: "atan".

Syntax

```
value = ArcTangent( expression )
```

iParameters

expression	any expression that resolves to an angle in radians
returns	the arc tangent of expression

Examples

```
value = ArcTangent( PI )           ' Returns 5.6.
value = ArcTangent( PI / 2 )       ' Returns 5.6.
```

ArcTangentXY

Returns the arc tangent of an angle specified in radians by the number X / Y. Returns result in radians. The range of the result is -PI to PI radians. The **ArcTangentXY** function uses the signs of both parameters to determine the quadrant of the return value. To convert angle from radians to degrees use [RadiansToDegrees](#) function.

Short form: "atan2".

Syntax

```
value = ArcTangentXY( X, Y )
```

iParameters

x	any expression
y	any expression
returns	the arc tangent of x/y

Example

```
value = ArcTangentXY( PI, 2 )       ' Returns 5.6.
```

Average

See series function [Average](#).

CAGR

The CAGR function will return the Compounded Annual Growth Rate as computed internally by Trading Blox.

Syntax

CAGR(*days*, *starting value*, *ending value*)

Trading Blox uses the following formula to compute CAGR

$$\text{CAGR} = (\text{ending value} / \text{starting value}) ^ { (1 / (\text{days} / 365.25)) } - 1$$

Ceiling

Ceiling reduces a decimal value to an integer as follows:
 Values greater than zero return the next integer away from zero.
 Values less than zero return the next integer towards zero.

Syntax:

```
Ceiling( AnyValue )
```

Parameter:	Description:
AnyValue	Any numeric value, Or any expression that results in a numeric value.

Example:

```

' ~~~~~
' BPV Manual Series Test Values
' ~~~~~

dVal[1] = -1.50
dVal[2] = -1.00
dVal[3] = -0.50
dVal[4] = 0.00
dVal[5] = 0.50
dVal[6] = 1.00
dVal[7] = 1.50

' ~~~~~

PRINT "Ceiling Function"
PRINT "-----"

FOR Ndx = 1 TO 7
  ' Ceiling Calculation
  PRINT "Ceiling(" + AsString(dVal[Ndx], 2) + ") = ", Ceiling( dVal[Ndx]
Next ' Ndx

' ~~~~~

```

Results:

Ceiling Function:

```

Ceiling(-1.50) = -1
Ceiling(-1.00) = -1
Ceiling(-0.50) = 0
Ceiling(0.00) = 0
Ceiling(0.50) = 1
Ceiling(1.00) = 1
Ceiling(1.50) = 2

```

Links:

[AsString](#), [FOR](#)

See Also:

[AsInteger](#), [Floor](#)

Correlation

See series function [Correlation](#).

CorrelationLog

See series function [CorrelationLog](#).

Cosine

Returns the arc sine of an angle specified in radians. The range of the result is 0 to PI radians. To convert angle from radians to degrees use [RadiansToDegrees](#) function.

Short form: "cos".

Syntax

```
value = Cosine( expression )
```

iParameters

expression	any expression that resolves to an angle in radians
returns	the cosine of expression

Examples

```
value = Cosine( PI )           ' Returns 1.0.
value = Cosine( PI / 2 )      ' Returns 0.0 approximately.
```

DegreesToRadians

Returns the angle in radians corresponding with an angle specified in degrees. The range of the result is 0 to PI radians. To convert angle from radians to degrees use [RadiansToDegrees](#) function.

Short form: "degtorad".

Syntax

```
value = DegreesToRadians( expression )
```

iParameters

expression	any expression that resolves to an angle in degrees
returns	the radians corresponding with expression

Examples

```
value = DegreesToRadians( 180 )           ' Returns 5.6.
value = DegreesToRadians( PI / 2 )      ' Returns 5.6.
```

EMA

Returns the Exponential Moving Average, based on the last value of the series, the new value, and the number of days in the moving average.

Syntax

```
value= EMA( lastValueOfSeries, movingAverageDays, newValue )
```

Parameters

<i>lastValueOfSeries</i>	the previous value in the series
<i>movingAverageDays</i>	the number of days in the moving average
<i>newValue</i>	the new value
returns	the current EMA value

Examples

A calculated indicator could be defined as follows. This would be the moving average of today's close minus yesterday's close, where the name of this calculated indicator is "closeChangeMovingAverage," and the number of days in the moving average is "daysInMovingAverage."

```
EMA( closeChangeMovingAverage[1], daysInMovingAverage,  
instrument.close - instrument.close[1] )
```

Or more simply, the 10 day exponential moving average of the close would be defined as follows, where closeEMA is a series variable.

```
closeEMA = EMA( closeEMA[1], 10, instrument.close )
```


Floor

Floor reduces a decimal value to an integer as follows:
 Values greater than zero return the next integer towards zero.
 Values less than zero return the next integer away from zero.

Syntax:

```
Floor( AnyValue )
```

Parameter:	Description:
AnyValue	Any numeric value, Or any expression that results in a numeric value.

Example:

```
' ~~~~~
' BPV Manual Series Test Values
' ~~~~~

dVal[1] = -1.50
dVal[2] = -1.00
dVal[3] = -0.50
dVal[4] = 0.00
dVal[5] = 0.50
dVal[6] = 1.00
dVal[7] = 1.50

' ~~~~~

PRINT "Floor Function:"
PRINT "-----"

FOR Ndx = 1 TO 7
  ' Floor Calculations
  PRINT "Floor(" + AsString(dVal[Ndx], 2) + ") = ", Floor( dVal[Ndx] )
Next ' Ndx
' ~~~~~
```

Results:

```
Floor Function:
-----
Floor(-1.50) = -2
Floor(-1.00) = -1
Floor(-0.50) = -1
Floor(0.00) = 0
Floor(0.50) = 0
Floor(1.00) = 1
Floor(1.50) = 1
```

Links:

[AsString](#), [FOR](#), [PRINT](#)

See Also:

[AsInteger](#), [Ceiling](#)

Hypotenuse

Calculates the length of the hypotenuse of a right triangle, given the length of the two sides `sideOne` and `sideTwo`.

Short form: "hypot".

Syntax

```
value = Hypotenuse( sideOne, sideTwo )
```

Parameters

<code>sideOne</code>	any expression used to define the length of the first side
<code>sideTwo</code>	any expression used to define the length of the second side
returns	the length of the hypotenuse

Examples

```
value = Hypotenuse( 3, 4 )      ' Returns 5
value = Hypotenuse( 9, 16 )   ' Returns 25
```

IfThenElse

Returns the second parameter value if the first parameter value is true. Returns the third parameter value if the first parameter value is false. This function is analogous to the Microsoft Excel IF function.

Syntax

```
value = IfThenElse( condition, trueValue, falseValue )
```

Parameters

<code>condition</code>	a condition that evaluates to true or false
<code>trueValue</code>	the value returned if condition is TRUE
<code>falseValue</code>	the value returned if condition is FALSE
returns	<code>trueValue</code> if condition is TRUE otherwise <code>falseValue</code>

Examples

```
value = IfThenElse( 1 = 2, 3, 4 ) ' Returns 4
value = IfThenElse( 2 = 2, 3, 4 ) ' Returns 3
```

This function is useful in calculated indicators, where you can only enter an expression.

Example of calculated indicator which will return the true low of the day. This could be part of a true range calculation:

```
IfThenElse( instrument.close[1] < instrument.low, instrument.close[1],  
instrument.low )
```

IsUndefined

Returns true if the variable is undefined. The only variables that are set as undefined are series and indicators prior to priming.

Syntax

```
booleanValue = IsUndefined( variable )
```

Parameters

variable	the variable to evaluate
returns	true if the variable is undefined

Examples

```
notDefined = IsUndefined( instrument.averageTrueRange )
```


Log

Returns the logarithm of a number.

Syntax

```
value = Log( number [ , base ] )
```

Parameters

number	any expression
base	the base of the logarithm, if omitted it will return the natural logarithm (assumes base e)
returns	the log value

Examples

value = Log (1)	' Returns 0
value = Log (16, 2)	' Returns 4
value = Log (100, 10)	' Returns 2

Max

Returns the highest value in a list of values. This function takes an unlimited number of arguments but requires at least one argument. See also [Highest](#).

Syntax

```
value = Max( expression, expression, expression, ... )
```

Parameters

expression1	any numeric expression
expression2	any numeric expression
expression3	any numeric expression
returns	the highest value of the list of expressions

Examples

value = Max (5, 6, 8)	' Returns 8
value = Max (2 + 1, 5)	' Returns 5

Min

Returns the lowest value in a list of values. This function takes an unlimited number of arguments but requires at least one argument. See also [Lowest](#).

Syntax

```
value = Min( expression1, expression2, expression3, ... )
```

Parameters

expression1	any numeric expression
expression2	any numeric expression
expression3	any numeric expression
returns	the lowest value of the list of expressions

Examples

```
value = Min( 5, 6, 8 )           ' Returns 5
value = Min( 2 + 1, 5 )         ' Returns 3
```

Min or [Max](#) can be used in place of an IF loop in certain cases. Instead of:

```
IF ( var1 <= var2 ) THEN
    var3 = var1
ELSE
    var3 = var2
ENDIF
```

You could just use this:

```
var3 = Min(var1, var2)
```

RadiansToDegrees

Returns the angle in degrees corresponding with an angle specified in radians. The range of the result is 0 to 360 degrees. To convert angle from degrees to radians use [DegreesToRadians](#) function.

Short form: "radtodeg".

Syntax

```
value = RadiansToDegrees( expression )
```

Parameters

expression	any expression
returns	the angle in degrees corresponding with the angle specified in radians

Examples

```
value = RadiansToDegrees( PI )           ' Returns 360
value = RadiansToDegrees( PI / 2 )       ' Returns 180
```

Random

Returns a random integer given a range of integers.

If just the range is passed in, the random value returned will be between 1 and the range. If both the

lowerValue and the optional upperValue is passed in, the random value returned will be between the lowerValue and the upperValue. The maximum value that can be passed to this function is 2147483647.

Syntax

```
value = Random( range [ or lowerValue ], [ upperValue ] )
```

iParameters

range or lowerValue	range or lower value of range
upperValue	optional upper value of range

returns	the random value
---------	------------------

Examples

```
PRINT Random( 10 )           ' Returns a random number from 1
to 10
PRINT Random( 10, 20 )      ' Returns a random number from 10 to 20
```

RandomDouble

Returns a random double between 0 and 1.

Syntax

```
value = RandomDouble( )
```

Examples

```
PRINT RandomDouble( )      ' Returns a random double between 0 and
1
```

RandomSeed

Seeds the random number generator with the optional seed value, or the time if seed value is excluded, so that the sequence of random numbers is different every time a test is run. If you don't use this function the Random function will return the same sequence of random numbers for every simulation run. This is good for debugging a problem, but not usually a desired outcome.

Note: Best to use in the Before Simulation script, so it is run just once at the start of the test.

Syntax

```
value = RandomSeed( [ seedValue ] )
```

iParameters

seedValue	optional seedValue
-----------	--------------------

returns	the seed value
---------	----------------

Examples

```
' Seeds the random number generator with the time. Returns the value
used.

seedValue = RandomSeed

' Seeds the random number generator with the current parameter test

RandomSeed( test.currentParameterTest )
```

Round

Returns the rounded value.

Syntax

```
value = Round( expression1, decimals )
```

Parameters

expression1	any numeric expression
decimals	number of decimals to round
returns	the lowest value of the list of expressions

Examples

```
value = Round( 5.123456, 2 )      ' Returns 5.12
value = Round( 1.25 + 1.5, 1 )    ' Returns 2.8
value = Round( 12345, -2 )        ' Returns 12300
```

Sign

Returns a value of 1 when the sign of a value is Positive, and a value of -1 when the sign of a value is negative.

Syntax:

```
Sign( AnyValue )
```

Parameter:

AnyValue

Description:

Any numeric value, Or numeric expression that results in a numeric value.

Examples:

```
' ~~~~~
' Simple Print Statement Examples
' ~~~~~
PRINT Sign( 13 )      ' Returns 1 - Indicating Sign is Positive
PRINT Sign( -13 )    ' Returns -1 - Indicating Sign is Negative
PRINT Sign( 2 * 2 )  ' Returns 1 - Indicating Sign is Positive
PRINT Sign( -2 * 2 ) ' Returns -1 - Indicating Sign is Negative
Or
' ~~~~~
' Control Sign of a Number
' ~~~~~
' Test this number
AnyNumber = 13
' Show Number
PRINT "AnyNumber ", AnyNumber ' Returns 13

' Get Sign of AnyNumber
NumberSign = Sign( AnyNumber )
' Show Number
PRINT "NumberSign ", NumberSign ' Returns 1

' Get Absolute Value of AnyNumber
AnyNumber = AnyNumber * NumberSign
PRINT "AnyNumber ", AnyNumber ' Returns 13

' Test this number
AnyNumber = -2 * 2
PRINT "AnyNumber ", AnyNumber ' Returns -4
' Assign Result of Sign Function
NumberSign = Sign( AnyNumber )
PRINT "NumberSign ", NumberSign ' Returns -1

' Use Sign Result to Absolute Value
AnyNumber = AnyNumber * NumberSign
PRINT "AnyNumber ", AnyNumber ' Returns 4
```

```
| ~~~~~  
| Show Sign of Last Calculation  
| ~~~~~  
If Sign( AnyNumber ) = TRUE THEN  
    PRINT "Sign( AnyNumber ) is Positive"  
ELSE  
    PRINT "Sign( AnyNumber ) is Negative"  
ENDIF
```

Returns:

Sign(AnyNumber) is Positive

Links:

[If THEN ELSE ENDIF, PRINT](#)

Sine

Returns the sine of an angle specified in radians. The range of the result is 0 to PI radians. To convert angle from radians to degrees use [RadiansToDegrees](#) function.

Short form: "sin".

Syntax

```
value = Sine( expression )
```

Parameters

expression	any expression that resolves to an angle in radians
returns	the sine of the angle specified in radians

Examples

value = Sine (PI)	' Returns 0.0 approximately
value = Sine (PI / 2)	' Returns 1.0
value = Sine (DegreesToRadians(30))	' Returns 0.5

Square Root

Returns the square root of the specified number. If expression is a negative number, SquareRoot returns square root of its absolute value.

Short form: "sqr".

Syntax

```
value = SquareRoot( expression )
```

Parameters

expression	any numeric expression
returns	the square root of the specified number

Examples

value = SquareRoot (4)	' Returns 2
value = SquareRoot (-4)	' Returns 2
value = SquareRoot (2)	' Returns 1.414.

StandardDeviation

See series function [StandardDeviation](#).

StandardDeviationLog

See series function [StandardDeviationLog](#).

SumValues

Returns the sum of a list of values. This function takes an unlimited number of arguments but requires at least one argument.

Syntax

```
value = SumValues( expression1, expression2, expression3, ... )
```

Parameters

expression1	any numeric expression
expression2	any numeric expression
expression3	any numeric expression
returns	the sum of the list of expressions

Examples

```
value = SumValues( 5, 6, 8 )           ' Returns 19  
value = SumValues( 2 + 1, 5 )         ' Returns 8
```


Tangent

Returns the tangent of an angle specified in radians.

Short form: "tan".

Syntax

```
value = Tangent( expression )
```

Parameters

expression	any expression that resolves to an angle in radians
------------	---

returns	the tangent of the specified angle
---------	------------------------------------

Examples

value = Tangent (PI)	' Returns 0.0 approximately
value = Tangent (PI / 4)	' Returns 1.0

9.6 String Functions

Trading Blox includes several different built-in functions to manipulate strings.

Parameter:	Description:
ASCII & Asc	Returns the ASCII character code corresponding to the first letter in a string.
ASCIIToCharacters & Chr	Converts its parameters from integers to corresponding ASCII characters and returns the string composed of these characters.
FindString & Instr	Returns the position of the first occurrence of one string within another.
FormatString	Formats the number or string using the format string.
GetField	This function will parse a comma delimited string and return the nth field.
GetFieldCount	This function returns the number of comma delimited values in a text string.
GetFieldNumber	This function returns the field number of a string.
LowerCase & LCase	Converts any Upper-Case text characters to Lower-Case.
LeftCharacters & Left	Returns a specified number of characters from the left-side of a string.
MiddleCharacters & Mid	Returns a specified number of characters to copy from the middle of a string.
RemoveCommasBetweenQuotes	Removes the commas that are between quotes from an imported file record string.
RemoveNonDigits	Removes any characters that are not numbers from the string.
ReplaceString	Replaces text found in the search string with the text provided as a replacement text.
RightCharacters & Right	Returns a specified number of characters from the right side of a string.
StringLength & Len	Returns the number of characters in the a string text expression
TrimLeftSpaces & LTrim	Returns a copy of the input string without the leading space characters.
TrimRightSpaces & RTrim	Returns a copy of the input string without the trailing space characters.
TrimSpaces & Trim	Returns a copy of the input string without the leading and trailing spaces.
UpperCase & UCase	Returns Lower-Case text characters to Upper-Case in a string expression.

Note:

Note that to concatenate strings, the Plus (+) sign is used as follows:

Example:

```
string1 = "Hello"  
string2 = "World"  
resultString = string1 + string2
```

```
PRINT resultString
```

Results:

```
Prints "HelloWorld"
```

See Also:

[Data Groups and Types](#)

ASCII

Returns the ASCII character code corresponding to the first letter in a string.

Syntax:

```
value = ASCII( expression )
```

OR

```
value = Asc( expression ) ' Short Keyword form usage: "Asc".
```

Parameter:

expression

Description:

expression, if it is not a string it will be converted to a string

value

ASCII character code

Example:

```
value = ASCII( "A" )           ' Returns 65  
value = ASCII( "a" )           ' Returns 97  
value = ASCII( "1" )           ' Returns 49  
value = Asc( 1 )               ' Returns 49
```

Results:

See above comments.

Links:**See Also:**

[Data Groups and Types](#)

ASCIIToCharacters

Converts its parameters from integers to corresponding **ASCII** characters and returns the string composed of these characters.

Syntax:

```
value = ASCIIToCharacters( expression )
OR
value = Chr( expression )      ' Short form usage: "Chr"
```

Parameter:	Description:
expression	Any ASCII character value, or a list of comma separated ASCII values
value	String which corresponds with the numeric ASCII codes.

Example:

```
value = ASCIIToCharacters( 65 ) ' Returns "A"
value = ASCIIToCharacters( 97 ) ' Returns "a"
value = Chr( "65" )           ' Returns "A"
value = Chr( 72, 97, 116 )    ' Returns "Hat"
```

Results:

See example comments.

Links:

[ASCII](#)

See Also:

[Data Groups and Types](#)

FindString

Returns the position of the first occurrence of one string within another.

Function returns `-1` if the string is not found. The search is case-sensitive.

Syntax:

```
value = FindString( searchString, targetString )
OR
value = Instr( searchString, targetString )      ' Short form usage:
Instr"
```

Parameter:	Description:
searchString	String being searched.
targetString	String which is being looked for in the search string.
value	Position of found targetString, or -1 when not found. First searchString character is in position 1, not 0.

Example:

```
value = FindString( "Hello", "o" )      ' Returns 5
value = FindString( "Hello", "e" )      ' Returns 2
value = Instr( "Hello", "A" )           ' Returns -1
```

Results:

See example comments.

Links:**See Also:**

[Data Groups and Types](#)

FormatString

Formats the number or string using the format string.

Syntax:

```
formattedString = FormatString( stringToFormat, value1, [value2], [value3]
```

Parameter:	Description:
stringToFormat	String to be formatted " <code>%[flags] [width] [.precision]</code> " Use <code>%i</code> when formatting integers, <code>%f</code> when formatting floats, and <code>%s</code> when formatting strings.
value1	Integer, Floating or String.
[value2]	Optional: Integer, Floating or String.
[value3]	Optional: Floating only.
formattedString	the formatted string

Additional Arguments:

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each `%`-tag specified in the *format* parameter, if any. There should be the same number of these arguments as the number of `%`-tags that expect a value.

`%[flags][width][.precision][length]specifier`

Where specifier is the most significant one and defines the type and the interpretation of the value of the corresponding argument:

<i>specifier</i>	Output	Example
<code>c</code>	Character	<code>a</code>
<code>d</code> or <code>i</code>	Signed decimal integer	<code>392</code>
<code>e</code>	Scientific notation (mantise/ exponent) using e character	<code>3.9265e+2</code>
<code>E</code>	Scientific notation (mantise/ exponent) using E character	<code>3.9265E+2</code>
<code>f</code>	Decimal floating point	<code>392.65</code>
<code>g</code>	Use the shorter of <code>%e</code> or <code>%f</code>	<code>392.65</code>
<code>G</code>	Use the shorter of <code>%E</code> or <code>%f</code>	<code>392.65</code>
<code>o</code>	Signed octal	<code>610</code>
<code>s</code>	String of characters	<code>sample</code>
<code>u</code>	Unsigned decimal integer	<code>7235</code>

x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (capital letters)	7FA
p	Pointer address	B800:0000
n	Nothing printed. The argument must be a pointer to a signed int, where the number of characters written so far is stored.	
%	A % followed by another % character will write % to the string.	

The tag can also contain *flags*, *width*, *precision* and *modifiers sub-specifiers*, which are optional and follow these specifications:

<i>flags</i>	description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o , x or X specifiers the value is preceded with 0 , 0x or 0X respectively for values different than zero. Used with e , E and f , it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see <i>width</i> sub-specifier).

<i>width</i>	description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

<i>precision</i> <i>n</i>	description
<i>number</i>	<p>For integer specifiers (d, i, o, u, x, X): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0.</p> <p>For e, E and f specifiers: this is the number of digits to be printed <u>after</u> the decimal point.</p> <p>For g and G specifiers: This is the maximum number of significant digits to be printed.</p> <p>For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered.</p> <p>For c type: it has no effect.</p> <p>When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.</p>
<i>.*</i>	The <i>precision</i> is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

<i>length</i>	description
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s .
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

Use **%i** when formatting integers, **%f** when formatting floats, and **%s** when formatting strings.

Example:

```
PRINT FormatString( "I am %i years old.", 5 )
```

```
PRINT FormatString( "The price of an %s is $%.2f today.", "apple", 100.1 )
```

```
PRINT FormatString( "I'd like to say %s to %s.", "hello", "Sam" )
```

Results:

```
I am 5 years old.
The price of an apple is $100.12 today.
I'd like to say hello to Sam.
```

The following will create a string variable with tabSize spaces:

Example:

```
Variables: myPaddingString    Type: String

Variables: tabSize           Type: Integer
tabSize = 6
myPaddingString = FormatString( "% *s", tabSize, "" )

Print "myPaddingString = ", myPaddingString + ":"
Print "myPaddingString length = ", len(myPaddingString)
```

Results:

```
myPaddingString =      :
myPaddingString length = 6
```

The following will print the string "1234" padded within 10 leading spaces:

Example:

```
PRINT "|", FormatString( "% 10s", "1234" ) + "|"
```

Results:

```
|      1234|
```

These numbers will be right justified in columns:

Example:

```
PRINT "|", _
+ FormatString( "% 10s", AsString( 5.12 ) ) _
+ FormatString( "% 10s", AsString( 500 ) ) _
+ "|"
```

Results:

```
| 5.120000000      500|
```

You can also replace the hard coded padding with an integer variable like this:

Example:

```

VARIABLES: tabSize TYPE: integer
tabSize = 10

PRINT "|", _
+ FormatString( "% *s", tabSize, AsString( 5 ) ) _
+ FormatString( "% *s", tabSize, AsString( 5.123 ) ) _
+ "|"

PRINT "|", _
+ FormatString( "% *s", tabSize, AsString( 5.12 ) ) _
+ FormatString( "% *s", tabSize, AsString( 500 ) ) _
+ "|"

```

Results:

```

|          55.123000000|
| 5.120000000      500|

```

The following will do the above, but limit the decimals to 2 places:

Example:

```

VARIABLES: tabSize Type: INTEGER
VARIABLES: floatValue1, floatValue2 TYPE: FLOATING

tabSize = 10
floatValue1 = 5
floatValue2 = 5.123

PRINT "|", _
+ FormatString( "% *.2f", tabSize, floatValue1 ) _
+ FormatString( "% *.2f", tabSize, floatValue2 ) _
+ "|"

```

Results:

```

|          5.00      5.12|

```

The following will do the dynamic padding, and also dynamic number of decimals:

Example:

```

VARIABLES: tabSize, decimals TYPE: INTEGER
VARIABLES: floatValue1, floatValue2 TYPE: FLOATING

tabSize = 20
decimals = 2

floatValue1 = Random( 1000 ) / Random( 10 )
floatValue2 = Random( 100 ) / Random( 10 )

PRINT "|", _
+ FormatString( "% *.*f", tabSize, decimals, floatValue1 ) _
+ FormatString( "% *.*f", tabSize, decimals, floatValue2 ) _
+ "|"

```

Results:

| 106.83 10.00 |

Links:**See Also:**[Data Groups and Types](#)

GetField

This function will parse a comma delimited string and return the nth field.

Returns a string, which will be converted to a number if necessary.

This function is often used in conjunction with the file manager's [ReadLine](#) function to read comma delimited files.

Syntax:

```
returnString = GetField( stringValue, fieldIndex )
```

Parameter:	Description:
stringValue	comma delimited string
fieldIndex	number of the field to extract
returnString	string value of the extracted field

Note:

The **GetField** function replaced both the **GetStringField** and the **GetNumberField** functions from This new function will return a string or a number as necessary.

Example:

```
stringValue = "S,10,20,30,40"
```

```
returnString = GetField( stringValue, 1 ) ' Returns "S"
```

```
returnValue = GetField( stringValue, 3 ) ' Returns the number 20
```

Results:

See example comments.

Links:

[ReadLine](#), [RemoveCommasBetweenQuotes](#), [RemoveNonDigits](#).

See Also:

[Data Groups and Types](#)

GetFieldCount

This function returns the number of comma delimited values in a text string.

Often used for looping over the fields, using the [GetField](#) function.

Syntax:

```
count = GetFieldCount( stringValue )
```

Parameter:	Description:
stringValue	Comma delimited string
count	Number of comma delimited values

Example:

```
stringValue = "S,10,20,30,40"
```

```
count = GetFieldCount( stringValue ) ' Returns 5
```

Results:

See example comments.

Links:

[GetField](#)

See Also:

[Data Groups and Types](#)

GetFieldNumber

This function returns the field number of a string.

The reverse of the [GetField](#) process

Syntax:

```
fieldNumber = GetFieldNumber( csvString, stringToFind )
```

Parameter:	Description:
csvString	Comma delimited string
stringToFind	String to find in the csvString
fieldNumber	field number

Example:

```
csvString = "S,Hi,There,Blox"
```

```
fieldNumber = GetFieldNumber( csvString, "There" ) ' Returns 3
```

Results:

See example comments.

Links:

[GetField](#)

See Also:

[Data Groups and Types](#)

LowerCase

Converts any Upper-Case text characters to Lower-Case.

Syntax:

```
value = LowerCase( inputString )    ' Short form: "LCase"
```

Parameter:

inputString

Description:

String to lower case.

value

Lower-Case version of the input string.

Example:

```
value = LowerCase( "Hello" )        ' Returns "hello"  
value = LowerCase( "HELLO" )       ' Returns "hello"
```

Results:

See example comments.

Links:

[UCase](#)

See Also:

[Data Groups and Types](#)

LeftCharacters

Returns a specified number of characters from the left side of a string. If length is less than 1, the empty string is returned.

If length is greater than or equal to the number of characters in string, the entire string is returned.

Syntax:

```
value = LeftCharacters( inputString, length )
OR
value = Left( inputString, length )      ' Short form usage: "Left"
```

Parameter:	Description:
inputString	String from which to extract the left-side characters.
length	Number of characters on the right side of text to extract.
value	Extracted characters.

Example:

```
value = LeftCharacters( "Hello", 3 )      ' Returns "Hel"
value = Left( "Hello", 2 )                ' Returns "He"
value = Left( "Hello", 40 )               ' Returns "Hello"
```

Results:

See example comments.

Links:

[MiddleCharacters](#), [RightCharacters](#)

See Also:

[Data Groups and Types](#)

MiddleCharacters

Returns a specified number of characters from the middle of a string.

Syntax:

```
value = MiddleCharacters( inputString, start, length )
OR
value = Mid( inputString, start, length )      ' Short form usage:
"Mid"
```

Parameter:	Description:
inputString	String from which to copy the characters
start	Starting character position from which to start the character copying First character in the text is considered character 1.
length	Number of characters to copy.
value	Copied characters.

Example:

```
value = MiddleCharacters( "Hello", 2, 3 ) ' Returns "ell"
value = MiddleCharacters( "Hello", 3, 2 ) ' Returns "ll"
```

Results:

See example comments.

Links:

[LeftCharacters](#), [RightCharacters](#)

See Also:

[Data Groups and Types](#)

RemoveCommasBetweenQuotes

Removes the commas that are between quotes from an imported file record string.

Often output from other systems and/or applications will put quotes around numbers with commas, in comma delimited files, so the number stays together in the same field. When that happens this function is what is needed to correct that problem. In use this happens when importing data from a file where the numbers might have been copied from a report and were formatted with comma so make reading their value easier. Not removing the comma from a comma separated list of values will cause the number with the comma to be view as two different values, instead of a single larger value.

For example, a comma separated string value read in from a file could have fields that are encased in quotes, and within that they may have additional commas. In order to use the [GetField](#) function on this string, it is necessary to remove the commas from between the quotes so the field count returned from [GetFieldCount](#) is correct. This function will also remove the quotes, since they are not required anymore.

The [GetField](#) function will read each value as a string, and convert numbers to numbers if necessary.

See also the [RemoveNonDigits](#) function to remove \$ signs and other non digits from number strings.

Syntax:

```
RemoveCommasBetweenQuotes( inputString )
```

Parameter:	Description:
inputString	String from which commas are to be removed.

This is example uses a simple method to emulates the contents of what a file record might contain within some of its numeric fields.

Example:

```
VARIABLES: stringRecord  Type: String

' Fabricate what a file record might appear.
stringRecord = "10,20,30," + Chr(34) + "40,000" + Chr(34) + ",50,60"

' Show file record and how each of the fields will be interpreted.
PRINT "Original string from file:", stringRecord
PRINT "Field 4 is:", GetField( stringRecord, 4 ), _
      + " Field 5 is:", GetField( stringRecord, 5 )
PRINT

' Remove the comma from the field with a formatting comma.
stringRecord = RemoveCommasBetweenQuotes( stringRecord )

' Show how removed comma record appears and how the field
' problem has been avoided.
PRINT "New converted string: ", stringRecord
PRINT "Field 4 is:", GetField( stringRecord, 4 ), _
      + " Field 5 is:", GetField( stringRecord, 5 )
```

Results:

Original string from file: 10,20,30,"40,000",50,60

Field 4 is: "40 Field 5 is: 000"

New converted string: 10,20,30,40000,50,60

Field 4 is: 40000 Field 5 is: 50

Links:

[Chr](#), [GetField](#), [GetFieldCount](#), [RemoveNonDigits](#)

See Also:

[Data Groups and Types](#)

RemoveNonDigits

Removes any characters that are not numbers from the string.

Useful to remove \$ signs and other currency symbols from number strings.

Syntax:

```
RemoveNonDigits( inputString )
```

Parameter:	Description:
inputString	String from which to remove the non-number digits

Example:

```
stringValue = "$40000"  
RemoveNonDigits( stringValue )
```

Results:

stringValue is now without a "\$" is now like this: "40000"

Links:

[RemoveCommasBetweenQuotes](#), [GetField](#).

See Also:

[Data Groups and Types](#)

ReplaceString

Replaces text found in the search string with the text provided as a replacement text.

Useful to replace colons with commas, as an example, in the correlation matrix so you can use the [GetField](#) function.

Syntax:

```
newString = ReplaceString( inputString, stringToReplace, stringToReplace
```

Parameter:	Description:
inputString	String where replacement will be found and replaced.
stringToReplace	Target text to find in the search string.
stringToReplaceWith	Replacement text that will be inserted into the place where the target string is removed.

Example:

```
PRINT ReplaceString( "Hello World", "Hello", "Gooby" )
```

Results:**Links:****See Also:**

[Data Groups and Types](#)

RightCharacters

Returns a specified number of characters from the right side of a string.

When character count is less than 1, the empty string is returned.

Should the length value be for a character count that is greater than, or equal to the number of characters in string, the entire character contents of the string is returned.

Syntax:

```

value = RightCharacters( inputString )
OR
value = Right( inputString )      ' Short form usage: "Right"

```

Parameter:	Description:
inputString	String from which to extract the right-side characters.
length	Number of characters on the right side of text to extract.
value	Extracted characters.

Example:

```

value = Right( "Hello", 3 )           ' Returns "llo"
value = Right( "Hello", 2 )           ' Returns "lo"
value = RightCharacters( "Hello", 5 ) ' Returns "Hello"

```

Results:

See example comments.

Links:

[LeftCharacters](#), [MiddleCharacters](#)

See Also:

[Data Groups and Types](#)

StringLength

Returns the number of characters in the a string text expression.

Syntax:

```
value = StringLength( inputString )
OR
value = Len( inputString )      ' Short form usage: "Len"
```

Parameter:	Description:
inputString	String being measured for its text character count.
value	Number of characters, including spaces in the string.

Example:

```
VARIABLES: inputString      TYPE: STRING

inputString = "Hello"
value = StringLength( inputString ) ' Returns 5

inputString = "Goodbye"
value = Len( inputString )          ' Returns 7

value = StringLength( "Hello" )    ' Returns 5
value = Len( "Goodbye" )           ' Returns 7
```

Results:

See example comments.

Links:**See Also:**

[Data Groups and Types](#)

TrimLeftSpaces

Returns a copy of the input string without the leading space characters.

Syntax:

```
value = TrimLeftSpaces( inputString )
```

OR

```
value = LTrim(inputString )           ' Short form usage: "LTrim".
```

Parameter:	Description:
inputString	String expression that needs the spaces on the left to be trimmed.
value	Same string expression minus the leading Space characters.

Example:

```
value = TrimLeftSpaces( "  Hello  " ) ' Returns "Hello  "
value = LTrim( "  Hello  " )         ' Returns "Hello  "
```

Results:

See example comments.

Links:

[TrimRightSpaces](#), [TrimSpaces](#)

See Also:

[Data Groups and Types](#)

TrimRightSpaces

Returns a copy of the input string without the trailing space characters.

Syntax:

```
value = TrimRightSpaces( inputString )
```

OR

```
value = RTrim(inputString )           ' Short form usage: "RTrim".
```

Parameter:	Description:
inputString	String expression that needs the spaces on the right to be trimmed.
value	Same string expression minus the trailing Space characters.

Example:

```
value = TrimRightSpaces( "   Hello   " ) ' Returns "   Hello"
value = RTrim( "   Hello   " )           ' Returns "   Hello"
```

Results:

See example comments.

Links:

[TrimLeftSpaces](#), [TrimSpaces](#)

See Also:

[Data Groups and Types](#)

TrimSpaces

Returns a copy of the input string without the leading and trailing spaces.

Syntax:

```
value = TrimSpaces (inputString )
```

OR

```
value = Trim(inputString )           ' Short form usage: "Trim".
```

Parameter:	Description:
inputString	String expression that needs the spaces on the right and left to be trimmed
value	Same string expression minus the leading and trailing Space characters.

Example:

```
value = TrimSpaces( "  Hello  " ) ' Returns "Hello"  
value = Trim( "  Hello  " )       ' Returns "Hello"
```

Results:

See example comments.

Links:

[TrimLeftSpaces](#), [TrimRightSpaces](#)

See Also:

[Data Groups and Types](#)

UpperCase

Returns text characters to Upper-Case in a string expression.

Syntax:

```
value = UpperCase( inputString )  
OR  
value = UCase( inputString )      ' Short form usage: "UCase"
```

Parameter:

inputString

Description:

Text or string expression with lower case characters

value

Lower Case characters converted to Upper Case characters.

Example:

```
value = UpperCase( "Hello" )      ' Returns "HELLO"  
value = UCase( "hello" )         ' Returns "HELLO"
```

Results:

Links:

See Also:

[Data Groups and Types](#)

9.7 Type Conversion Functions

"AS" functions convert a numeric expression to the **TYPE** class used in the function's name.

"IS" functions examine and report the truth of a **TYPE** class expression being the **TYPE** class used in the function's name.

TYPE:	Description:
AsFloating	Converts an passed expression to a TYPE FLOATING point numeric value.
AsInteger	Converts an passed expression to a TYPE INTEGER point numeric value.
AsSeries	Changes how a TYPE SERIES array is passed to a Chart Object function.
AsString	Converts an passed expression to a TYPE STRING a character text value.
IsFloating	Use this function when you need to be sure the expression is of TYPE FLOATING .
IsInteger	Use this function when you need to be sure the expression is of TYPE INTEGER .
IsString	Use this function when you need to be sure the expression is of TYPE STRING .

See Also:

[Data Groups and Types](#)

WWW.TRADING-SOFTWARE-DOWNLOAD.COM

AsFloating

Converts an passed expression to a **TYPE FLOATING** point numeric value.

Syntax:

```
value = AsFloating( expression )
```

Parameter:

expression

Description:

Expression to convert to a floating numeric value

value

Converted expression is returned as a TYPE Floating number.

Example:

```
VARIABLES: stringOne, stringTwo TYPE: STRING
```

```
stringOne = "123.456"
```

```
stringTwo = "456.789"
```

```
print stringOne + stringTwo
```

```
print AsFloating(stringOne) + AsFloating(stringTwo)
```

Results:

This prints the string "123.456456.789"

This prints the number 580.245

Links:

[AsInteger](#), [AsSeries](#), [AsString](#), [IsFloating](#), [IsInteger](#), [IsString](#)

See Also:

[Data Groups and Types](#)

AsInteger

Converts an passed expression to a **TYPE INTEGER** point numeric value.

Displays the given value as an integer. Function does not round a number with decimals, instead the value returned is the integer portion of the number.

Syntax:

```
AsInteger( AnyValue )
```

Parameter:

AnyValue

Description:

Any numeric value, Or expression that results in a numeric value.

Example:

```

' ~~~~~
' BPV Manual Series Test Values
' ~~~~~
dVal[1] = -1.50
dVal[2] = -1.00
dVal[3] = -0.50
dVal[4] = 0.00
dVal[5] = 0.50
dVal[6] = 1.00
dVal[7] = 1.50

' ~~~~~
PRINT "AsInteger Function:"
PRINT "-----"

FOR Ndx = 1 TO 7
  ' Floor Calculations
  PRINT "AsInteger(" + AsString(dVal[Ndx], 2) + ") = ", AsInteger( dVal[
Next ' Ndx

' ~~~~~

```

Results:

```

AsInteger Function:
-----
AsInteger(-1.50) = -1
AsInteger(-1.00) = -1
AsInteger(-0.50) = 0
AsInteger(0.00) = 0
AsInteger(0.50) = 0
AsInteger(1.00) = 1
AsInteger(1.50) = 1

```

Links:

[AsFloating](#), [AsSeries](#), [AsString](#), [IsFloating](#), [IsInteger](#), [IsString](#), [Ceiling](#), [Floor](#),

See Also:

[Data Groups and Types](#)

AsSeries

Changes how a **TYPE SERIES** array is passed to a [Chart Object](#) function.

Custom Chart Director series data parameter assignment methods requires that all data series passed to a chart parameter use this function.

AsSeries passes the location data's for the first element in a series instead of passing all the values in the series. Charting functions also require data passed to their function also include an element count.

This function is not required for static variables or properties that are not a series.

Syntax:

```
AsSeries( Any_BPV_Series )
```

Parameter:

Any_BPV_Series

Description:

Any BPV Series assigned to any Custom Chart parameter is required to use this conversion function.

Example:

```
' Add 5 element values to represent "bar1"
chart.AddBarSeries( AsSeries( bar1 ), 5 )
```

Links:

[Chart](#), [AsFloating](#), [AsInteger](#), [AsString](#), [IsFloating](#), [IsInteger](#), [IsString](#)

See Also:

[Data Groups and Types](#)

AsString

Converts an passed expression to a **TYPE STRING a** character text value.

Usually this is wanted when the results are to be used in a report where the number of decimal value must be controlled. It will also provide comma separation for large numbers.

Syntax:

```
stringVariable = AsString( expression, [ decimals ], [addCommas] )
```

Parameter:	Description:
expression	Expression to convert
[decimals]	Optional number of decimals to display
[addCommas]	Optional True/False to include commas in the number
stringVariable	Expression converted to a character string.

Example:

```

-----
VARIABLES: integerOne, integerTwo TYPE: INTEGER
integerOne = 123
integerTwo = 456

PRINT integerOne + integerTwo
PRINT AsString(integerOne) + AsString(integerTwo)

```

Results:

```

This prints "579"
This prints "123456"

```

```

-----
VARIABLES: floatVar TYPE: FLOATING
floatVar = 123.456789

PRINT AsString( floatVar, 2 )

```

Results:

```

This prints "123.45"

```

```

-----
VARIABLES: floatVar TYPE: FLOATING
floatVar = 123456.456789

PRINT AsString( floatVar, 2, true )

```

Results:

```

This prints "123,456.45"

```

Links:

[AsFloating](#), [AsInteger](#), [AsSeries](#), [IsFloating](#), [IsInteger](#), [IsString](#)

See Also:

IsFloating

Use this function when you need to be sure the expression is of **TYPE FLOATING**.

Function examines the expression numeric **TYPE** to determine if it is a Floating Point or decimal number.

Syntax:

```
value = IsFloating( expression )
```

Parameter:

expression

Description:

expression to check

value

Returns **True** when the expression is a floating value, or a **False** when it isn't.

Example:

```
VARIABLES: variableOne TYPE: STRING
variableOne = "ABC"

IF IsFloating(variableOne) THEN
    print variableOne, " is floating."
ELSE
    print variableOne, " is NOT floating."
ENDIF
```

Results:

This prints "ABC is NOT floating."

Links:

[AsFloating](#), [AsInteger](#), [AsSeries](#), [AsString](#), [IsInteger](#), [IsString](#)

See Also:

[Data Groups and Types](#)

IsInteger

Use this function when you need to be sure the expression is of **TYPE INTEGER**.

Syntax:

```
value = IsInteger( expression )
```

Parameter:	Description:
expression	Expression you need to check
value	Returns True when the expression is an Integer value

Example:

```
VARIABLES: VariableOne TYPE: STRING
```

```
VariableOne = "ABC"
```

```
IF ( IsInteger(variableOne) ) THEN
```

```
    print variableOne, " is an integer."
```

```
ELSE
```

```
    print variableOne, " is NOT an integer."
```

```
ENDIF
```

Results:

This prints "ABC is NOT an integer."

Links:

[AsFloating](#), [AsInteger](#), [AsSeries](#), [AsString](#), [IsFloating](#), [IsString](#)

See Also:

[Data Groups and Types](#)

IsString

Use this function when you need to be sure the expression is of **TYPE STRING**.

Syntax:

```
value = IsString( expression )
```

Parameter:

expression

Description:

expression to check

value

TRUE when the expression is of **TYPE STRING** value.

Example:

```
VARIABLES: variableOne    TYPE: STRING
variableOne = "ABC"

IF ( IsString(variableOne) ) THEN
    print variableOne, " is a string."
ELSE
    print variableOne, " is NOT a string."
ENDIF
```

Results:

This prints "ABC is a string."

Links:

[AsFloating](#), [AsInteger](#), [AsSeries](#), [AsString](#), [IsFloating](#), [IsInteger](#)

See Also:

[Data Groups and Types](#)

Section 10 – Indicator Reference

You can use a variety of indicators in a trading system. They can be used in your scripts, as well as graphed with the trading data.

Types of indicators	Description:
Basic Indicators	that use a built in formula. These are computed pre test.
Calculated Indicators	that take an expression. These can only use pre test static data as they are computed pre test.
Custom Indicators	that are computed during the test using the Update Indicators script.
Extended Indicators Indicator Pack 1	

1. To create a new Basic or Calculated indicator for your block, click on the Indicators item in the lower left panel of the Blox Editor and hit New.
2. To create a Calculated Indicator, select the Calculated type from the type drop down box. You can then enter an expression in the Indicator Value Expression box.
3. To create a Custom Indicator, create an Auto-Indexed [Instrument Permanent Variable](#) of type Series. Then assign this value in the [Update Indicators](#) script.

10.1 Basic Indicators

Indicators listed below are the names of the standard built-in indicators accessible from the Indicator section dialog shown in the [Creating Indicators](#) topic section.

Most of the indicators require Parameter values, but not all of them.

When selecting an indicator examine the fields in the dialog that are colored with a white background and enter, or change the value shown where necessary.

Indicator Names:	Descriptions:
Accumulation/Distribution	Volume-based momentum indicator
ADX - Average Directional Index	J. Welles Wilder's trend strength indicator
Average True Range	EMA of the True Range. Computes a simple moving average to start, then an exponential moving average.
Average True Range Simple	SMA of the True Range
Bar History Value	Historical Value of the indicator
Bar Plot Color	This is a special type of indicator that is only used by the SetSeriesColorStyle function to dynamically set the color of the bars on the trade chart. This indicator has no values and does not plot itself. Turn Plotting ON and Display OFF.
Bar Value	Value assigned to the indicator like "High + Low / 2"

Indicator Names:	Descriptions:
Bollinger Lower	Lower channel of the Bollinger band
Bollinger Upper	Upper channel of the Bollinger band. Uses a standard deviation based channel width from an ema. Both the StdDev and EMA are computed in the alternate fashion, with the bar 1 values as the prime value.
Calculated	See Calculated Indicators for more information
DI- - Negative Directional Indicator	J. Welles Wilder's Negative Directional Indicator
DI+ - Positive Directional Indicator	J. Welles Wilder's Positive Directional Indicator
EMA Alternate	Exponential Moving Average of the Value that primes with the Value itself on bar 1 and uses the EMA smoothing constant only.
Exponential Moving Average	Exponential Moving Average of the Value that primes with a simple moving average of the Value, and then uses the EMA smoothing constant thereafter.
Highest Value	Highest Value for N bars
Keltner Lower	Lower channel of the Keltner Band.
Keltner Upper	Upper channel of the Keltner Band. Uses an atr based channel width from an EMA. Both the ATR and EMA are computed in the alternate fashion, with the bar 1 values as the prime value.
Lowest Value	Lowest Value for N bars
MACD - MA Convergence Divergence	MACD for the value. The short moving average minus the long moving average. Uses the simple moving average to prime the short and long exponential moving averages.
MACD Alternate	Alternate version of the MACD, using the bar 1 value as the prime value rather than using a simple moving average.
Midpoint	Price that is the value between the current price, and the price at the offset price bar.
Range	Returns the range value of the highest-high to lowest-low from bar prices within the range.
Parabolic SAR	J. Welles Wilder's entry and exit indicator
RSI - Relative Strength Index	J. Welles Wilder's movement momentum indicator
Simple Moving Average	Simple Moving Average of the Value
Standard Deviation	Standard Deviation of the value of n-Bars
Standard Deviation Log	Standard Deviation of the log of the ratio bar change over n-Bars
Stochastic Oscillator	%K Stochastic Value for n-Bars
Stochastic Oscillator Full	Smoothed Stochastic
Stochastic Oscillator Slow	%D Stochastic Value for n-Bars
Unknown	This appears at the top of the built-in indicator list, and it is a marker for when an optional add-on indicator is missing it will be obvious that indicator type is now no longer available.



10.2 Creating Indicators

To create an Indicator in the Blox Editor, select the Indicators item and right click or use the Items menu. This will bring up the new indicator dialog:

Name for Code

This is the name which will be used to access the indicator in a script. In this case we named our indicator `averageClose`. We can use this like a variable in our scripts, with or without indexing. You can use any name here that complies with the rules for creating variables.

Using the indicator name without indexing refers to the most recent available data, today. Example:

```
IF averageClose > averageClose[1] THEN
```

OR

```
IF instrument.averageClose > instrument.averageClose[1] THEN
```

Type

Indicators listed below are the names of the standard built-in indicators accessible from the Indicator section dialog shown above.

Most of the indicators require [Parameter](#) values, but not all of them. When selecting an indicator examine the fields in the dialog that are colored with a white background and enter or change the value shown where necessary.

Available Indicators:[Available Indicators](#)**Parameter Values:**

The Value is the basis for the computation of the Indicator. The choices are:

Price Field:	Description:
Open	the open for the bar
High	the high for the bar
Low	the low for the bar
Close	the close of the bar
High + Low / 2	the average of the high and low
High + Low + Close / 3	the average of the high, low, and close
OHLC / 4	the average of the open, high, low, and close
Volume	the volume for the bar
Open Interest	the open interest for the bar (futures only)
Unadjusted Close	the unadjusted close for the bar
Extra Data 1	the value of the extra data 1 field for the bar
Extra Data 2	the value of the extra data 2 field for the bar
Not Applicable	Used for cases like the Average True Range, which do not require a value input

For example, a "Simple Moving Average" indicator that used a value of "High" would be a simple moving average of the instrument's high.

Time Frame

Reserved for future use.

Parameters

Most types of indicators require numeric constants for their computation. For instance, the MACD indicator requires the days for the long and short moving averages. You can select from a list of [Parameters](#) that you have created, or you can choose "Enter Value" and enter a constant value in the box to the right.

For many indicators, there is a final option called "Smoothing." This option will smooth the indicator by the bars indicated, using the EMA formula. If you enter 1 for the number of bars to smooth, there will be no smoothing.

Example: To create an RSI indicator with a smoothed signal RSI line, create two indicators, one with smoothing, and one without.

Scope

Set the scope based on which blocks and scripts need access to this indicator. If you set Scope to Block (default value), only the scripts in the same block will have access to the indicator. If you set to Scope to System, then all scripts and blocks in the system will have access to the indicator. When System Scoped indicators are shared across a system, an IPV Series variable using the same name as the indicator must be declared in the other blox. In addition, you should enable the

option "Defined Externally in another Block" so the same name IPV will be linked to the indicator in a different blox by telling the script parser the IPV variable is declared and defined elsewhere.

Plots on Trade Graph

Check to have the indicator plotted on the trade chart. When you plot an indicator on the trade graph, you can select the Display Name, the Color, and whether the indicator should be offset by one day.

Displays on Trade Graph

Check to have the indicators value displayed in the right panel of the trade chart, as the cross hairs are moved by the mouse or cursor. By clicking on the indicator name on the trade chart, plotting can be dynamically enabled or disabled. The indicator can also be removed from the chart.

Offset Plot by One Day

This option shifts the indicator ahead one day. This is useful when your indicator is used for stop or limit orders, and you want the indicator to visually cross the bar as an indication your order was hit. This is only a visual change on the graph, and does not change the calculations or results.

Graph Area

The text in this field will determine where the indicator is plotted. If you select "Price Chart" the value will be plotted on the price chart area. If you select any other text value, it will create a new chart and put the indicator there. You can have multiple indicators on the same chart area.

If the values of the indicator are not within the range of values shown for the instrument price bars, the indicator will not appear because its value will be out of range. If this absence is only occasional, assigning its Graph Area to the Price Chart will be useful. If it is most of the time, it will be best to assign the indicator to its own Graph Area by entering a name different than Price Chart.

Graph Style

Select a graph style for the plot.

Notes on Priming

The maximum amount of bars required to prime this indicator plus one will be added to overall priming. If the indicator is a 10 day moving average, then the first day scripts will run is day 11. Overall priming is the maximum bars required for indicators plus one, plus the maximum lookback parameter plus one.

10.3 Calculated Indicators

Calculated Indicators

Calculated indicators are calculated before the test has begun to run so all their values are available during the test. See [Basic Indicators](#) for more information on the other controls and options in this dialog.

To create a calculated indicator, select the Calculated type from the type drop down box. Use this to create a simple expression based on indicators, parameters, or values. An example for the channel top from the ATR Channel Breakout system is:

Example:

```
closeAverageDays + ( channelWidth * averageTrueRange )
```

Test Computed Indicators:

This property is computed dynamically during the simulation run and can be used in scripting as needed. This property cannot be used in [Calculated Indicators](#) because all values for all calculated indicators are computed before the simulation starts running, and this property has not been computed yet.

Test computed indicators do not support look-back references. However, their calculated result from each instrument bar can be stored in an IPV series that will support look-back referencing if that process is coded into the blox.

You can use indexes of other indicators, and the current value of indicators that are declared above this one. Please be careful, as the syntax checker cannot fully verify your expression. An illegal expression will cause your test to return unexpected results.

You can access past values of other indicators, so to create an easy smoothing of the RSI:

Example:

```
( rsiIndicator[1] + rsiIndicator[2] + rsiIndicator[3] ) / 3
```

Scripted Calculated example from the Turtle system:

Edit Indicator

Name for Code:

Type:

Value:

Time Frame:

Smoothing:

Not Applicable:

Not Applicable:

Indicator Value Expression

```
entryBreakoutHigh + ( entryOffset * averageTrueRange )
```

Expression looks ok.

Scope:

Plots

Graph Title:

Plot Color:

Graph Area:

Graph Style:

Offset Plot Ahead One Bar

Valid items to use in the expression:

Parameters, other indicators, numbers, certain instrument properties that are available pre test. To use another calculated indicator in the expression of a calculated indicator, be sure that the other indicator is listed first so that the value is updated for the bar prior to being used.

Instrument object properties that are static prior to the test start can be used. Dynamic instrument object properties cannot be used. No other objects can be used.

10.4 Custom Indicators

To create a Custom Indicator, create a System Scoped Auto-Indexed [Instrument Permanent Variable](#) of type Series. Then assign this value in the [Update Indicators](#) script.

An example of a custom indicator might be the average close since trade entry. This value cannot be determined pre test, so it cannot be a calculated indicator and must be a custom indicator.

Create a new Auxiliary Block. Create a system scoped auto indexed IPV series variable. Set it to plot. Let's call it averageClose.

The screenshot shows the 'Instrument Permanent Variable' dialog box with the following settings:

- Name for Code: averageClose
- Name for Humans: Average Close since Entry
- Defined Externally in Another Block:
- Variable Type:
 - Integer - whole number values e.g. 1, -2, 400, 5, etc.
 - Floating Point - fractional number e.g. 2.5, 1.414, etc.
 - Price - fractional number in the range of prices
 - String - "Hello", "Goodbye", etc.
 - Series - a series or list of numbers
- Plotting Controls:
 - Plots on Trade Graph
 - Plot Color: [Teal Color]
 - Graph Area: Price Chart
 - Graph Style: Small Dot
 - Offset Plot by One Day
- Variable Options:
 - Default Value: 0.000000
 - Scope: Block
 - Auto-Index -- Uses [n] as Lookback from Current Day

Now in the **Update Indicators** script section set the created **averageClose** series value like this:

Example:

```
' We can only compute this value when we are in a position
IF instrument.position <> OUT THEN

  ' Compute the number of bars in the trade including the entry bar.
  bars = instrument.unitBarsSinceEntry[1] + 1

  ' Compute the average close over the last 'bars' number of bars
  averageClose = Average( instrument.close, bars )
ENDIF
```

This value will be set everyday of the test. It will be set at the start of the instrument bar, so it can be used as part of the order fill process, stop adjustment, risk adjustment, after trading day work as well as entry and exit signals for the next trading day.

10.5 Indicator Access

Access:

You can access Indicators through scripting two ways. **NOTE:** Indicators are READ ONLY. They can be used by scripts, but not changed.

A way to access the indicator so that it performs when the condition is TRUE, is to create a statement like this:

Example:

```
IF myIndicator = 5 THEN PRINT "It is 5"
```

Or you can access using the instrument object as follows:

Example:

```
IF instrument.myIndicator = 5 THEN PRINT "It is 5"  
IF sp500Index.myIndicator = 5 THEN PRINT "It is 5"
```

Using the instrument '.' syntax is equivalent to using the indicator directly.

You can access indicators of other instrument objects using instrument variables and the '.' syntax. For the following example assume that an instrument variable called "sp500Index" has been created and set to the data for the S&P 500 stock index.

Example:

```
' When both conditions are TRUE, Go Long on next OPEN  
IF sp500Index.shortMovingAverage > sp500Index.longMovingAverage AND  
   instrument.shortMovingAverage > instrument.longMovingAverage THEN  
  
   ' Go Long on the Open.  
   broker.EnterLongOnOpen( instrument.longMovingAverage )  
ENDIF
```

System Scoped Indicators:

To access indicators in other blox of your system, you can set them to System Scoped. In the other block, create an IPV Series variable and check Defined Externally in another Block.

Here is an example:

The screenshot shows a dialog box titled "Instrument Permanent Variable". It contains the following fields and options:

- "Name for Code" text box containing "averageClose".
- "Name for Humans" text box containing "Average Close since Entry".
- Two buttons: "OK" and "Cancel".
- A checked checkbox labeled "Defined Externally in Another Block".
- A "Variable Type" section with five radio button options:
 - Integer - whole number values e.g. 1, -2, 400, 5, etc.
 - Floating Point - fractional number e.g. 2.5, 1.414, etc.
 - Price - fractional number in the range of prices
 - String - "Hello", "Goodbye", etc.
 - Series - a series or list of numbers (This option is selected with a filled radio button).

Now you can use this `averageClose` variable in the block and the value will be consistent across the whole system.

Section 11 – Indicator Pack 1

All Indicators and Series Functions listed in this section are contained in the DLL files listed below:

DLL File Name:	Use with Trading Blox 4 & Installed Windows Bit-Size Version:
IndicatorPack1-32.dll IndicatorPack1-64.dll	32-Bit & 64-Bit version of Windows. Both can be the direct the Extension folder at the same time. Trading Blox will use the version that it needs and will ignore the other DLL.

Note:

Reference the Trading Blox User Help file's Getting Started topic that discusses "Installing and Running Trading Blox."

Description information about this indicators and series functions is available:

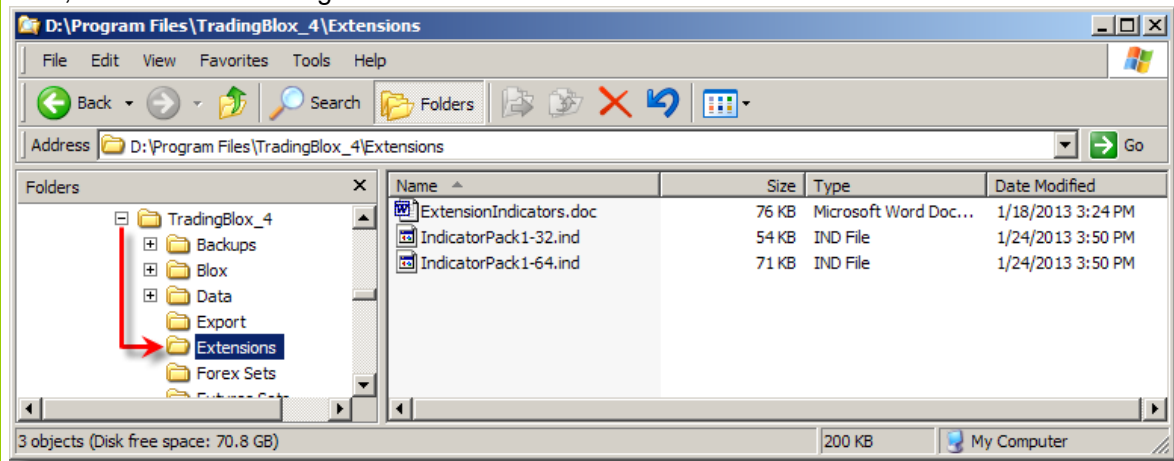
Indicators:

[Indicator Pack 1 Indicators](#)

Series Functions:

[Indicator Pack 1 Series Functions](#)

Indicator Pack extensions are placed in the Trading Blox Extension folder that is created when Trading Blox is installed. If you need to discover where the files are located or need to know where to place them, use this director image detail:



11.1 Indicator Pack 1 Indicators

All the Indicators listed in this topic are contained in both versions of this **Indicator Pak1** extension.

Indicators:	Description:
Average Trend Channel	Simple moving average that utilizes the moving average of the highs and lows to determine a channel. When prices are above the channel high the moving average is only allowed to increase and vice-versa. This mechanism can be used to reduce whipsaws when price trades rapidly above/below a simple moving average.
Chaikin Money Flow	<p>Chaikin Money Flow is a volume weighted average of Accumulation/ Distribution over the specified period. The principle behind the Chaikin Money Flow, is when the close is nearer to the high the more accumulation has taken place. Conversely the nearer the close is to the low, the more distribution has taken place.</p> <p>If the price action consistently closes above the bar's midpoint on increasing volume then the Chaikin Money Flow will be positive. Conversely, if the price action consistently closes below the bar's midpoint on increasing volume, then the Chaikin Money Flow will be a negative value.</p>
Commodity Channel Index	The Commodity Channel Index (CCI) is an oscillator that measures price variation from the statistical average. Depending on the period 70 to 80 percent of CCI values will fall in the range of -100 to 100.
Dema	<p>Double Exponential Moving Average is a lower lag moving average based on combining a single and double exponential moving average. The formula for Dema = $(2 * EMA(x, n) - EMA(EMA(x, n), n))$.</p> <p>The double exponential moving average was created by Patrick Mulloy and discussed in the January 1994 issue of Stocks and Commodities Magazine"</p>
Dominant Cycle	<p>"Estimated dominant cycle period of the market. The dominant cycle is the cycle period that has the most influence on prices. Based on Homodyne Discriminator algorithm described in "Rocket Science for Traders" by John Ehlers."</p> <p>Algorithm:</p> <ol style="list-style-type: none"> 1: Homodyne Discriminator Algorithm 2: Filter Bank Algorithm
Dominant Cycle Highest	Returns the highest high over the dominant cycle period. The dominant cycle period is calculated on the supplied time series (e.g Close)
Dominant Cycle Lowest	The lowest low of the dominant cycle period. The dominant cycle period is calculated on the supplied time series (e.g Close)
Dominant Cycle Phase	Provides the phase position from 0 to 360 degrees within the current dominant cycle period. The phase can be negative due to wraparound effects from 360 to 0. The phase can also be negative in a persistent downtrend. Based on phase algorithm described in "Rocket Science for Traders" by John Ehlers."
Ehlers Lead Sinewave	The Sin of the dominant cycle phase + 45 degrees. Combined with the Sinewave Indicator provides cycles turning points.

	The lead sinewave crosses the sinewave indicator 1/16 of a cycle before the turning point of the cycle is reached.
Ehlers Nonlinear Ma	Weighted moving average where the weights are dynamically calculated on each bar. Bars where the distance is largest from n-Bars ago will have the greatest weights.
Ehlers Sinewave	The sin of the dominant cycle phase. Combined with the lead Sinewave indicator provides cycles turning points.
Ehlers Zero Lag Ema	Adaptive exponential moving average. The moving average adapts based on the distance between the moving average and the current price. Described in November 2010 issue of Technical Analysis of Stocks and Commodities Magazine .
FAMA	Following Adaptive Moving Average is complementary moving average to MAMA . The moving average adapts half as fast as the MAMA moving average. Ehlers suggests a basic system can be developed based on the crossovers of MAMA and FAMA . Ehlers recommends values of 0.5 and 0.05 for FastLimit and SlowLimit arguments respectively.
Historic Volatility	Volatility measured as the standard deviation of close to close returns.
Instantaneous Trendline	Moving average taken over the dominant cycle. The moving average has the effect of removing the dominant cycle. As the dominant cycle can vary at each bar the effect is an adaptive moving average. The lag is half of the calculated dominant cycle.
Instantaneous Trendline Alternate	A lower lag version of the Instantaneous Trendline moving average. The standard Instantaneous Trendline has a lag of N/2 where N is the measured dominant cycle. The lag of the alternate calculation is considerably less and is only 8 bars when the dominant cycle is 40.
Kaufman Adaptive Moving Average	"Kaufman's Adaptive Moving Average is an adaptive moving average that uses the noise level of the market to determine the length of the trend required to calculate the average. The more noise in the market, the slower the trend used to calculate the average."
Keltner Channel	Keltner channel is a technical analysis indicator showing a central moving average line plus channel lines at a distance above and below.
Laguerre Moving Average	Adaptive moving average where low frequency (trend) parts of price are delayed much more than the high frequency components. The moving average will rapidly follow prices changes but will flatten out during consolidation periods.
MAMA	Mother of Adaptive Moving Averages (MAMA) is an adaptive exponential moving average created by John Ehlers. The moving average adapts to the rate of change of the phase of the dominant cycle. Ehlers recommends values of 0.5 and 0.05 for FastLimit and SlowLimit arguments respectively.
Momentum	Momentum is the difference between current price and the price a specified number of bars ago.
Money Flow Index	Money Flow Index measures the flow of money into and out of a security over the specified Period. Its calculation is similar to that of the Relative Strength Index (RSI), but takes volume into account in its calculation. The indicator is calculated by accumulating positive and negative money flow values, then creating a ratio of the two values. The final ratio is then scaled to fall between 0 – 100.

On Balance Volume	On Balance Volume is a cumulative indicator that uses volume to gauge the strength of a market. If prices close up, the current bar's volume is added to OBV, and if prices close down, it is subtracted.
Percent R	Percent R (%R) is a momentum indicator developed by Larry Williams. Like the Stochastic Oscillator, %R is used to gauge overbought and oversold levels, and ranges between 0 and 100.
Percent Rank	Calculates the percentile ranking of a value in a price series.
Percentile	Returns a price value estimate of the specified percentile level
Range	High minus Low of the current bar
Rate of Change	The Rate of Change (ROC) indicator provides a percentage that the security's price has changed over the specified period.
Simons Historic Volatility	Historic volatility measurement that incorporates the high, low, and gaps as well as close to close returns. Described in " The Dynamic Option Selection System " by Howard Simons."
TEMA	<p>Triple Exponential Moving Average is a unique combination of simple exponential moving average, double exponential moving average and a triple exponential moving average.</p> $\text{Tema} = (3 * \text{EMA} - 3 * \text{EMA}(\text{EMA})) + \text{EMA}(\text{EMA}(\text{EMA}))$ <p>The TEMA, or Triple Exponential Moving Average, was introduced by Patrick Mulloy in Technical Analysis of Stocks & Commodities Magazine, February 1994.</p>
Trend Vigor	<p>The strength of the trend as measured over the dominant cycle. Trend Vigor is defined as the momentum over the dominant cycle period divided by the amplitude of the dominant cycle.</p> <p>Ehlers suggests trend trades should only be taken when the trend vigor is $> +1$ for longs and < -1 for shorts.</p>
True High	The maximum of the current high and the previous close.
True Low	The minimum of the current low and the previous close.
True Range	Equal to the "True High" minus the "True Low"
Weighted Moving Average	Linearly Weighted Moving Average (WMA). The WMA applies more weight to recent data and less weight to older elements.
ZScore	ZScore is a statistical function that indicates the number of standard deviation an item is above or below the average.

Average Trend Channel

Simple moving average that utilizes the moving average of the highs and lows to determine a channel. When prices are above the channel high the moving average is only allowed to increase and vice-versa. This mechanism can be used to reduce whipsaws when price trades rapidly above/below a simple moving average.

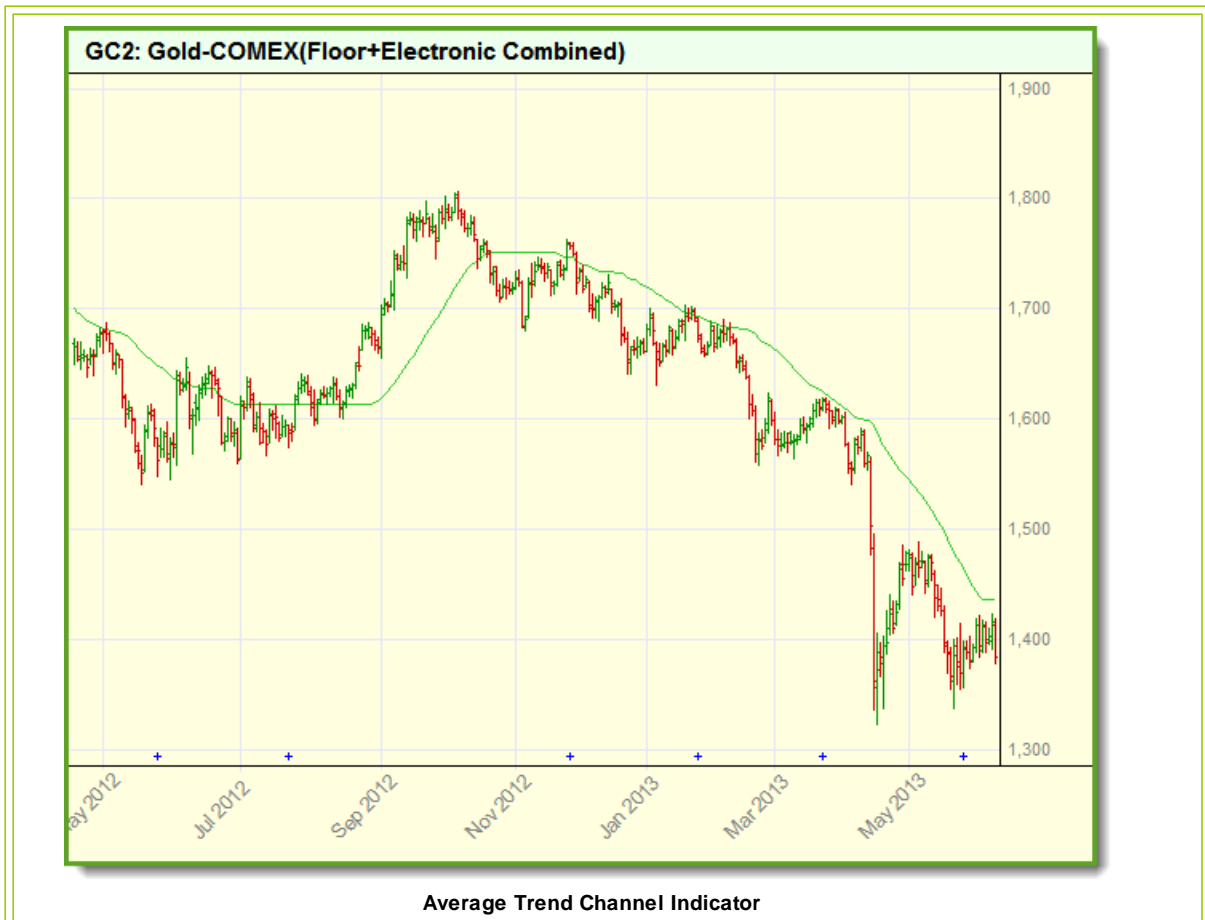
Parameter:	Description:
TrendChannelBars	Number of bars used in determining the average trend channel results.

Setup Example (Click on Images to enlarge):

The screenshot shows the 'Edit Indicator' dialog box for the 'Average Trend Channel' indicator. The 'Name for Code' is 'AvgTrendChannel'. The 'Type' is 'Average Trend Channel'. The 'Value' is 'OHLC / 4'. The 'Time Frame' is 'Bar'. The 'Period' is 'TrendChannelBars'. The 'Indicator Value Expression' field is empty, and a message below it says 'The parser likes the expression.' The 'Scope' is set to 'Block'. The 'Plots' section is checked, and the 'Display Value' checkbox is also checked. The 'Graph Title' is 'AvgTrendChannel', the 'Plot Color' is green, the 'Graph Area' is 'Price Chart', and the 'Graph Style' is 'Thin Line'. The 'Offset Plot Ahead One Bar' checkbox is unchecked.

Average Trend Channel setup dialog

Chart Example (Click on image to enlarge):



Links:

See Also:

Chaiken Money Flow

Indicator calculates and indexed value based on the price and volume for the number of bars that are being specified in the input length.

The Chaikin Money Flow is used to determine when a stock is being accumulated or distributed. It makes this determination by comparing the closing price to the high-low range value of the instrument on the same price bar. and then comparing the sum volume to the closing price and the daily peaks. It does not define the number of instrument issues being purchased and sold. Chaikin uses both the price and the volume over a 21-day calculation price-bar period to get the sum of the accumulation/distribution volume numbers and then divides the volume difference by the sum of the volume for the same price-bar period.

Chaikin Money Flow indicator theory tells us accumulation is happening when a stock with a volume increase has a Close price near the high of the market. Distribution happens with an increase in volume when the stock closes near the low of the price-bar.

Indicator values above 0, indicates accumulation, and values below 0 indicates distribution is in effect. Money Flow values values above +0.25 or below -0.25 indicates the bullish or bearish trends are strong and winning positions can add units on minor corrections.

Divergences may show up in the indicator has an increasing oscillator value while the price action makes a lower low informing us that there is less selling pressure pulling the security's price lower.

Parameter:	Description:
ChaikenCalcBars	Number of price bars over which to calculate the accumulation/distribution ratio.

Setup Example (Click on Images to enlarge):

Edit Indicator

Name for Code:

Type:

Value:

Time Frame:

Period:

Not Applicable:

Not Applicable:

Indicator Value Expression

The parser likes the expression.

Scope:

Plots

Display Value

Graph Title:

Plot Color:

Graph Area:

Graph Style:

Offset Plot Ahead One B:

OK Cancel

Chaikin Money Flow Oscillator setup dialog

The screenshot shows the 'Edit Indicator' dialog box for the 'ChaikenZero' indicator. The dialog is titled 'Edit Indicator' and contains the following fields and options:

- Name for Code:** ChaikenZero
- Type:** Calculated
- Value:** (empty)
- Time Frame:** Bar
- Smoothing:** Enter Value, with a value of 1 entered in the adjacent text box.
- Not Applicable:** (empty)
- Indicator Value Expression:** 0
- Expression looks ok:** (checked)
- Scope:** Block
- Plots:** Plots
- Display Value:** Display Value
- Graph Title:** Chaiken Zero
- Plot Color:** (dark grey)
- Graph Area:** Chaikin Money Flow
- Graph Style:** Thin Line
- Offset Plot Ahead One Bar:** Offset Plot Ahead One Bar

Buttons for 'OK' and 'Cancel' are located in the top right corner.

Chaiken Money Flow Oscillator Zero-Line setup dialog

Chaiken Zero plotting line is an optional line that improves in aiding the visibility of the index value crossing from positive to negative.

Chart Example (Click on image to enlarge):



Links:

See Also:

Commodity Channel Index

The Commodity Channel Index (**CCI**) is an oscillator that measures price variation from the statistical average. Depending on the period 70 to 80 percent of **CCI** values will fall in the range of -100 to 100.

Parameter::	Description:
CCIChannelBars	Number of bars to use in calculating CCI.

Setup Example (Click on Images to enlarge):

The screenshot shows the 'Edit Indicator' dialog box for the Commodity Channel Index (CCI). The dialog is titled 'Edit Indicator' and contains the following fields and options:

- Name for Code:** CommodityChannelIndex
- Type:** Commodity Channel Index
- Value:** Close
- Time Frame:** Bar
- Period:** CCIChannelBars
- Not Applicable:** (Two empty dropdown menus)
- Indicator Value Expression:** (Empty text area)
- Graph Title:** CCI
- Plot Color:** (Red color swatch)
- Graph Area:** Commodity Channel Index
- Graph Style:** Thin Line
- Scope:** Block
- Display Value:** (Checked checkbox)
- Offset Plot Ahead One Bar:** (Unchecked checkbox)

Buttons for 'OK' and 'Cancel' are located in the top right corner.

Commodity Channel Index setup dialog

The screenshot shows the 'Edit Indicator' dialog box for the 'CCIZeroLine' indicator. The dialog is titled 'Edit Indicator' and has 'OK' and 'Cancel' buttons in the top right corner. The fields are as follows:

- Name for Code: CCIZeroLine
- Type: Calculated
- Value: (empty)
- Time Frame: Bar
- Smoothing: Enter Value, with a value of 1 entered in the adjacent text box.
- Not Applicable: (empty)
- Not Applicable: (empty)

The 'Indicator Value Expression' field contains the number '0'. Below this field, a message says 'Expression looks ok.' To the right of this message are two checkboxes: 'Plots' (checked) and 'Display Value' (unchecked). Below these are several configuration options:

- Graph Title: CCIZero
- Plot Color: (dark grey)
- Graph Area: Commodity Channel Index
- Graph Style: Thin Line
- Offset Plot Ahead One B: (unchecked)

The 'Scope' dropdown is set to 'Block'.

Commodity Channel Index Zeroline setup dialog

Zeroline plotting line is an optional line that improves in aiding the visibility of the index value crossing from positive to negative.

Chart Example (Click on image to enlarge):



Links:

See Also:


Kaufman Adaptive Moving Average

Kaufman's Adaptive Moving Average, KAMA, is an adaptive calculation that uses the noise level of the market to determine the length of the trend required to calculate the average. The more noise in the market, the slower the trend used to calculate the average.

On page 731 of Perry Kaufman's *New Trading Systems and Methods*, 4th edition, he describes how his adaptive trend calculations work, and his ideas on how they can be applied to trading ideas.

Parameter:	Description:
nBarsLen	Number of price bars over which to calculate the Kaufman averages.

Setup Example (Click on Images to enlarge):



The image shows a software dialog box titled "Edit Indicator" for configuring the "Kaufman Adaptive High" indicator. The dialog is organized into several sections:

- Basic Settings:** Includes fields for "Name for Code" (KAdaptiveHigh), "Type" (Kaufman Adaptive Moving Average), "Value" (High), "Time Frame" (Bar), "Period" (nBarsLen), and two "Not Applicable" dropdowns.
- Indicator Value Expression:** A large text area for defining the indicator's logic, currently empty.
- Display Options:** Includes a "Scope" dropdown set to "Block", a "The parser likes the expression." text box, and checkboxes for "Plots" and "Display Value" (both checked).
- Graphing Options:** Includes "Graph Title" (KAdaptiveHigh), "Plot Color" (green), "Graph Area" (Price Chart), "Graph Style" (Thin Line), and an unchecked "Offset Plot Ahead One Bar" checkbox.

Buttons for "OK" and "Cancel" are located in the top right corner.

Kaufman Adaptive High Dialog Settings

The screenshot shows the 'Edit Indicator' dialog box for the 'KAdaptiveLow' indicator. The dialog is titled 'Edit Indicator' and contains the following fields and options:

- Name for Code:** KAdaptiveLow
- Type:** Kaufman Adaptive Moving Average
- Value:** Low
- Time Frame:** Bar
- Period:** nBarsLen
- Not Applicable:** (two empty dropdown menus)
- Indicator Value Expression:** (empty text area)
- Scope:** Block
- Plots:** Plots
- Display Value:** Display Value
- Graph Title:** KAdaptiveLow
- Plot Color:** (brown color swatch)
- Graph Area:** Price Chart
- Graph Style:** Thin Line
- Offset Plot Ahead One Bar:** Offset Plot Ahead One Bar

Buttons for 'OK' and 'Cancel' are located in the top right corner.

Kaufman Adaptive Low Dialog Settings

Chart Example (Click on image to enlarge):



Links:

See Also:

Keltner Channel

Keltner channel is a technical analysis indicator showing a central moving average line plus channel lines at a distance above and below.

To create a Keltner Channel Trading Blox provides two indicators to create the upper and lower channel indicator calculations. It also provides the simple [Average](#) function to create the average price line created by the average of the High, Low and Close price values.

Example shows the period lengths for the Keltner upper and lower bands, Average True Range, and Center price average. All the calculations used the same parameter bar count value.

Indicator lines displayed above and below the center average price are drawn a distance from center indicator price using the a floating point parameter value to spread the channel lines.

Parameter:	Description:
KeltnerBars	Integer value of the number of price bars to use in the calculation of the moving average, and the number of bars to use in the calculation of the ATR Bars (Average True Range). It is also the parameter used in the calculation of the average price shown between the upper and lower channel bands.
KeltnerBandOffset	Decimal value used to expand the average price standard deviation distance for the upper and lower bands from the average price value.

Setup Example (Click on Images to enlarge):

The image shows a software dialog box titled "Edit Indicator". It contains several configuration options for an indicator:

- Name for Code:** KeltnerUpperBand
- Type:** Keltner Upper
- Value:** Close
- Time Frame:** Bar
- Moving Average Bars:** KeltnerBars
- ATR Bars:** KeltnerBars
- Channel Width:** KeltnerBandOffset

Below these settings is a text area for the "Indicator Value Expression" which is currently empty. Underneath this is a note: "The parser likes the expression." followed by a large empty text box.

At the bottom of the dialog, there are additional options:

- Scope:** Block
- Plots**
- Display Value**
- Graph Title:** Keltner Upper Band
- Plot Color:** Green
- Graph Area:** Price Chart
- Graph Style:** Thin Line
- Offset Plot Ahead One Bar**

Buttons for "OK" and "Cancel" are located in the top right corner.

Keltner Upper Channel setup dialog.

The image shows a software dialog box titled "Edit Indicator" for configuring a "Keltner Lower Channel". The dialog is organized into several sections:

- Top Section:** Contains fields for "Name for Code" (KeltnerLowerBand), "Type" (Keltner Lower), "Value" (Close), "Time Frame" (Bar), "Moving Average Bars" (KeltnerBars), "ATR Bars" (KeltnerBars), and "Channel Width" (KeltnerBandOffset). "OK" and "Cancel" buttons are on the right.
- Indicator Value Expression:** A large text area for defining the indicator's logic.
- Parser Note:** A smaller text area with the text "The parser likes the expression."
- Scope:** A dropdown menu set to "Block".
- Display Options:** Includes checkboxes for "Plots" and "Display Value" (both checked), a "Graph Title" field (Keltner Lower Band), a "Plot Color" dropdown (green), a "Graph Area" dropdown (Price Chart), a "Graph Style" dropdown (Thin Line), and an unchecked checkbox for "Offset Plot Ahead One B".

Keltner Lower Channel setup dialog.

Edit Indicator

Name for Code: KeltnerCenter

Type: Simple Moving Average

Value: High + Low + Close / 3

Time Frame: Bar

Moving Average Bars: KeltnerBars

Smoothing: Enter Value 1

Not Applicable: Not Applicable

Indicator Value Expression

Please enter an expression. The expression is the part of the equation to the right of the = sign, such as 'a+b'.

Scope: Block

Plots

Display Value

Graph Title: KeltnerCenter

Plot Color: [Blue]

Graph Area: Price Chart

Graph Style: Thin Line

Offset Plot Ahead One Bar

OK Cancel

Keltner Average Price setup dialog

Chart Example (Click on image to enlarge):



Links:

See Also:

Trend Vigor

The strength of the trend as measured over the dominant cycle. Trend Vigor is defined as the momentum over the dominant cycle period divided by the amplitude of the dominant cycle.

Ehlers suggests trend trades should only be taken when the trend vigor is $> +1$ for longs and < -1 for shorts.

Parameter::	Description:
TrendVigorBars	Number of price-bars to use in the Trend Vigor calculation.

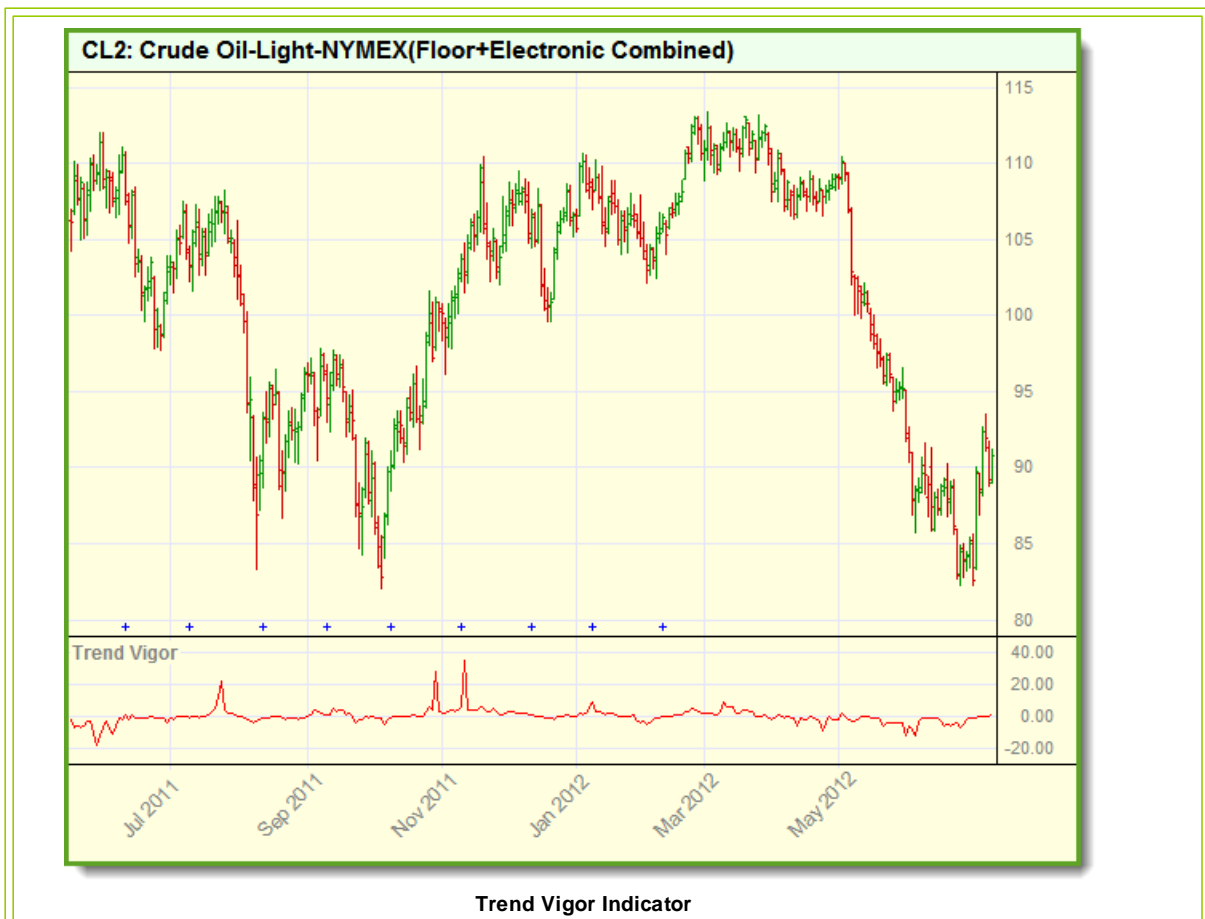
Setup Example (Click on Images to enlarge):

The screenshot shows the 'Edit Indicator' dialog box for the 'Trend Vigor' indicator. The settings are as follows:

- Name for Code: TrendVigor
- Type: Trend Vigor
- Value: Close
- Time Frame: Bar
- Algorithm: TrendVigorBars
- Not Applicable: (empty)
- Not Applicable: (empty)
- Indicator Value Expression: (empty)
- Scope: Block
- Plots: Plots
- Graph Title: TrendVigor
- Plot Color: Red
- Graph Area: Trend Vigor
- Graph Style: Thin Line
- Offset Plot Ahead One Bar: Offset Plot Ahead One Bar
- Display Value: Display Value

Trend Vigor Indicator setup dialog

Chart Example (Click on image to enlarge):



Links:

See Also:

11.2 Indicator Pack 1 Series Functions

All the Series Functions listed in this topic are contained in both version of **Indicator Pak1** extension.

Function Name:	Description:
<u>EhlersZeroLagEma</u>	Adaptive exponential moving average. The moving average adapts based on the distance between the moving average and the current price.
<u>InstantaneousTrendLine</u>	Adaptive moving average whose length is determined each bar by the dominantCycleSeries parameter.
<u>MarketNoise</u>	Calculates an estimate of the “noise” in the market over the specified number of bars. Noise is defined as the maximum absolute deviation from the momentum trend line. The momentum is calculated over the number of bars specified. This estimate can be helpful in placing stops outside the noise of the market.
<u>MedianAbsoluteDeviation</u>	Median Absolute Deviation (MAD) is a measure of variation used in a similar manner to the standard deviation. The MAD is more robust in the presence of outliers and does not require a gaussian distribution. In order to estimate the standard deviation for a gaussian distribution the MAD can be multiplied by 1.4826
<u>Momentum</u>	Momentum is the difference between current price and the price a specified number of bars ago.
<u>MRO</u>	Most Recent Occurrence returns the numbers of bars ago that the condition was true. If the condition was not true during the lookback the function returns the value -1.
<u>Percentile</u>	Returns a price value estimate of the specified percentile level.
<u>PercentRank</u>	Most Recent Occurrence returns the numbers of bars ago that the condition was true. If the condition was not true during the lookback the function returns the value -1.
<u>RateOfChange</u>	The Rate of Change (ROC) indicator provides a percentage that the security's price has changed over the specified period.
<u>SpearmanCorrelation</u>	Spearman correlation is an alternative to the Pearsons correlation and correlates based on the ranking of the series values.
<u>SpearmanCorrelationSync</u>	Spearman correlation is an alternative to the Pearsons correlation and correlates based on the ranking of the series values . Used for IPV Auto Indexed Series or Price Series and synchs the dates of the two series before computing the correlation.
<u>SpearmanLogCorrelation</u>	Spearman correlation is an alternative to the Pearsons correlation and correlates based on the ranking of the series values . This version computes the log returns before computing the correlation.
<u>SpearmanLogCorrelationSync</u>	Spearman correlation is an alternative to the Pearsons

	correlation and correlates based on the ranking of the series values . Used for IPV Auto Indexed Series or Price Series and synchs the dates of the two series before computing the correlation. This version computes the log returns before computing the correlation.
<u>WMA</u>	Weighted Moving Average calculation method applies more weight to recent data and less weight to older elements.
<u>ZScore</u>	ZScore is a statistical function that indicates the number of standard deviation an item is above or below the average.
<u>ValueChart</u>	Volatility adjusted overbought/oversold oscillator. Value chart levels between -4 and +4 are considered "fair value"; +4 to +8 moderately overbought; -4 to -8 moderately oversold. Levels above +8 are considered significantly overbought and below -8 significantly oversold.

EhlersZeroLagEma

Adaptive exponential moving average.

The moving average adapts based on the distance between the moving average and the current price.

Syntax:

```
EhlersZeroLagEma( series , emaseries , bars , lastValueOfSeries )
```

Parameter:	Description:
series	The name of the series
emaseries	Normal exponential moving average (ema) series that was previously calculated. This ema should use the same period and is used as a starting point for adapting the moving average.
bars	The number of bars over which to find the moving average
lastValueOfSeries	The previous value in the series

Returns:

The zero lag exponential moving average.

Example:

```

| ~~~~~
| Update Indicators Script
| ~~~~~
| EZLEma = IPV numeric auto-index series
| EmaClose = Built-In Indicator EMA Close - 21-bars
| Bar_Count = 21 - Parameter - Lookback enabled
| Ehler's Zero Lag Ema - EmaClose = built-in Ema of Close
EZLEma = EhlersZeroLagEma( Instrument.Close, _
                          EmaClose, _
                          Bar_Count, _
                          EZLEma[1])
| ~~~~~

```

Chart Display:**Links:**

[EMA](#)

See Also:

[Data Group and Types](#)

InstantaneousTrendLine

Adaptive moving average whose length is determined each bar by the dominantCycleSeries parameter.

Syntax:

```
InstantaneousTrendLine(series, dominantCycleSeries)
```

Parameter:

series

Description:

The input series to be averaged

dominantCycleSeries

Previous calculated series corresponding to the result of the dominant cycle basic indicator.

See basic indicator - Dominant Cycle documentation for more information.

Returns:

Adaptive moving average of the input series.

Example:**Links:****See Also:****MarketNoise**

Calculates an estimate of the “noise” in the market over the specified number of bars. Noise is defined as the maximum absolute deviation from the momentum trend line. The momentum is calculated over the number of bars specified. This estimate can be helpful in placing stops outside the noise of the market.

Syntax:

```
MarketNoise(series, bars)
```

Parameter:

series

Description:

The name of the series.

bars

The number of bars over which to find the noise.

Returns:

Estimate of market noise over the specified number of bars.

Example:

Links:

See Also:

MedianAbsoluteDeviation

Median Absolute Deviation (MAD) is a measure of variation used in a similar manner to the standard deviation. The MAD is more robust in the presence of outliers and does not require a gaussian distribution.

In order to estimate the standard deviation for a gaussian distribution the MAD can be multiplied by 1.4826

Syntax:

```
MedianAbsoluteDeviation( series, bars )
```

Parameter:

bars

Description:

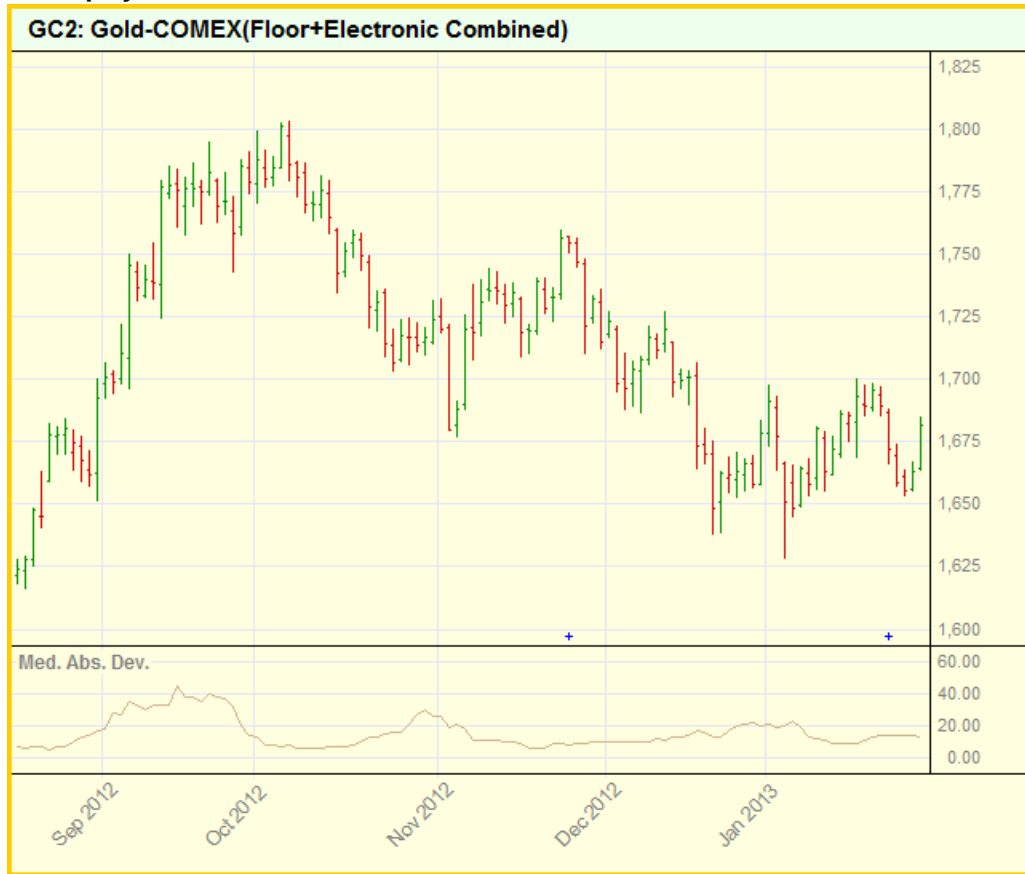
The number of bars over which to find the median absolute deviation

Returns:

The median absolute deviation over the specified number of bars.

Example:

```
' ~~~~~  
' Update Indicators Script  
' ~~~~~  
' MedAbsDev = IPV - Numeric Series  
' Bar_Count = 21  
MedAbsDev = MedianAbsoluteDeviation( Instrument.Close, Bar_Count )  
' ~~~~~
```

Chart Display:**Links:****See Also:**

Momentum

Momentum is the difference between current price and the price a specified number of bars ago.

Syntax:

```
Momentum( series, bars )
```

Parameter:

Description:

series	Names of the data series
bars	Number of bars over which to find the momentum value.

Returns:

Momentum value over the specified number of bars.

Example:

```

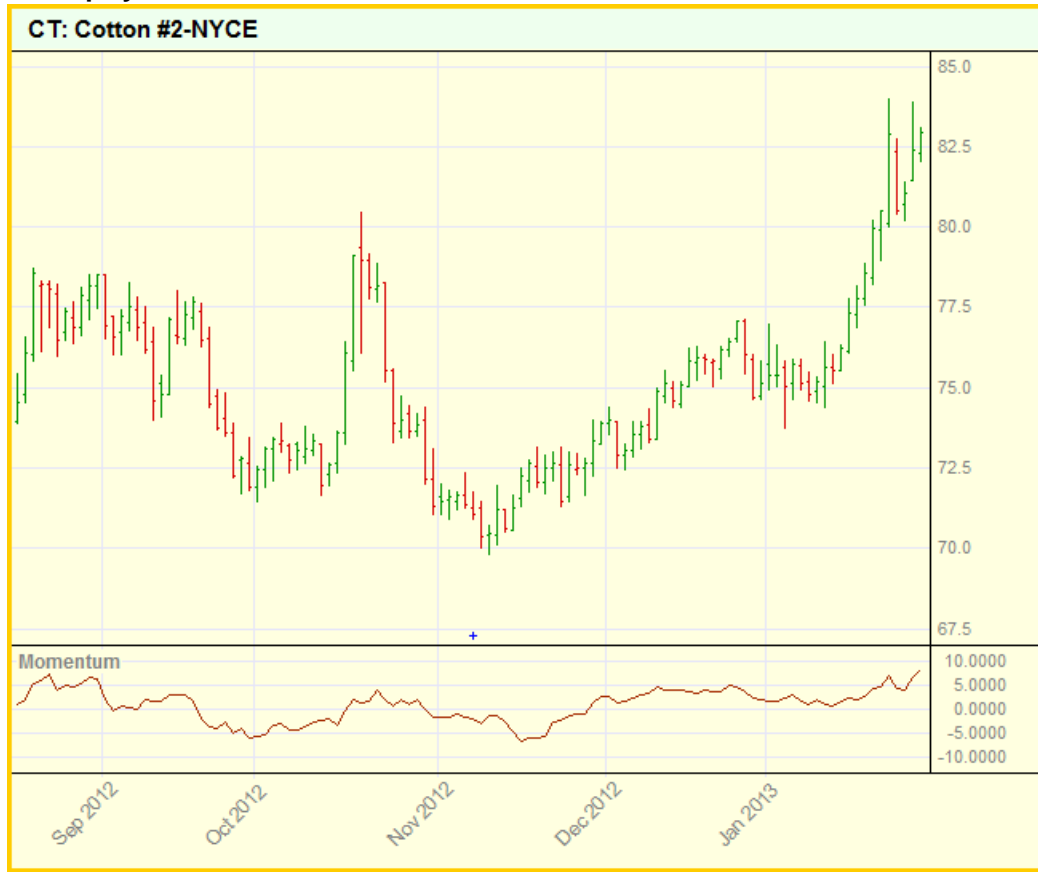
| ~~~~~
| Update Indicators Script
| ~~~~~
| Bar_Count = 21 - Parameter - Lookback enabled
| Difference in Close of Today and the Close[Bar_Count]
Mom_Close = Momentum( instrument.close, Bar_Count )
| ~~~~~

```

OR

The screenshot shows the 'Edit Indicator' dialog box with the following configuration:

- Name for Code: Close_Momentum
- Type: Momentum
- Value: Close
- Time Frame: Bar
- Lookback: Bar_Count
- Not Applicable: Not Applicable
- Not Applicable: Not Applicable

Chart Display:**Links:****See Also:**

MRO

Most Recent Occurrence returns the numbers of bars ago that the condition was true.

If the condition was not true during the lookback the function returns the value -1.

Syntax:

```
MRO( conditionseries, bars, instance )
```

Parameter:	Description:
conditionseries	Name of the condition series. The Condition series should use 1 for TRUE, and 0 for FALSE.
bars	The number of bars over which to find the instance of the condition.
instance	Which occurrence the condition to search for; for example, 1 = most recent, 2 = 2nd most recent.

Returns:

The number of bars ago that the condition was true or -1 if not found.

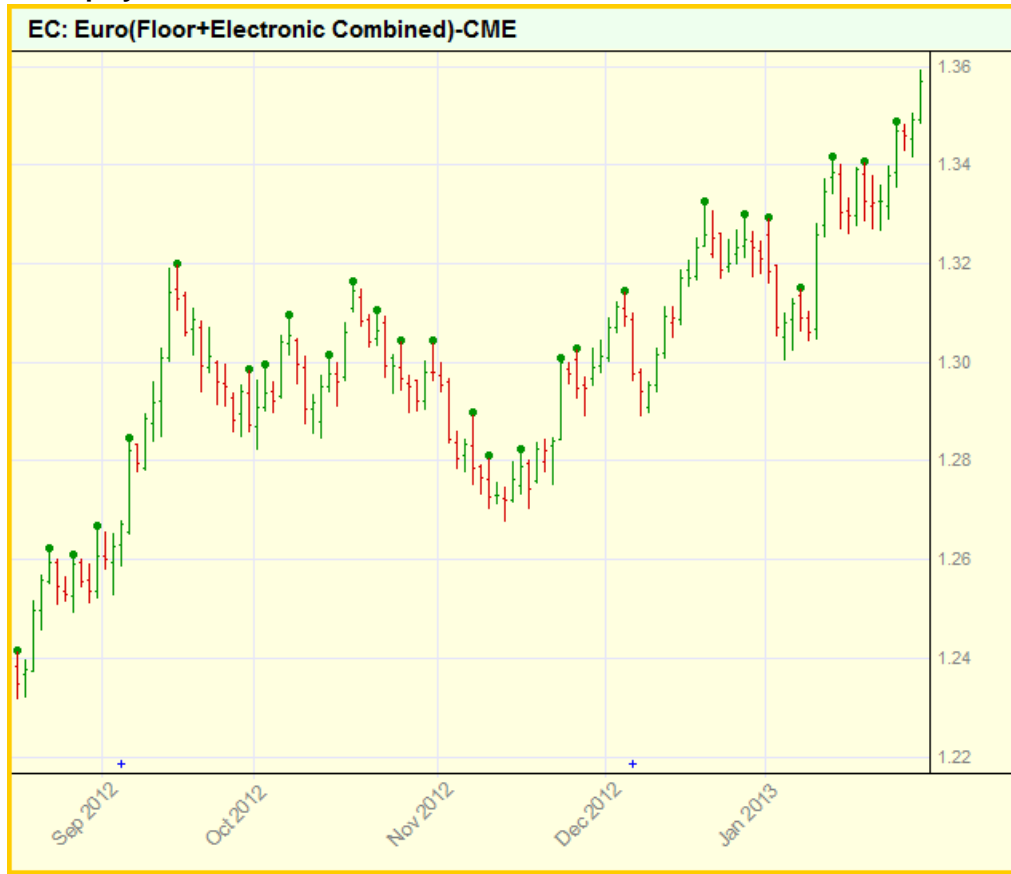
Example:

```
' ~~~~~
' Update Indicators Script
' ~~~~~
' HighPivot = IPV Series - Auto-Index - Default = False
' PlotHighPivot = IPV Series - Auto-Index - Default = Zero
'           Plot Dot on Pivot High
' Bar_Count = 21
' Instance = 1 = Most Recent

' Test for a simple High Pivot Bar Pattern
If instrument.high[2] < instrument.high[1] AND
instrument.high[1] > instrument.high[0] THEN
  ' Assign this element a True State
  HighPivot = TRUE
ENDIF

' Show High Pivot Locations
If MRO( HighPivot, Bar_Count, 1 ) = 1 THEN
  ' Plot Dot over Pivot High
  PlotHighPivot[2] = instrument.high[2] + (instrument.minimumTick * 4)
Else
  PlotHighPivot = Undefined
ENDIF

' ~~~~~
```

Chart Display:**Links:****See Also:**

Percentile

Returns a price value estimate of the specified percentile level.

Syntax:

```
Percentile(series, bars, percentileLevel)
```

Parameter:**Description:**

series	The name of the series.
bars	The number of bars over which to find the percentile
percentileLevel	The percentage level used to find the percentile. Specified as a value between 0 and 100, representing 0-100%.

Returns:

Percentile estimate.

Example:**Links:****See Also:**

PercentRank

Calculates the percentile ranking of a series value within the range of elements specified for the price series.

Syntax:

```
PercentRank ( series, bars )
```

Parameter:	Description:
series	Name of numeric series
bars	Count of the number of series elements over which to find the percentile.

Returns:

The percentile ranking.

Example:

```

| ~~~~~
| Update Indicators Script
| ~~~~~
| Bar_Count = 21 - Parameter - Lookback enabled
| Calculates percentile ranking of a value in a price series period
Percent_Rank_Close = PercentRank( instrument.close, Bar_Count )
| ~~~~~

```

OR

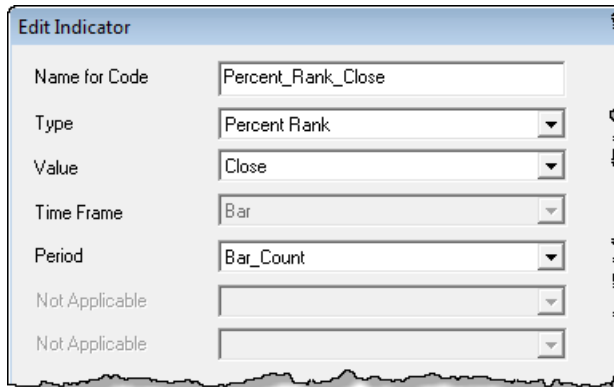
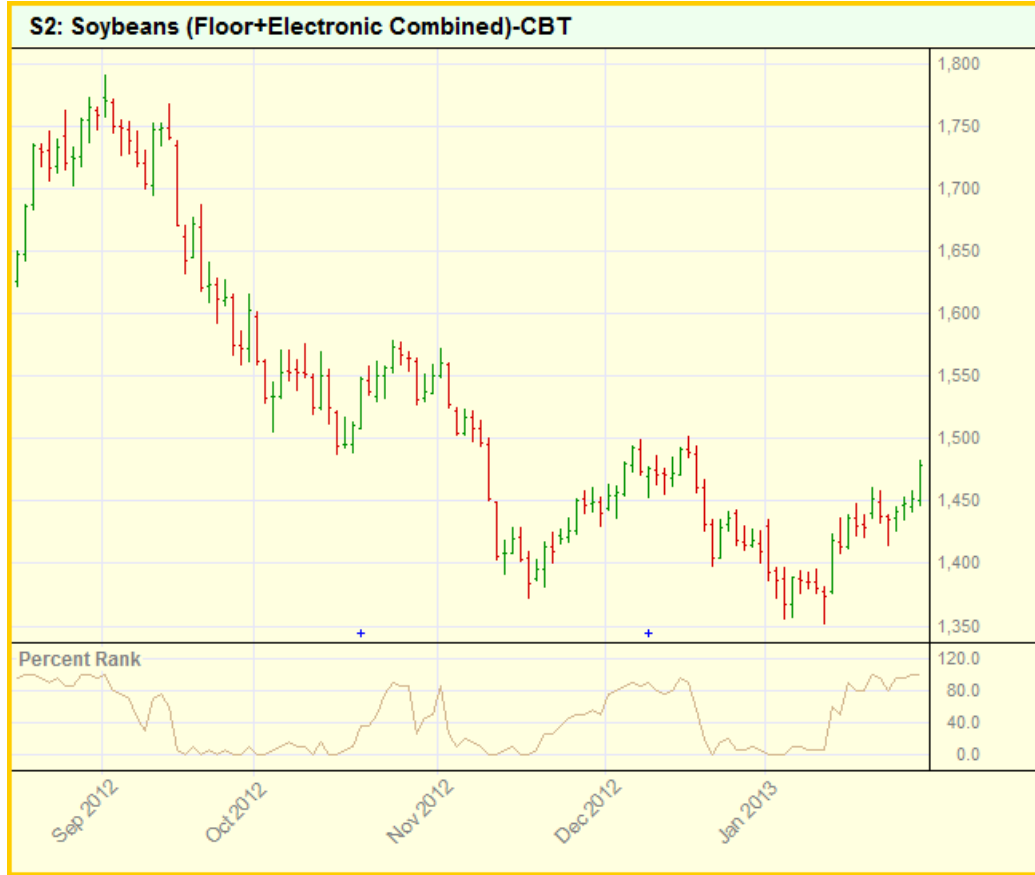


Chart Display:



Links:

See Also:

RateOfChange

The Rate of Change (ROC) provides a percentage that the instrument's price has changed over the specified period.

Syntax:

```
RateOfChange ( series, bars )
```

Parameter:	Description:
series	Name of data series.
bars	The number of bars over which to find the rate of change

Returns:

The rate of change over the specified number of bars.

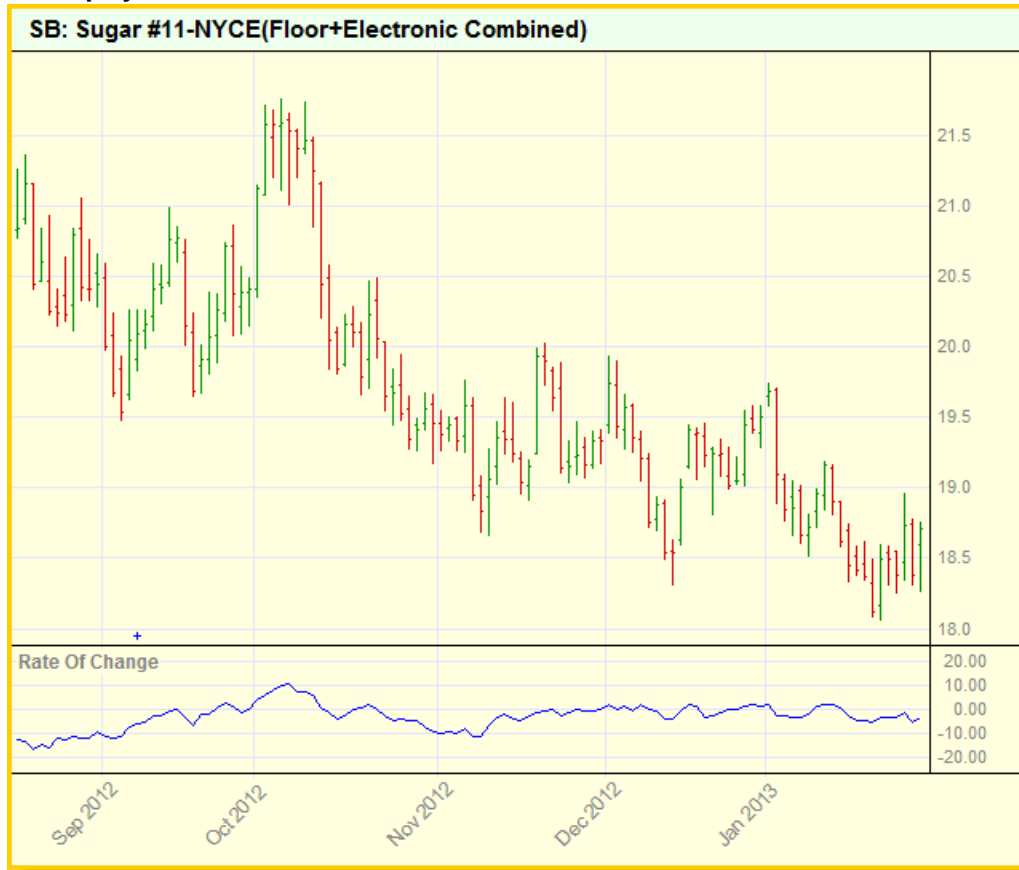
Example:

```
' ~~~~~
' Update Indicators Script
' ~~~~~
' Bar_Count = 21 - Parameter - Lookback enabled
' Calculates Rate of Price over price period length.
ROC_Close = RateOfChange ( instrument.close, Bar_Count )
' ~~~~~
```

OR

The screenshot shows the 'Edit Indicator' dialog box with the following settings:

- Name for Code: ROC_Close
- Type: Rate of Change
- Value: Close
- Time Frame: Bar
- Lookback: Bar_Count
- Not Applicable: (empty)
- Not Applicable: (empty)

Chart Display:**Links:****See Also:**

SpearmanCorrelation

Spearman correlation is an alternative to the Pearsons correlation and correlates based on the ranking of the series values.

Syntax:

```
SpearmanCorrelation( series1, series2, bars )
```

Parameter:**Description:**

series1	Name of first data series
series2	Name of second data series
bars	Number of bars over which to find the correlation.

Returns:

The correlation over the specified number of bars.

Example:**Links:****See Also:**

SpearmanCorrelationSync

Spearman correlation is an alternative to the Pearsons correlation and correlates based on the ranking of the series values . Used for IPV Auto Indexed Series or Price Series and synchs the dates of the two series before computing the correlation.

Syntax:

```
SpearmanCorrelationSync( series1, series2, bars )
```

Parameter:	Description:
series1	Name of first data series
series2	Name of second data series
bars	Number of bars over which to find the correlation.

Returns:

The correlation over the specified number of bars.

Example:**Links:****See Also:**

SpearmanLogCorrelation

Spearman correlation is an alternative to the Pearsons correlation and correlates based on the ranking of the series values . This version computes the log returns before computing the correlation.

Syntax:

```
SpearmanLogCorrelation( series1, series2, bars )
```

Parameter:**Description:**

series1	Name of first data series
series2	Name of second data series
bars	Number of bars over which to find the correlation.

Returns:

The correlation over the specified number of bars.

Example:**Links:****See Also:**

SpearmanLogCorrelationSync

Spearman Correlation is an alternative to the Pearsons Correlation and correlates based on the ranking of the series values . Used for IPV Auto Indexed Series or Price Series and synchs the dates of the two series before computing the Correlation. This version computes the Log returns before computing the Correlation.

Syntax:

```
SpearmanLogCorrelationSync( series1, series2, bars )
```

Parameter:	Description:
series1	Name of first data series
series2	Name of second data series
bars	Number of bars over which to find the correlation.

Returns:

The correlation over the specified number of bars.

Example:**Links:****See Also:**

WMA - Weighted M-Avg.

The WMA applies more weight to recent data and less weight to older elements.

Syntax:

```
WMA( series, bars )
```

Parameter:	Description:
series	Name numeric series
bars	Count of the number of series elements bars over which to find the WMA value

Returns:

The weighted moving average.

Example:

```

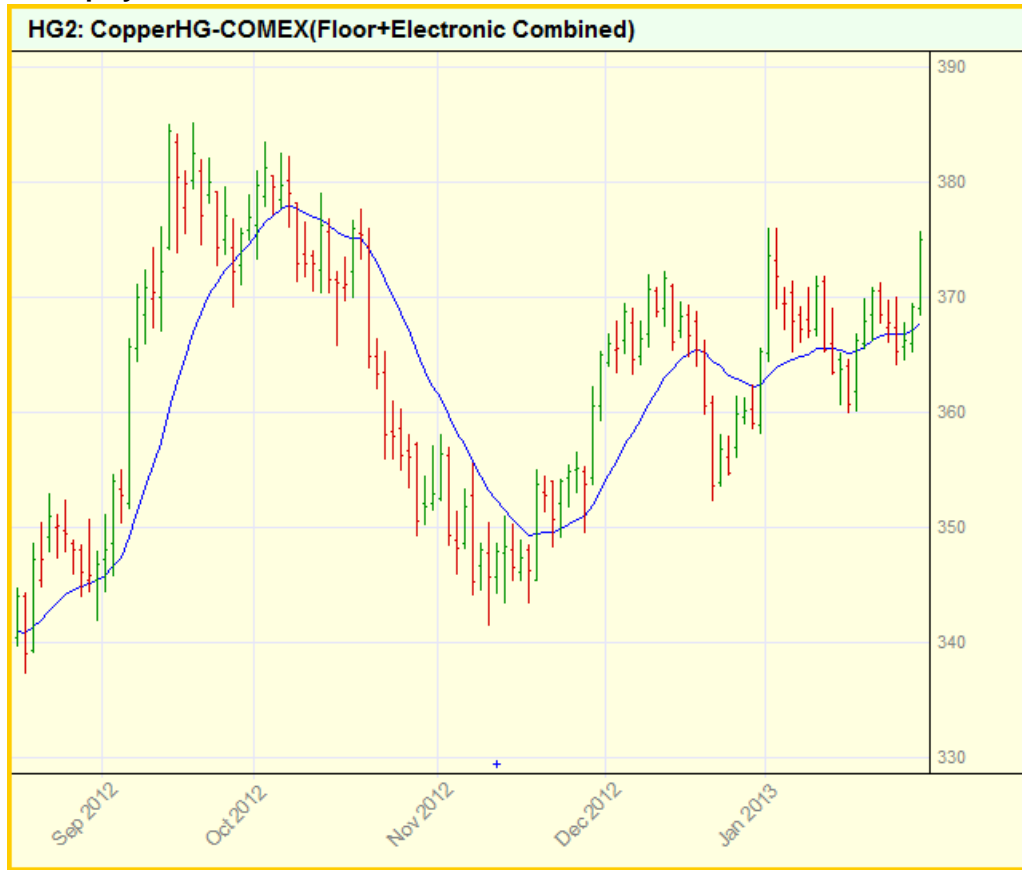
| ~~~~~
| Update Indicators Script
| ~~~~~
| Bar_Count = 21 - Parameter - Lookback enabled
| Calculates Weigthed Moving Avg over price period length.
WMA_Close = WMA( instrument.close, Bar_Count )
| ~~~~~

```

OR

The screenshot shows the 'Edit Indicator' dialog box with the following settings:

- Name for Code: WMA_Close
- Type: Weighted Moving Average
- Value: Close
- Time Frame: Bar
- Period: Bar_Count
- Not Applicable: (empty)
- Not Applicable: (empty)

Chart Display:**Links:****See Also:**

Z-Score

ZScore is a statistical function that indicates the number of standard deviation an item is above or below the average.

Syntax:

```
ZScore ( series, bars )
```

Parameter:	Description:
series	Name of data series
bars	Number of bars over which to find the Z-Score value.

Returns:

Z-score value.

Example:

```

| ~~~~~
| Update Indicators Script
| ~~~~~
| Bar_Count = 21 - Parameter - Lookback enabled
| Calculates Number of Std. Deviation Above & Below Averag
ZScoreClose = ZScore( instrument.close, Bar_Count )
| ~~~~~

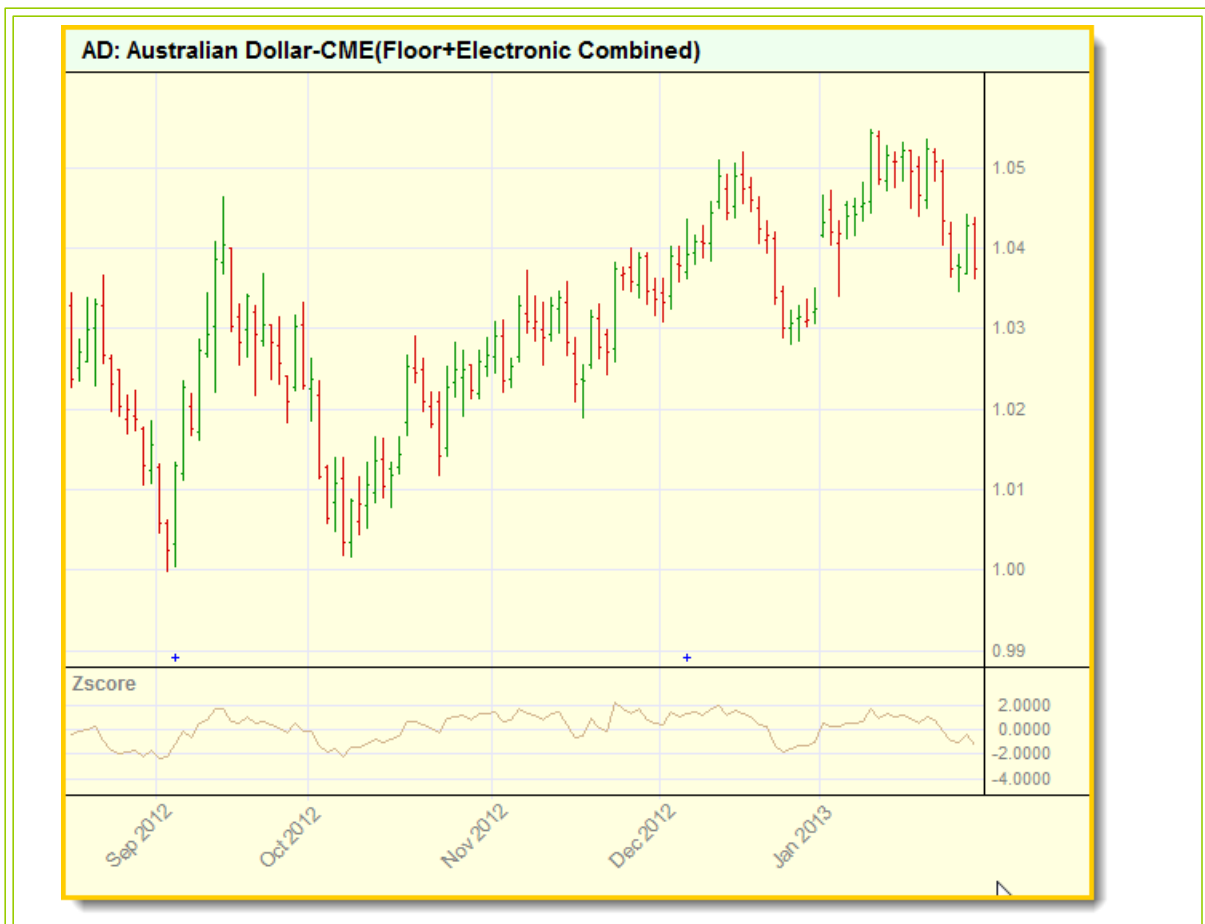
```

OR

The screenshot shows a 'New Indicator' dialog box with the following configuration:

- Name for Code: ZScoreClose
- Type: ZScore
- Value: Close
- Time Frame: Bar
- Period: Bar_Count
- Not Applicable: (empty)
- Not Applicable: (empty)

Chart Display:



Links:

See Also:

ValueChart

Volatility adjusted overbought/oversold oscillator. Value chart levels between -4 and +4 are considered “fair value”; +4 to +8 moderately overbought; -4 to -8 moderately oversold. Levels above +8 are considered significantly overbought and below -8 significantly oversold.

Value Charts are discussed in the book “Dynamic Trading Indicators” by Mark W. Helweg & David C. Stendahl.

Syntax:

```
ValueChart ( series, bars )
```

Parameter:	Description:
series	Name of the series. The series must be one of Instrument.High, Instrument.Low, Instrument.Close, Instrument.Open.
bars	The number of bars over which to find the value chart level.

Notes:

In copyright publication dated 2002 Value Charts are described on page 127 to 145.

Book uses the value of the Value Chart return to decide if a bar on the chart qualifies for a signal.

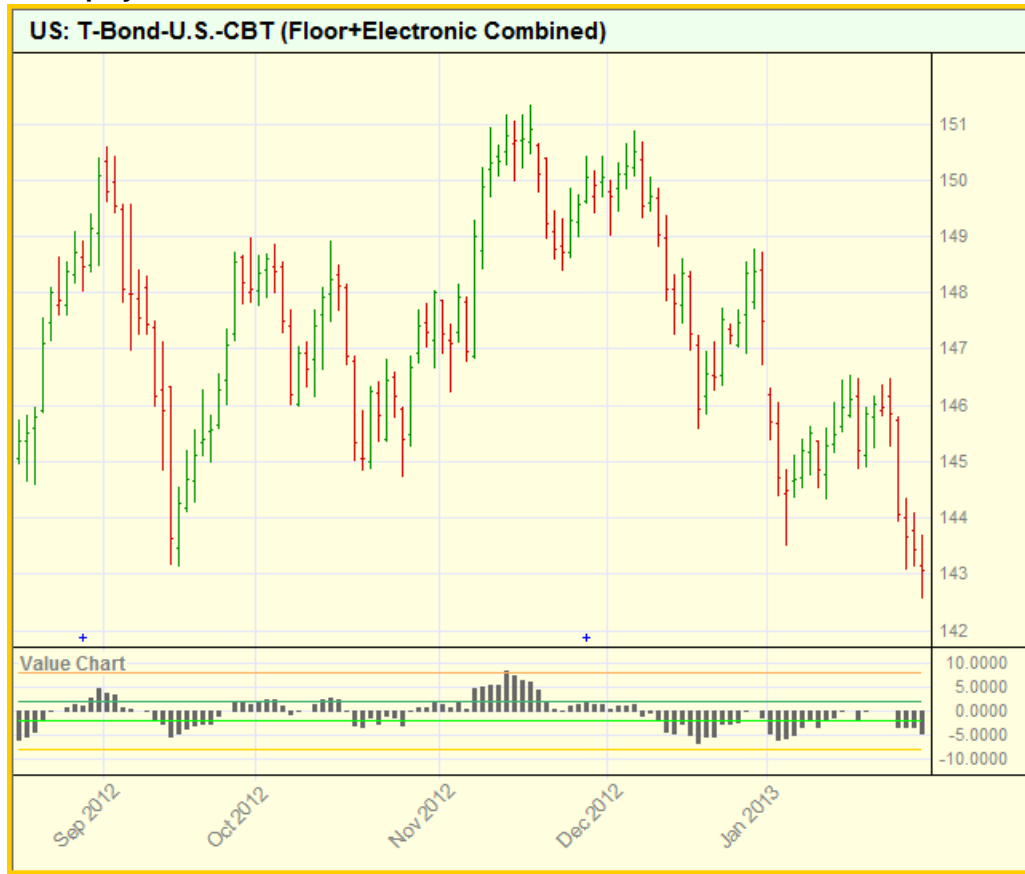
Returns:

The value chart level over the specified number of bars.

Example:

```
' ~~~~~
' Update Indicators
' ~~~~~
' Boundary Levels above and below zero
' Two Parameter positive numbers feed upper and boundary
' levels by inverting values for lower boundaries
VC_Level2Up = VC_Level2   ' 8  IPV Series Plot
VC_Level1Up = VC_Level1   ' 2  IPV Series Plot
VC_Level1Dn = -VC_Level1  ' -2 IPV Series Plot
VC_Level2Dn = -VC_Level2  ' -8 IPV Series Plot

' Value Chart Indicator Feeds IPV Series with BarCount length
Value_Chart = ValueChart(Instrument.Close, BarCount)
' ~~~~~
```

Chart Display:**Links:**[Numeric Series](#)**See Also:**[Data Group and Types](#)

Section 12 – Operator Reference

Mathematical Operators: (+, -, *, /, ^, mod or %, >, <, <> or !=):

All mathematical expressions should be enclosed in parentheses when you are uncertain about how the precedence of calculations will be applied.

Operator Symbol:	Description:
+	Sums two variables. result = (expression1 + expression2)
-	Finds the difference between two numbers. result = (expression1 - expression2)
*	Multiplies two numbers. result = (expression1 * expression2)
/	Divides two numbers. result = (expression1 / expression2)
^	Raises expression1 to the power of an expression2. The result is always floating. result = (expression1 ^ expression2)
mod or %	Divides the value of one expression by the value of another, and returns the remainder. The left and right expressions can be floating or integer. The results will be float or int depending on the values used. result = (expression1 mod expression2) example: <code>dayOfWeek = instrument.julianDate mod 7</code>
>	Greater than symbol used in a conditional reference test where a True or a False is needed. 1 > 2 = True, 2 > 1 = False
<	Less than symbol used in a conditional reference test where a True or a False is needed. 1 < 2 = True, 2 < 1 = False
<> or !=	Both symbol pairs are used when the values on either side are Not True. 1 <> 2 = True, 2 <> 2 = False, 4 != 3 = True

Boolean Operators (AND NOT OR):

Operator Names:	Description:
AND	Performs a logical conjunction on two expressions. The result is always an Integer. expression1 AND expression2
NOT	Performs logical negation on an expression. The result is always an Integer. NOT expression
OR	Performs a logical disjunction on two expressions. The result is always an Integer. expression1 OR expression2

These operators can be enclosed in parentheses like a mathematical expression to force evaluation in a certain order.

```

If ( ( a = 1 ) AND ( b > 5 ) ) OR ( ( a = 2 ) AND ( b < 6 ) AND ( NOT
c ) ) THEN
    ' Do something
Else
    ' Do something else
Endif

```

12.1 Comparison

Comparison in Blox Basic is similar to other programming languages. It is recommended that expressions should be enclosed in parentheses so it is not ambiguous what is being compared with what.

=

Equivalent to.

```
IF ( var1 = var2 ) THEN ...
```

!=

<>

Not equivalent to. Both symbols are valid.

```
VARIABLES: i TYPE: Integer, var1 TYPE: String
```

```

WHILE ( i != 10 )
    var1 = "Message for you, sir."
    i = ( i + 1 )
ENDWHILE

```

>

Greater than.

<

Less than.

<=

Less than or equal to.

>=

Greater than or equal to.

Section 13 – Permanent Variables

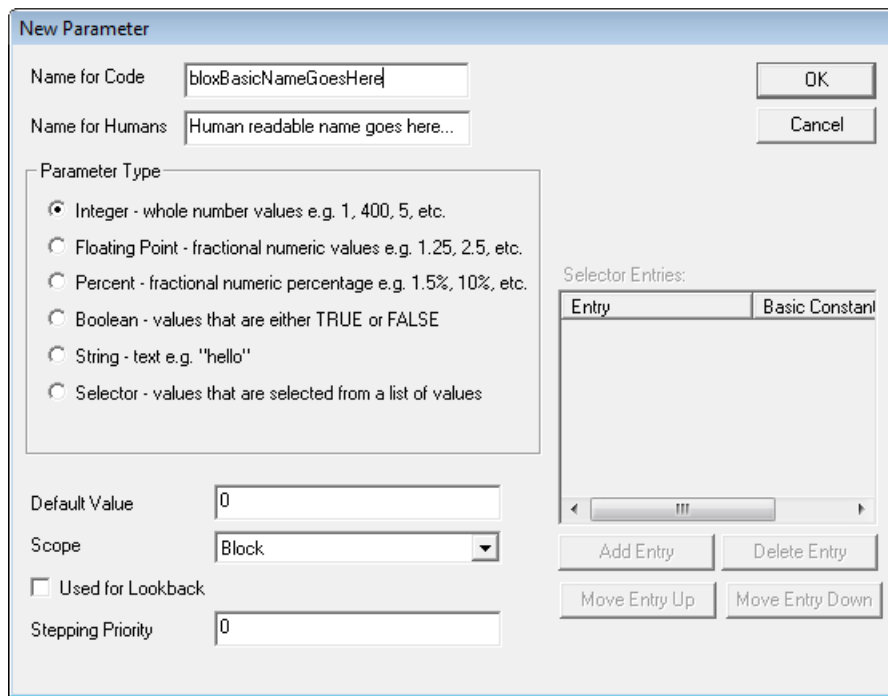
Enter topic text here.

13.1 Data Parameter Reference

Parameters are a very powerful feature of Trading Blox. Parameters allow you to create systems that present their own user interface to allow the user to easily change system settings.

For example, you can create a parameter to be used for the number of days in a moving average, and then change that value when you run your system for historical testing or trading purposes without altering your Blox or Systems. Trading Blox automatically creates a user interface for your parameters which give the user the option of setting a fixed value or stepping through a series of values for every parameter you define. Parameters can also be referenced directly as if they were a variable in any scripts in the block where they are defined.

To create a Parameter, select "Parameters" on the left and click the "New" button.



The Parameter Type is an Integer in the case above. The supported types are:

Integer	whole number values e.g. 1, 400, 5, -10
Floating Point	fractional numeric values e.g. 1.25, 2.5, 187.41415
Percent	numeric percentage e.g. 1.5%, 10%. Enter these as a decimal (.50 for 50%).
Boolean	values that are either TRUE or FALSE
String	a string value such as "hello" or "20081001"

Selector

values that are selected from a list of values. You can assign values using the "Selector Entries". These values will appear in a drop-down menu, and can be stepped through. An example might be "Trade Long", "Trade Short", and "Trade All". The Basic Constant is the code you would use to reference this selected value. This is filled in automatically when you create a selector

Default Value

The default value is the number initially assigned to the parameter when first presented in the Parameter Editor for a system. After that initial setting Trading Blox Builder will remember the current value, so the Default Value is only used the first time a system is used for a Test Suite..The Default Value for the parameter above is 0.

Scope

Set the scope based on what blocks and scripts need access to this parameter. If you set to Block scope (default) only the scripts in the block will have access. If you set to System scope, then all scripts in all block in the system will have access. If you set to Test scope, then all scripts in the test will have access.

Used for Lookback

Check this box if the parameter is going to be used to reference past values of an indicator or past values of an instrument.

For example, if you are using the following type of code in your script, the parameter "closeLookback" should be a lookback parameter:

```
IF instrument.close[ closeLookback ] > instrument.close THEN
```

If you are only using this parameter as input to an indicator, then you do not need to check the lookback box.

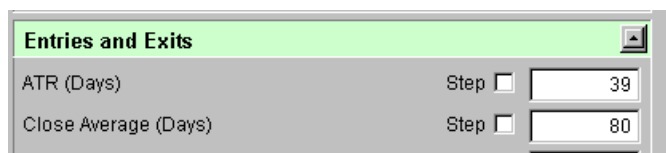
Priming will be increased by this lookback value plus 1. So for a lookback value of 5, the first day scripts would run is day 6. Overall priming is the maximum bars required for indicators plus one, plus the maximum lookback parameter plus one.

Stepping Priority

This sets the priority of this parameter for stepping purposes. When stepping multiple parameters, the highest value priority will be stepped first, and the lowest will be the outer step. The global parameters have a step priority of zero, so to have your custom parameters step after the globals, use a negative step value. This can be left at zero for most situations.

User Interface

Here is how this parameter looks to the user:



We can access this parameter in scripts by using "closeAverageDays". Parameter can also be used for inputs to [Indicators](#). In fact, this is probably the most common use for parameters.

NOTE: Parameters are READ ONLY. They can be accessed by scripts, but not changed by them.

13.2 Block Permanent Variables

Block Permanent Variables are the same regardless of instrument, and retain their value (unlike variables declared with the [VARIABLES](#) statement). For instance, dayNumber (shown below) will be the same for Gold or Silver unless changed in a script.

To create a Block Permanent Variable, select "Block Permanent Variables" on the left and click the "New" button.

Name for Code

The name of variable as you would use it in a Script. Common convention is that you start variable names with a lower case, and use upper case for the first letter of each subsequent word or part of the variable name. No spaces or special characters are allowed in variable names.

Name for Humans

A more friendly description of the variable. In the case of Block Permanent this name is not displayed anywhere, but is useful to remember what the variables purpose is.

Variable Type

The kind of value the variable will be. This cannot be changed in a script.

For a description of the different types, see the [VARIABLES](#) section. In addition to the two types listed there Block Permanent Variables can also be of type Series and Instrument:

- Series - A series is a list of numbers, sometimes referred to as an array. These can be tied to the test day using the "Auto-Index" feature described below.
- A Series of Strings can be used for many purposes, but also as the x axis label of custom [charts](#).

- Instrument - An instrument variable can be used to refer to specific market indexes or to iterate over the instruments in a portfolio.

The Default Value is the value that will be assigned to the variable when it is first used. So in this case at the start of running the program, the variable `dayNumber` will be set to 50.

Upon creation, you can now use this variable in any of the Scripts of the Block using the variable name "dayNumber".

The **Scope** determines where you can use this variable.

- Block -- You can only use this variable in the scripts that are in the block.
- System -- You can use this variable in any block in the System.
- Test -- You can use this variable in any block in the Test.
- Simulation -- Same as Test scoped except the value is not reset for every test (parameter run).

Plotting BPV Test Scope Series variables on the Summary Custom Chart

If you define a BPV as a Series variable that is System or Test scope you have the option to plot this series on a Summary Custom Chart. Select Plots on Trade Graph, the color, style, and select a Graph Area. The Graph Area is used for the title of the chart. If you have more than one plotting variable with the same Graph Area, they will be plotted together so make sure the range is common and consistent. If you have multiple different Graph Areas in your test, then multiple Summary Custom Charts will be created for each one.

Be sure to check the *Custom Graph* checkbox in *Preferences* for the chart to show up. See the Trading Blox User manual for details.

If any plotting BPV series are defined as "Log Scale" then all series plotting in the same Graph Area will plot as log scale.

A BPV Series with Auto Index checked, will track the `test.currentDay` index.

Example:

To plot the Margin to Equity ratio, create the BPV below and put the following in the After Trading Day script:

```
plotMarginEquity = test.totalMargin / test.totalEquity * 100
```


The screenshot shows a dialog box titled "Block Permanent Variable". It contains the following fields and options:

- Name for Code:** A text box containing "plotMarginEquity".
- Name for Humans:** A text box containing "Margin/Equity".
- Defined Externally in Another Block:** An unchecked checkbox.
- Variable Type:** A group box containing several radio button options:
 - Integer - whole number values e.g. 1, -2, 400, 5, etc.
 - Floating Point - fractional number e.g. 2.5, 1.414, etc.
 - Price - fractional number in the range of prices
 - String - "Hello", "Goodbye", etc.
 - Series - a series or list of numbers** (selected)
 - Series - a series or list of strings
 - Instrument - used to load and access alternate markets
- Plotting Controls:** A group box containing:
 - Checked checkboxes for "Plots" and "Display Value".
 - An unchecked checkbox for "Log Scale".
 - "Plot Color": A color selection box currently showing blue.
 - "Graph Area": A text box containing "Margin".
 - "Graph Style": A dropdown menu currently set to "Thin Line".
- Variable Options:** A group box containing:
 - "Default Value": A text box containing "0.000000".
 - "Scope": A dropdown menu currently set to "System".
 - Checked checkbox for "Auto-Index - Uses [n] as Lookback from Current Day".

Buttons for "OK" and "Cancel" are located in the top right corner.

13.3 Instrument Permanent Variables

These variables can be different for each instrument, but retain their value (unlike variables declared with the [VARIABLES](#) statement). For instance, totalProfit (shown below) can be 100 for Soybeans and 50 for Gold.

To create a Instrument Permanent Variable, select "Instrument Permanent Variables" on the left and click the "New" button.

The Name for Code is the name of variable as you would use it in a Script. Common convention is that you start variable names with a lower case, and use upper case for the first letter of each subsequent word or part of the variable name. No spaces or special characters are allowed in variable names.

The Name for Humans is a more friendly description of the variable. In the case of Instrument Permanent this name is not displayed anywhere unless the variable is a series, but is useful to remember what the variables purpose is. For series variables, the Name for Humans is used as the label.

Defined Externally in Another Block -- check this option if this variable has been declared as System Scope in another block in the system. This option lets the Syntax Checker know about this variable.

The Variable Type is the kind of value the variable will be, and cannot be changed in a script. For a description of the different types, see the [VARIABLES](#) section.

An IPV of type Series String can be used to display a different string value for each bar on the trade chart. It will not plot of course.

The Default Value is the value that will be assigned to the variable when it is first used. So in this case at the start of running the program, the variable lastMonth will be set to 0.

Upon creation, you can now use this variable in any of the Scripts of the Block using the variable name "lastMonth."

The **Scope** determines where you can use this variable. Instrument Permanent Variables cannot be Test scope.

- Block -- You can only use this variable in the scripts that are in the block.
- System -- You can use this variable in any block in the System by declaring the variable as External in the other blocks.

- Simulation -- Same as System scoped except the value is not reset for every test (parameter run).

Note that for Instrument Permanent Variables, if you select the Series variable type, you have the option to plot. If you want to plot the value, it is recommended to use the Auto Index feature. The Auto Index will set the index using the instrument.bar property.

As with [Indicators](#), select Plots on Trade Chart, Displays on Trade Chart, Select the color, set the Graph Area, Select Offset by One Day if desired. The Graph Style has many options. Use the one most appropriate to your situation.

For both Block Permanent and Instrument Permanent, if you do not select Auto Index, you must specify a size for the array. This example, currentStopPrice, is an Auto Index series that plots in the Price Chart graph area in Red as Small Dots.

Access

You can access Instrument Permanent Variables through scripting two ways.

1. Using the variable directly. This will return the variable for the current instrument:

```
myInstrumentVariable = 5
IF myInstrumentVariable = 5 THEN PRINT "It is 5"
```

If a Series object you can access and set the index values:

```
mySeriesVariable[1] = 5
IF mySeriesVariable[1] = 5 THEN PRINT "Yesterday was 5"
```

2. Or you can access instrument variables using the instrument or another instrument variable object as follows:

```
instrument.myInstrumentVariable = 5
sp500Index.myInstrumentVariable = 5
sp500Index.mySeriesVariable[1] = 5

IF instrument.myInstrumentVariable = 5 THEN PRINT "S&P has 5"
IF sp500Index.mySeriesVariable[1] = 5 THEN PRINT "S&P has 5"
```

Accessing a variable using the instrument '.' syntax is equivalent to using the variable directly. The '.' syntax is the only way to access the value of instrument variables which are not part of the current instrument.

Section 14 – Series Functions

The following functions can be used with series variables.

Function Name:	Description:
Average	Finds the average value of the series.
Correlation	Finds the correlation of two series.
CorrelationLog	Finds the correlation of two series, using the log of the change in price.
CorrelationLogSynch	Used for IPV Auto Indexed Series or Price Series and synchs the dates of the two series before computing the correlationlog
CorrelationSynch	Used for IPV Auto Indexed Series or Price Series and synchs the dates of the two series before computing the correlation
CrossOver	Returns true if two series have crossed over.
Data Series Indexing	Explanation of how Series are indexed.
GetReference	Used to pass a series reference to a Custom Function
GetSeriesSize	Returns the current size of the series.
Highest	Find the highest value of the series.
HighestBar	Returns the number of bars back from the starting offset of the highest bar.
Lowest	Finds the lowest value of the series.
LowestBar	Returns the number of bars back from the starting offset of the lowest bar.
Median	Returns the median.
RegressionEnd	Finds the end point (Y axis) after a call to RegressionSlope
RegressionSlope	Finds the slope of the linear regression.
RegressionValue	Finds the value of a linear regression of the series at any point using an offset
RSI	Computes the RSI of a series.
SetSeriesColorStyle	Function will color an IPV Auto-Index series, or a Indicator section indicator created to hold decimal-numbers or text String .
SetSeriesSize	Sets the size of the series.
SetSeriesValues	Sets a value into every element of the series
SortSeries	Sorts the series
SortSeriesDual	Sorts series1 based on the values of series2
StandardDeviation	Finds the standard deviation of the series.
StandardDeviationLog	Returns the standard deviation of the log of the change in prices.
Sum	Finds the sum of the series.
SwingHigh	Returns the swing high value.

SwingHighBar	Returns the number of bars back from the starting offset of the swing high bar.
SwingLow	Returns the swing low value.
SwingLowBar	Returns the number of bars back from the starting offset of the swing low bar.

The series on which these function apply:

```

instrument.open
instrument.high
instrument.low
instrument.close
instrument.volume
instrument.openInterest
instrument.unAdjustedClose
instrument.extraData1 through instrument.extraData8
instrument.weekOpen (indexed by week)
instrument.weekHigh (indexed by week)
instrument.weekLow (indexed by week)
instrument.weekClose (indexed by week)
BPV (Block Permanent Series) Variable
IPV (Instrument Permanent Series) Variable
Indicators

```

Example:

```

' myCustomArray is defined as an Instrument Permanent
' non Auto Indexed Series Variable
' Finds the average of elements number 8, 9, and 10:
myAverage = Average( myCustomArray, 3, 10 )

```

NOTES:

If you use this function on an "Auto Indexed" Instrument Permanent or Block Permanent Series variable, then the offset parameter sets the start index as a **lookback** from the current instrument bar or test day. However, if you use this function on a non "Auto Indexed" series variable, then the offset parameter is the **start index**. The function uses the bars prior to and including the start index for the calculation.

For information on using functions with non auto indexed series review [Series Functions](#).

14.1 Average

Finds the average value of the series.

Syntax

```
Average( series, bars, [offset] )
```

Parameters

<i>series</i>	the name of the series
<i>bars</i>	the number of bars over which to find the value
<i>offset</i>	the number of bars to offset before finding the value
returns	the average

Example

```
VARIABLES: highestClose, highestHigh, lowestLow, averageClose,
standDev TYPE: Price

' Find the highest close of the last 50 bars
highestClose = Highest( instrument.close, 50 )

' Find the lowest low of the last 100 bars
lowestLow = Lowest( instrument.low, 100 )

' Find the highest high since the entry of the first unit of the
current position
IF instrument.position <> OUT THEN
  highestHigh = Highest( instrument.high, instrument.unitBarsSinceEntry
)
ENDIF

' Find the 10 day average of the close starting 20 days ago
averageClose = Average( instrument.close, 10, 20 )

' Find the standard deviation of the close over the last 100 days
standDev = StandardDeviation( instrument.close, 100 )
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.2 Correlation

The Correlation function returns the statistical correlation for the specified number of bars between two markets.

Syntax

```
Correlation( series1, series2, barsToMeasure, [offset1], [offset2] )
```

Parameters

<i>barsToMeasure</i>	the number of bars over which to measure the correlation
<i>series1</i>	the first series

<i>series2</i>	the second series
offset1	the offset of the first series
offset2	the offset of the second series
returns	statistical correlation for last <code>barsToMeasure</code> bars

Example

```
soybeans.LoadSymbol( "S" )
gold.LoadSymbol( "GC" )
correlation = Correlation( soybeans.close, gold.close, 500 )
```

Returns the correlation between GC and S over the last 500 days

Returns a decimal number between -1 and 1. Returns -1 if the two series are perfectly negatively correlated. Returns 0 if the two series are not correlated at all. Returns 1 if the two series are perfectly positively correlated.

Generally less than .7 is considered uncorrelated, while .7 to .9 is considered loosely correlated and greater than .9 is considered closely correlated.

14.3 CorrelationLog

The CorrelationLog function returns the statistical correlation for the specified number of bars between two markets. It measure the correlation based on the change in the log of the values in the series.

Syntax

```
CorrelationLog( series1, series2, barsToMeasure, [offset1], [offset2] )
```

Parameters

<code>barsToMeasure</code>	the number of bars over which to measure the correlation
<i>series1</i>	the first series
<i>series2</i>	the second series
offset1	the offset of the first series
offset2	the offset of the second series
returns	statistical correlation for last <code>barsToMeasure</code> bars

Example

```
soybeans.LoadSymbol( "S" )
gold.LoadSymbol( "GC" )
correlation = CorrelationLog( soybeans.close, gold.close, 500 )
```


Returns the correlation between GC and S over the last 500 days

Returns a decimal number between -1 and 1. Returns -1 if the two series are perfectly negatively correlated. Returns 0 if the two series are not correlated at all. Returns 1 if the two series are perfectly positively correlated.

Generally .7 or greater is considered loosely correlated and .9 or greater is considered closely correlated.

14.4 CorrelationLogSynch

Syntax:	
Parameter:	Description:
Example:	
Results:	
Links:	
See Also:	

14.5 CorrelationSynch

Syntax:	
Parameter:	Description:
Example:	
Results:	
Links:	
See Also:	

14.6 CrossOver

Finds the cross over of two series.

Syntax

```
CrossOver( series1, series2, [direction], [crossOverSize], [offset] )
```

Parameters

series1	series1
series2	series2
direction	the direction of the cross over. A positive number looks for when series1 goes from below series2 to above series2. A negative number looks for when series1 goes from above series2 to below series2. The default is 1.
crossOverSize	the number of bars to consider when looking for the cross. Default value is 1 bars which will check if the cross occurred between the last bar and the current bar.
offset	the number of bars to offset before finding the value. The default is to look at the current bar.
returns	TRUE if there was a cross over, FALSE if not.

Example

' Check if there was a cross over between the short moving average and the long moving average.

```
IF CrossOver( shortMovingAverage, longMovingAverage ) THEN  
  PRINT "Yes, there was a cross over today."  
ENDIF
```

14.7 Data Series Indexing

Series can be Auto Indexed or indexed manually.

Series can be auto indexed according to a test's `test.currentDay`, an instrument's `instrument.bar`, or manually indexed by script code. An Auto-Indexed series is sized automatically to match the highest value of their indexing index. A manually indexed series must first be sized in its declaration dialog, and then maintained by increasing, clearing and even reducing for some uses by script code.

This means that when accessing the series variable the value in the "[]" braces is a location reference. Auto-Indexed series also automatically have their size set according to the number of bars of instrument data or number of days in the test depending whether they are an Instrument or Block Permanent Variable.

Example of an **Auto-Indexed** series:

Day 1:

```
customIndicator = 10      ' Sets the value of 10 for the series
                          ' at day 1.
```

Day 2:

```
customIndicator = 20      ' Sets the value of 20 for the series
                          ' at day 2.
```

Day 3:

```
Print customIndicator[1]  ' Will print the number 20 since that
                          ' is the value from one day back.
```

Note that the Block Permanent Series will Auto Index on the test's CurrentDay value, while the Instrument Permanent Series will Auto Index on the instrument's bar value. This difference is because some markets/instruments don't trade on certain days because of exchange-specific holidays.

Example of accessing a series that is **not Auto-Indexed** (regular array):

If you do not select Auto-Index then each element is accessed the same regardless of the movement in through time. You need to set each value using the [] and retrieve them using the same index. You also need to set the size yourself since Trading Blox won't know how many elements you want to store in the series.

Errors will be generated when the non auto indexed array is used without an index [] or when the index is less than 1 or greater than the number of defined elements.

In the following example, customArray is a non auto-indexed array of size 10.

Day 1:

```
customArray[1] = 10      ' Sets the value of 10 into the
                          ' series at index 1
```

Day 2

```
customArray[3] = 20      ' Sets the value of 20 into the
                          ' series at index 3
```

Day 3

```
Print customArray[1]     ' Will print the value 10, since
```

```
' that is the value at index 1
```

14.8 GetReference

Syntax:	
Parameter:	Description:
Example:	
Results:	
Links:	
See Also:	

14.9 GetSeriesSize

Get the current size of the series. Used with Non Auto Indexed Series to find the size when assigning values. If the series is too small, increase with [SetSeriesSize](#).

Syntax:

```
seriesSize = GetSeriesSize( seriesName )
```

Parameter:

seriesName

Description:

Name of manually sized IPV & BPV series arrays. Manually means the Auto-Index option is disabled.

returns

Variable 'seriesSize' shown above will contain the number of elements contained in the series.

Example:

```
VARIABLES: seriesSize    TYPE: integer

' Set instrument.myCustomSeries size to 34
SetSeriesSize(instrument.myCustomSeries, 34)

' Get the current size.
seriesSize = GetSeriesSize( instrument.myCustomSeries )

' If we have enough space, then set the value.
IF index < seriesSize THEN
    instrument.myCustomSeries[ index ] = someNumber
ELSE
    ERROR "The index is too large for the series myCustomSeries"
ENDIF

Print "seriesSize = ", seriesSize
```

Results:

```
seriesSize = , 34
```

Links:

[SetSeriesSize](#), [SetSeriesValues](#)

See Also:

[Series Functions](#)

14.10 Highest

Finds the highest value of the series.

Syntax

```
Highest( series, bars, [offset] )
```


Parameters

<i>series</i>	the name of the series
<i>bars</i>	the number of bars over which to find the value
<i>offset</i>	the number of bars to <i>offset</i> before finding the value
returns	the highest value

Example

```
VARIABLES: highestClose, highestHigh, lowestLow, averageClose,
standDev TYPE: Price

' Find the highest close of the last 50 bars
highestClose = Highest( instrument.close, 50 )

' Find the lowest low of the last 100 bars
lowestLow = Lowest( instrument.low, 100 )

' Find the highest high since the entry of the first unit of the
current position
IF instrument.position <> OUT THEN
  highestHigh = Highest( instrument.high, instrument.unitBarsSinceEntry
)
ENDIF

' Find the 10 day average of the close starting 20 days ago
averageClose = Average( instrument.close, 10, 20 )

' Find the standard deviation of the close over the last 100 days
standDev = StandardDeviation( instrument.close, 100 )
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.11 HighestBar

Finds the highest value of the series, then return the bars back.

Syntax

```
HighestBar( series, bars, [offset] )
```

Parameters

<i>series</i>	the name of the series
<i>bars</i>	the number of bars over which to find the value
<i>offset</i>	the number of bars to <i>offset</i> before finding the value
returns	the number of bars back from the offset starting

index to the highest value bar

Example

```
VARIABLES: highestCloseBar, highestHighBar, lowestLowBar TYPE: Price

' Find the highest close bar of the last 50 bars
highestCloseBar = HighestBar( instrument.close, 50 )

' Find the lowest low bar of the last 100 bars
lowestLowBar = LowestBar( instrument.low, 100 )

' Find the highest high bar since the entry of the first unit of the
current position
IF instrument.position <> OUT THEN
    highestHighBar = HighestBar( instrument.high,
instrument.unitBarsSinceEntry )
ENDIF

' Now print the close of the highest bar:
PRINT instrument.close[ highestHighBar ]
```

The return value is the number of bars back from the starting index. If no starting index is used, then it is the bars back from the current bar. If a starting index offset is used, then the return value is the bars back from that offset.

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.12 Lowest

Finds the lowest value of the series.

Syntax

```
Lowest( series, bars, [offset] )
```

Parameters

series	the name of the series
bars	the number of bars over which to find the value
offset	the number of bars to <i>offset</i> before finding the value

returns	the lowest value
---------	------------------

Example

```
VARIABLES: highestClose, highestHigh, lowestLow, averageClose,
standDev TYPE: Price

' Find the highest close of the last 50 bars
highestClose = Highest( instrument.close, 50 )
```

```

' Find the lowest low of the last 100 bars
lowestLow = Lowest( instrument.low, 100 )

' Find the highest high since the entry of the first unit of the
current position
IF instrument.position <> OUT THEN
  highestHigh = Highest( instrument.high, instrument.unitBarsSinceEntry
)
ENDIF

' Find the 10 day average of the close starting 20 days ago
averageClose = Average( instrument.close, 10, 20 )

' Find the standard deviation of the close over the last 100 days
standDev = StandardDeviation( instrument.close, 100 )

```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.13 LowestBar

Finds the lowest value of the series, then return the bars back.

Syntax

```
LowestBar( series, bars, [offset] )
```

Parameters

series	the name of the series
bars	the number of bars over which to find the value
offset	the number of bars to <i>offset</i> before finding the value
returns	the number of bars back from the offset starting index to the lowest value bar

Example

```

VARIABLES: highestCloseBar, highestHighBar, lowestLowBar TYPE: Price

' Find the highest close bar of the last 50 bars
highestCloseBar = HighestBar( instrument.close, 50 )

' Find the lowest low bar of the last 100 bars
lowestLowBar = LowestBar( instrument.low, 100 )

' Find the highest high bar since the entry of the first unit of the
current position
IF instrument.position <> OUT THEN
  highestHighBar = HighestBar( instrument.high,
instrument.unitBarsSinceEntry )
ENDIF

```

```
' Now print the close of the highest bar:
PRINT instrument.close[ highestHighBar ]
```

The return value is the number of bars back from the starting index. If no starting index is used, then it is the bars back from the current bar. If a starting index offset is used, then the return value is the bars back from that offset.

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.14 Median

Finds the median value of the series. The median is the middle value of the series if the series has an odd number of elements. If there are an even number of elements, the median is the average of the middle two values.

Syntax

```
Median( series, [bars], [offset] )
```

Parameters

series	the name of the series
bars	the number of bars over which to find the value. Default is the whole series.
offset	the number of bars to offset the starting index. Default is zero.
returns	the median

Example

```
' Find the median value of the last 100 bars of the series.

medianValue = Median( instrument.close, 100 )
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.15 RegressionEnd

Finds the end point of a linear regression of the series. Note that this function must be used in conjunction with [RegressionSlope](#). This function merely returns the value already calculated by [RegressionSlope](#), so this function must follow [RegressionSlope](#) to return the correct value. The series is a required input parameter. There are no other required values since the end point has already been determined.

Syntax

```
RegressionEnd( series )
```

Parameters

series	the name of the series
returns	the end point

Example

```
' Find the slope of the linear regression of the last 10 closes.
' Once the slope has been found, we can retrieve the end point as
well.
slope = RegressionSlope( instrument.close, 10 )
endPoint = RegressionEnd( instrument.close )

' Plot the linear regression on the bars that have been used to
calculate it. This is postdictive.
' The variable plotRegression is declared as an Instrument Permanent
Auto Index Series Variable with plotting.

FOR i = 0 to 9
    plotRegression[i] = endPoint - i * slope
NEXT
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.16 RegressionSlope

Finds the linear regression slope of the series.

Syntax

```
RegressionSlope( series, bars, [offset] )
```

Parameters

series	the name of the series
bars	the number of bars over which to find the value
offset	the number of bars to <i>offset</i> before finding the value
returns	the slope

Example

```
' Find the slope of the linear regression of the last 10 closes.
' Once the slope has been found, we can retrieve the end point as
well.
slope = RegressionSlope( instrument.close, 10 )
endPoint = RegressionEnd( instrument.close )
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.17 RegressionValue

Finds the value of a linear regression of the series at any point using an offset. Note that this function must be used in conjunction with [RegressionSlope](#). This function merely returns the slope and endpoint already calculated by [RegressionSlope](#), so this function must follow [RegressionSlope](#) to return the correct value. The series is a required input parameter and an offset.

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

Syntax:

```
RegressionValue( series, offset )
```

Parameter:	Description:
series	Name of the series
offset	Number of bars back
returns	Value at the offset

Example:

```
' Find the slope of the linear regression of the last 10 closes.
slope = RegressionSlope( instrument.close, 10 )

' Plot the linear regression on the bars that have been used to calculate slope
' The variable plotRegression is declared as an Instrument Permanent Auto Indexed Series
FOR i = 0 to 9
  plotRegression[i] = RegressionValue( instrument.close, i )
NEXT
```

Results:

Links:

[RegressionSlope](#)

See Also:

[Series Functions](#)

14.18 RSI

Computes the RSI of a series

Syntax

```
RSI( series, rsiBars, [elementCount], [offset] )
```

Parameters

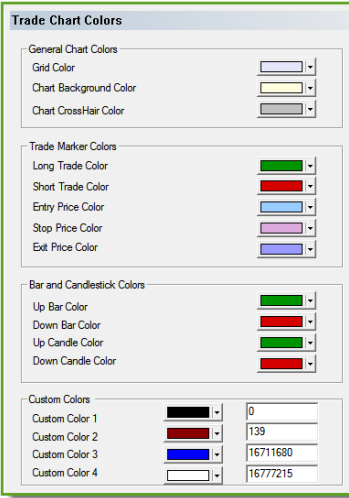
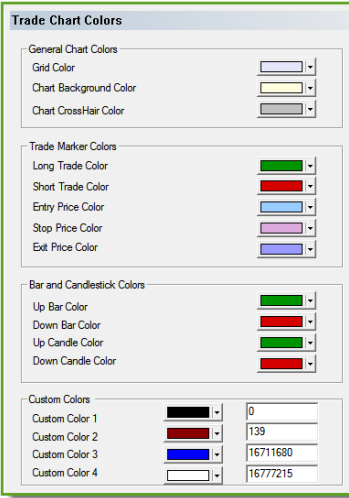
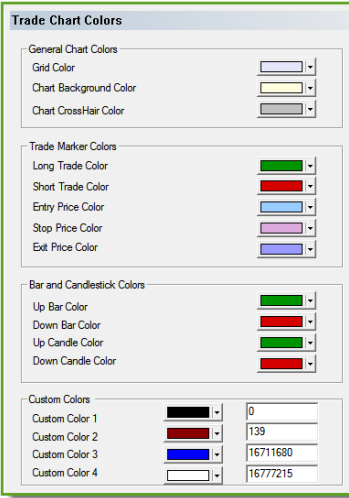
series	the name of the series
rsiBars	the number of bars to use for the RSI computation
elementCount	the optional number of bars to use out of the entire data series for this computation.
offset	the optional offset from the current bar, for auto indexed, and the start location for non auto indexed. Defaults to zero.
returns	the RSI value

14.19 SetSeriesColorStyle

Function allows each element of an Indicator created as a **Floating** or **String** IPV Auto-Index series, or a Indicator section indicator. Series must have its ability to Plot on the software's instrument chart area for this function to work. When applied to a single element the element color will display the color that was assigned.

Syntax:

```
SetSeriesColorStyle(series, plotColor, [plotStyle], [offset] )
```

Parameter:	Description:															
series	Name of series															
plotColor	Element Color to use with named "series"															
	<table border="1"> <thead> <tr> <th>Preference Color Setup:</th> <th>Color Constant Name:</th> </tr> </thead> <tbody> <tr> <td rowspan="13">  <p>User Preference Color Table</p> </td> <td>ColorGrid</td> </tr> <tr> <td>ColorBackground</td> </tr> <tr> <td>ColorCrossHair</td> </tr> <tr> <td>ColorLongTrade</td> </tr> <tr> <td>ColorShortTrade</td> </tr> <tr> <td>ColorTradeEntry</td> </tr> <tr> <td>ColorTradeStop</td> </tr> <tr> <td>ColorTradeExit</td> </tr> <tr> <td>ColorUpBar</td> </tr> <tr> <td>ColorDownBar</td> </tr> <tr> <td>ColorUpCandle</td> </tr> <tr> <td>ColorDownCandle</td> </tr> </tbody> </table>	Preference Color Setup:	Color Constant Name:	 <p>User Preference Color Table</p>	ColorGrid	ColorBackground	ColorCrossHair	ColorLongTrade	ColorShortTrade	ColorTradeEntry	ColorTradeStop	ColorTradeExit	ColorUpBar	ColorDownBar	ColorUpCandle	ColorDownCandle
Preference Color Setup:	Color Constant Name:															
 <p>User Preference Color Table</p>	ColorGrid															
	ColorBackground															
	ColorCrossHair															
	ColorLongTrade															
	ColorShortTrade															
	ColorTradeEntry															
	ColorTradeStop															
	ColorTradeExit															
	ColorUpBar															
	ColorDownBar															
	ColorUpCandle															
	ColorDownCandle															

		ColorCustom1																						
		ColorCustom2																						
		ColorCustom3																						
		ColorCustom4																						
	Note: Review color information available here: Colors																							
<code>[plotStyle]</code>	<table border="1"> <thead> <tr> <th>Style Value:</th> <th>Style Type:</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Thin Line</td> </tr> <tr> <td>2</td> <td>Thick Line</td> </tr> <tr> <td>3</td> <td>Trace</td> </tr> <tr> <td>4</td> <td>Small Dot</td> </tr> <tr> <td>5</td> <td>Large Dot</td> </tr> <tr> <td>6</td> <td>Up Arrow</td> </tr> <tr> <td>7</td> <td>Down Arrow</td> </tr> <tr> <td>8</td> <td>Histogram</td> </tr> <tr> <td>9</td> <td>Area</td> </tr> <tr> <td>10</td> <td>Staircase</td> </tr> </tbody> </table>	Style Value:	Style Type:	1	Thin Line	2	Thick Line	3	Trace	4	Small Dot	5	Large Dot	6	Up Arrow	7	Down Arrow	8	Histogram	9	Area	10	Staircase	
Style Value:	Style Type:																							
1	Thin Line																							
2	Thick Line																							
3	Trace																							
4	Small Dot																							
5	Large Dot																							
6	Up Arrow																							
7	Down Arrow																							
8	Histogram																							
9	Area																							
10	Staircase																							
<code>[offset]</code>	Zero will change the color of the bar information at the current location. An offset value of 1 will change the color of the previous bar. Offset values with this function operate in the same way as a price or auto-index series.																							

Example:

In the following, `barPlotColor` is a Bar Plot Color type indicator that is checked for plotting and unchecked for display.

`barMessage` is an auto indexed string series.

`plotTrades` is an auto indexed numerical series.

```

' ~~~~~
' Setting the location, color, and style of a plotting IPV series.
If instrument.unitBarsSinceEntry = 1 AND instrument.position = LONG THEN
  plotTrades[1] = instrument.low[1] - instrument.minimumTick * 10
  SetSeriesColorStyle( plotTrades, ColorLongTrade, 6, 1 )
ENDIF

If instrument.unitBarsSinceEntry = 1 AND instrument.position = SHORT TH
  plotTrades[1] = instrument.high[1] + instrument.minimumTick * 10
  SetSeriesColorStyle( plotTrades, ColorShortTrade, 7, 1 )
ENDIF

' Bar Plot Color type indicator example to color the trade
' chart bars dynamically.
If instrument.close > instrument.open THEN
  SetSeriesColorStyle( barPlotColor, ColorUpBar )
  barMessage = "Up Day!"
ELSE
  SetSeriesColorStyle( barPlotColor, ColorDownBar )
  barMessage = "Down Day!"
ENDIF
' ~~~~~

```

Example:

```

' ~~~~~
' This will set the color to a random color for each bar
' and the style to a random style - nice look!
SetSeriesColorStyle( averageTrueRange, ColorRGB( Random(255), _
  Random(255), Random(255) ), Random(10))
' ~~~~~

```

Example:

```

' ~~~~~
' Color of the up bar AND a thin line:
SetSeriesColorStyle( averageTrueRange, ColorUpBar, 1 )
' ~~~~~

```

Example:

```

' ~~~~~
' Label Volume using the direction of bar close change value
If instrument.close[0] > instrument.close[1] THEN
'   Display Volume Series
SeriesVolume = instrument.volume[0]
'   Use ColorUpBar Color on this Volume bar
SetSeriesColorStyle(SeriesVolume, ColorUpBar, 8, 0 )
ELSE
If instrument.close[0] < instrument.close[1] THEN
'   Display Volume Series
SeriesVolume = instrument.volume[0]
'   Use ColorDownBar Color on this Volume bar
SetSeriesColorStyle(SeriesVolume, ColorDownBar, 8, 0 )
ELSE
If instrument.close[0] = instrument.close[1] THEN
'   Use Default Color for Volume bar item
SeriesVolume = instrument.volume[0]
ENDIF ' i.close[0] = i.close[1]
ENDIF ' i.close[0] < i.close[1]
ENDIF ' i.close[0] > i.close[1]
' ~~~~~

```

Script code output in this example creates this next chart of volume changes:

Results:



Links:

[Colors](#), [ColorRGB](#)

See Also:

[Preferences](#)

-
- Or use a number like 8388736 which is violet. The Color picker in Preferences displays the number for each custom color, so you can copy paste from there to set a fixed color.

If you use one of the custom colors from preferences, the user will be able to modify the colors as desired.

14.20 SetSeriesSize

Sets the series size of a manually sized and manually indexed series of floating numbers or strings. Manually sized series are created by selecting the one of the two series types available in the IPV and BPV variable creation dialog, and then disabling the Auto-Indexed option.

This function is used to increase or decrease the size of a manually Indexed series. When the size of a series is changed, the values currently in the series are retained, and the new elements (if bigger, or includes more elements) are set to the default value used in the variable creation dialog.

Syntax:

```
SetSeriesSize( seriesName, newSize )
```

Parameter:	Description:
seriesName	Name of the series which is to be adjusted to a new size.
newSize	Integer value of the new series size.

Example:

```
VARIABLES: seriesSize    TYPE: integer

' Get the current series size.
seriesSize = GetSeriesSize( instrument.myCustomSeries )

' If we don't have enough space, then resize.
' Make bigger than necessary so we don't have to do this every time.
IF index > seriesSize THEN
    SetSeriesSize( instrument.myCustomSeries, index + 10 )
ENDIF

' Set the value into the series.
instrument.myCustomSeries[ index ] = someNumber
```

Results:

SetSeriesSize doesn't return any value, but the size of a series can be obtained using the GetSeriesSize function.

Links:

[GetSeriesSize](#), [SetSeriesValue](#)

See Also:

[Series Functions](#)

14.21 SetSeriesValues

Sets a value into all the elements of the series. Optional Start and End element parameters are available to control which elements in the series will be assigned the replacement seed-Value. When the Start and End elements index numbers are not used the entire series will be contain the replacement seed-value.

This function makes clearing the entire series value of its previous values a fast and simple process.

Syntax:

```
SetSeriesValues( seriesName , seedValue , [ Start-Element-Num ] , [ End-Element-Num ] )
```

Parameter:	Description:
seriesName	Name of series to be changed.
seedValue	Replacement value to use for the series, or for the range of elements assigned in the Start and End element number range.
Start-Element-Num	Earliest or lowest series index value from which seed-value will begin replacing the values in the series.
End-Element-Num	Latest or highest series index value at which the seed-value replacements will stop being replaced.

Example:

```

' ~~~~~
' Create a Series with 5-elements
SetSeriesSize( TestSeries, 5 )

' Check the count of the TestSeries
iElementCount = GetSeriesSize( TestSeries )

' Create Column Titles
PRINT
PRINT "Series Elements with Values:"
PRINT "Series Element", "Element Value"
PRINT "-----", "-----"

' Assign an Ndx number to each Element
For Ndx = 1 TO iElementCount STEP 1
' ClearRankAdjustments
TestSeries[Ndx] = Ndx

' Show Each Element Value
PRINT "TestSeries[" + AsString(Ndx, 0) + "] = ", TestSeries[Ndx]
Next ' ndx

' Set All the Elements in the Series to Zero
SetSeriesValues( TestSeries, 0 )

' Create More Column Titles
PRINT
PRINT "Series Elements with New Values:"
PRINT "Series Element", "Element Value"
PRINT "-----", "-----"

' Show each element after the new value is assigned
For Ndx = 1 TO iElementCount STEP 1
' Show Each Element Value
PRINT "TestSeries[" + AsString(Ndx, 0) + "] = ", TestSeries[Ndx]
Next ' ndx
' ~~~~~

```

Results:

```

Series Elements with Values:
Series Element Element Value
-----
TestSeries[1] = 1.000000000
TestSeries[2] = 2.000000000
TestSeries[3] = 3.000000000
TestSeries[4] = 4.000000000
TestSeries[5] = 5.000000000

```

```

Series Elements with New Values:
Series Element Element Value
-----
TestSeries[1] = 0.000000000

```

```
TestSeries[2] = 0.000000000  
TestSeries[3] = 0.000000000  
TestSeries[4] = 0.000000000  
TestSeries[5] = 0.000000000
```

Links:

[GetSeriesSize](#), [SetSeriesSize](#)

See Also:

[Series Functions](#)

14.22 SortSeries

Sorts the series in ascending and descending order. Will sort a limited part of the series any where in the series.

Only use with non-auto indexed series.

Does not work with Auto Indexed series.

Syntax:

```
SortSeries( seriesName, [element count], [offset], [direction] )
```

Parameter:	Description:
seriesName	Series name to be sorted.
[element count]	Default is to sort all the elements in the series. Option is to specify the number of elements to sort.
[offset]	Default is to sort all the elements in the series. When a value is entered into the "element count" field, the value entered in this field will be the starting index from which element count uses to start the optional "element count" amount of elements to sort.
[direction]	Default, or no entry in this field will sort the elements in an ascending order. A positive value sorts the elements in an ascending order. A negative value sorts the elements in a descending order.

Example 1:

```

' ~~~~~
' WtFactors = Manual, or Non Auto-Indexing Series
' ~~~~~
' Adjust Series Element Count
SetSeriesSize( WtFactors, 3)
' Get Series New Size
PRINT
PRINT "Show Series Size"
PRINT "Series Size = ", GetSeriesSize( WtFactors )
' Assign Zero to all the Element in the series
SetSeriesValues( WtFactors, 0)

' Assign Weight Factor to Series
WtFactors[1] = 0.30
WtFactors[2] = 0.40
WtFactors[3] = 0.30

PRINT
PRINT "Unsorted Series"
For Ndx = 1 TO 3 STEP 1
    PRINT "WtFactors[" + AsString(Ndx,0) + "] = ", WtFactors[Ndx]
Next ' Ndx

' Sort Series in Ascending Order
SortSeries( WtFactors)

PRINT
PRINT "Series Sorted in Ascending Order"
For Ndx = 1 TO 3 STEP 1
    PRINT "WtFactors[" + AsString(Ndx,0) + "] = ", WtFactors[Ndx]
Next ' Ndx

' Sort Series in Descending Order
SortSeries( WtFactors, 3, 3, -1)

PRINT
PRINT "Series Sorted in Descending Order"
For Ndx = 1 TO 3 STEP 1
    PRINT "WtFactors[" + AsString(Ndx,0) + "] = ", WtFactors[Ndx]
Next ' Ndx
' ~~~~~

```

Returns:

Show Series Size
Series Size = 3

Unsorted Series
WtFactors[1] = 0.300000000
WtFactors[2] = 0.400000000
WtFactors[3] = 0.300000000

Series Sorted in Ascending Order

```
WtFactors[1] = 0.300000000
WtFactors[2] = 0.300000000
WtFactors[3] = 0.400000000
```

Series Sorted in Descending Order

```
WtFactors[1] = 0.400000000
WtFactors[2] = 0.300000000
WtFactors[3] = 0.300000000
```

Example 2:

```
' Now sort the results series.
SortSeries( results, elementsToSort, elementsToSort, -1 )
```

Links:

[SortSeriesDual](#)

See Also:

[Series Functions](#)

14.23 SortSeriesDual

Sorts the series1 based on the values in series2. Only for non auto indexed series. Not available for auto indexed series.

Note that the three optional parameters are only optional if the preceding parameter is included. The following are valid:

```
SortSeriesDual( series1, series2 )
SortSeriesDual( series1, series2, elementCount )
SortSeriesDual( series1, series2, elementCount, offset )
SortSeriesDual( series1, series2, elementCount, offset, direction )
```

The following is not valid:

```
SortSeriesDual( series1, series2, direction )
```

Syntax

```
SortSeriesDual( series1, series2, [element count], [offset], [direction] )
```

Parameters

series1	Series 1, that will be sorted
series2	Series 2, to use for the sorting
element count	the number of elements to sort. Default is series element count.
offset	the starting index from which element count goes back for non auto indexed series. Default

	is series element count.
direction	positive number sorts ascending, negative number sorts descending. Default is ascending sort.
returns	n/a

Example

```

' Sort resultsIndex based on the values of the results series.
SortSeriesDual( resultsIndex, results, elementsToSort,
elementsToSort, -1 )

' Now sort the results series.
SortSeries( results, elementsToSort, elementsToSort, -1 )

```

14.24 StandardDeviation

Finds the standard deviation of the series.

Syntax

```
StandardDeviation( series, bars, [offset] )
```

Parameters

series	the name of the series
bars	the number of bars over which to find the value
offset	the number of bars to <i>offset</i> before finding the value
returns	the standard deviation

Example

```

VARIABLES: highestClose, highestHigh, lowestLow, averageClose,
standDev TYPE: Price

' Find the highest close of the last 50 bars
highestClose = Highest( instrument.close, 50 )

' Find the lowest low of the last 100 bars
lowestLow = Lowest( instrument.low, 100 )

' Find the highest high since the entry of the first unit of the
current position
IF instrument.position <> OUT THEN
  highestHigh = Highest( instrument.high, instrument.unitBarsSinceEntry
)
ENDIF

' Find the 10 day average of the close starting 20 days ago
averageClose = Average( instrument.close, 10, 20 )

```

```
' Find the standard deviation of the close over the last 100 days
standDev = StandardDeviation( instrument.close, 100 )
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.25 StandardDeviationLog

Finds the standard deviation of the series. Uses the change in the log of the values.

Syntax

```
StandardDeviationLog( series, bars, [offset] )
```

Parameters

series	the name of the series
bars	the number of bars over which to find the value
offset	the number of bars to <i>offset</i> before finding the value
returns	the standard deviation

Example

```
VARIABLES: highestClose, highestHigh, lowestLow, averageClose,
standDev TYPE: Price

' Find the highest close of the last 50 bars
highestClose = Highest( instrument.close, 50 )

' Find the lowest low of the last 100 bars
lowestLow = Lowest( instrument.low, 100 )

' Find the highest high since the entry of the first unit of the
current position
IF instrument.position <> OUT THEN
    highestHigh = Highest( instrument.high, instrument.unitBarsSinceEntry
)
ENDIF

' Find the 10 day average of the close starting 20 days ago
averageClose = Average( instrument.close, 10, 20 )

' Find the standard deviation of the close over the last 100 days
standDev = StandardDeviationLog( instrument.close, 100 )
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.26 Sum

Finds the sum of the series.

Syntax

```
Sum( series, bars, [offset] )
```

Parameters

series	the name of the series
bars	the number of bars over which to find the value
offset	the number of bars to offset before finding the value
returns	the sum

Example

```
' Find the sum of the 10 closes starting 20 days ago
sumCloses = Sum( instrument.close, 10, 20 )
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.27 SwingHigh

Finds the swing high value.

Syntax

```
swingValue = SwingHigh( series, [occurrence], [forwardStrengthBars],
[backwardStrengthBars], [lookbackBars], [offsetBars] )
```

Parameters

series	the series to use
occurrence	the occurrence to look for. Default is the first occurrence of this swing high back from the current bar. A value of 2 will find the second occurrence back, etc.
forwardStrengthBars	the number of bars forward to check for a given bar to determine if a swing occurred. Default is to check one bar. If all forwardStrengthBars are lower than the bar, then the swing occurred. If any are equal, then the swing did not occur. Flat tops are ignored.
backwardStrengthBars	the number of bars backward to check for a given bar to determine if a swing occurred. Default is to check forwardStrengthBars number of bars. If all backwardStrengthBars are lower than the bar, then the swing occurred. If any are

	equal, then the swing did not occur. Flat tops are ignored.
lookbackBars	the total number of bars to check for a swing. Default is to use all available bars.
offset	the number of bars to offset before finding the value. The default is 0, using the current bar.
returns	the swing value. Returns -1 if not found.

Example

```
' Find the second occurrence back where the post 4 and prior 5 bar
values were lower than the current value.
' Check the last 500 bars.

swingValue = SwingHigh( instrument.high, 2, 4, 5, 500 )
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.28 SwingHighBars

Finds the swing high bar.

Syntax

```
barLookbackIndex = SwingHighBar( series, [occurrence],
[forwardStrengthBars], [backwardStrengthBars], [lookbackBars], [offsetBars]
)
```

Parameters

series	the series to use
occurrence	the occurrence to look for. Default is the first occurrence of this swing high back from the current bar. A value of 2 will find the second occurrence back, etc.
forwardStrengthBars	the number of bars forward to check for a given bar to determine if a swing occurred. Default is to check one bar. If all forwardStrengthBars are lower than the bar, then the swing occurred. If any are equal, then the swing did not occur. Flat tops are ignored.
backwardStrengthBars	the number of bars backward to check for a given bar to determine if a swing occurred. Default is to check forwardStrengthBars number of bars. If all backwardStrengthBars are lower than the bar, then the swing occurred. If any are equal, then the swing did not occur. Flat tops are ignored.
lookbackBars	the total number of bars to check for a swing. Default is to use all available bars.

offset	the number of bars to offset before finding the value. The default is 0, using the current bar.
returns	the number of bars back from the offset starting index of the swing bar. Returns -1 if not found.

Example

```
' Find the second occurrence back where the post 4 and prior 5 bar
values were lower than the current value.
' Check the last 500 bars.

swingBar = SwingHighBar( instrument.high, 2, 4, 5, 500 )

' Print the date of the swing bar.
PRINT instrument.date[ swingBar ]
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.29 SwingLow

Finds the swing low value.

Syntax

```
swingValue = SwingLow( series, [occurrence], [forwardStrengthBars],
[backwardStrengthBars], [lookbackBars], [offsetBars] )
```

Parameters

series	the series to use
occurrence	the occurrence to look for. Default is the first occurrence of this swing high back from the current bar. A value of 2 will find the second occurrence back, etc.
forwardStrengthBars	the number of bars forward to check for a given bar to determine if a swing occurred. Default is to check one bar. If all forwardStrengthBars are higher than the bar, then the swing occurred. If any are equal, then the swing did not occur. Flat tops are ignored.
backwardStrengthBars	the number of bars backward to check for a given bar to determine if a swing occurred. Default is to check forwardStrengthBars number of bars. If all backwardStrengthBars are higher than the bar, then the swing occurred. If any are equal, then the swing did not occur. Flat tops are ignored.
lookbackBars	the total number of bars to check for a swing. Default is to use all available bars.
offset	the number of bars to offset before finding the

value. The default is 0, using the current bar.

returns the swing value. Returns -1 if not found.

Example

```
' Find the second occurrence back where the post 4 and prior 5 bar
values were higher than the current value.
' Check the last 500 bars.

swingValue = SwingLow( instrument.low, 2, 4, 5, 500 )
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

14.30 SwingLowBars

Finds the swing low bar.

Syntax

```
barLookbackIndex = SwingLowBar( series, [occurrence],
[forwardStrengthBars], [backwardStrengthBars], [lookbackBars], [offsetBars]
)
```

Parameters

series	the series to use
occurrence	the occurrence to look for. Default is the first occurrence of this swing low back from the current bar. A value of 2 will find the second occurrence back, etc.
forwardStrengthBars	the number of bars forward to check for a given bar to determine if a swing occurred. Default is to check one bar. If all forwardStrengthBars are higher than the bar, then the swing occurred. If any are equal, then the swing did not occur. Flat tops are ignored.
backwardStrengthBars	the number of bars backward to check for a given bar to determine if a swing occurred. Default is to check forwardStrengthBars number of bars. If all backwardStrengthBars are higher than the bar, then the swing occurred. If any are equal, then the swing did not occur. Flat tops are ignored.
lookbackBars	the total number of bars to check for a swing. Default is to use all available bars.
offset	the number of bars to offset before finding the value. The default is 0, using the current bar.
returns	the number of bars back from the offset starting index of the swing bar. Returns -1 if not found.

Example

```
' Find the second occurrence back where the post 4 and prior 5 bar
values were higher than the current value.
' Check the last 500 bars.

swingBar = SwingLowBar( instrument.low, 2, 4, 5, 500 )

' Print the date of the swing bar.
PRINT instrument.date[ swingBar ]
```

This example shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

Section 15 – Statement Reference

Statement syntax is case-insensitive. "WHILE" is the same as "while". We capitalize these for clarity. Also, while the commands within a statement do not have to be indented, this is recommended to show the "flow" of the script at a glance. Trading Blox Builder includes the following statement types:

Keyword	Description
Assignment	Assign a value to a different variable.
DO	a general-purpose loop that repeats a group of statements
ERROR	a statement that can be used to indicate unexpected conditions
FOR	a special purpose loop that executes a set of statements a particular number of times
IF	a statement that does something only if certain conditions are met
PRINT	a statement that writes values to the log window and file
WHILE	a loop that repeats a group of statements as long as a certain condition exists

15.1 Assignment

A [variable](#) is assigned a new value with an Assignment statement.

Syntax

variable = expression

variable An expression to evaluated and assigned to variable

expression An expression to evaluated and assigned to variable

Variables should be declared before being assigned to in an assignment statement.

```
VARIABLES: variableOne TYPE: Floating, greeting TYPE: String
```

```
variableOne = 50.60
```

```
greeting = "Have a nice day."
```

```
PRINT greeting
```

15.2 DO

Repeats a block of statements while a condition is **TRUE** or until a condition becomes **TRUE**

Syntax

```
DO [WHILE | UNTIL condition ]
   [statements ]
LOOP
```

condition Expression that evaluates to True or False .

statements One or more statements executed while or until condition is True .

In this case if a **WHILE** is used, the *condition* is checked first and if the *condition* is **TRUE** then the *statements* are executed and the *condition* is reevaluated again until the *condition* becomes **FALSE** at which point the loop stops.

If an **UNTIL** the *condition* is checked first and if the *condition* is **FALSE** then the *statements* are executed and the *condition* is reevaluated again until the *condition* becomes **TRUE** at which point the loop stops.

Or, you can use this syntax:

```
DO
   [statements ]
LOOP [WHILE | UNTIL condition ]
```

condition Expression that evaluates to True or False .

statements One or more statements executed while or until condition is True .

The second form where the **WHILE** and **UNTIL** follow the **LOOP** keyword at end of the **DO ... LOOP** differs from the first form in that the *statements* are always executed at least once after which the *condition* is evaluated.

If **WHILE** follows the **LOOP** and the condition is **TRUE**, the statements are executed again and the condition is reevaluated until the condition becomes **FALSE**.

If **UNTIL** follows the **LOOP** and the condition is **FALSE**, the statements are executed again and the condition is reevaluated until the condition becomes **TRUE**.

Examples

```
DO
   print a
   a = ( a + 1 )
LOOP WHILE ( a < 11 )
```

```
VARIABLES: fibonacci, lastFibonacci TYPE: INTEGER
fibonacci = 1
lastFibonacci = 1
```

```
DO UNTIL fibonacci > 100
  ' Print out the Fibonacci number.
  PRINT fibonacci

  ' Compute the next fibonacci number
  fibonacci = ( fibonacci + lastFibonacci )
  lastFibonacci = fibonacci
LOOP
```

15.3 ERROR

Stops the program in the debugger on the line where this statement occurs and displays the message defined by the expressions passed to the ERROR statement in the Debugger's message area. Similar to using a [Breakpoint](#), but in this case the program will terminate rather than continue.

Syntax

```
ERROR [expression, expression,...]
```

expression An expression to be printed, separated by commas or semicolons

ERROR without any parameters displays a blank message.

The ERROR statement is useful for detecting unusual conditions that you don't believe should occur.

Example

```
IF stopPrice < 0 THEN
    ERROR "The stop price was negative ", stopPrice
ENDIF
```

See also: [Test.AbortTest](#) and [Test.AbortAllTests](#)

Keywords: Stop, Break, Abort, Assert

15.4 FOR

Repeats a group of statements a specified number of times.

Syntax

```
FOR counter = start TO end [STEP step ]
    [statements ]
NEXT
```

counter Variable used as a loop counter. This variable must be declared as an integer before using. See the [VARIABLES](#) statement.

start Initial value of counter (can be a complex expression)

end Final value of counter (can be a complex expression)

step Integer amount counter is changed each time through the loop. If not specified, step defaults to one.

statements One or more statements between FOR and NEXT that are executed the specified number of times.

First the expression *start* and *end* are evaluated.

When the NEXT statement is encountered, *step* is added to the variable defined as the *counter*. At this point, if counter is less than or equal to *end*, the *statements* in the loop execute again. If *counter* is greater than *end*, then the loop is exited and execution continues with the statement following the NEXT statement.

The expressions *start*, *end* and *step* can be any expression or variable of any type. However, unlike with the WHILE statement, if *end* is an expression, the FOR statement evaluates this expression only once at the start of the loop and stores this value for subsequent comparisons. So you should not write code that relies on the *end* expression being evaluated every time through the loop.

Examples

```
FOR index = 1 TO 25
    PRINT index
NEXT
```

The following loop decrements index:

```
FOR index = -1 TO -25 STEP -1
    PRINT index
NEXT
```

You can nest FOR...NEXT loops by placing one FOR...NEXT loop within another. The following illustrates a nested FOR statement:

```
FOR monthIndex = 1 TO 12 STEP 1
    FOR dayIndex = 1 TO 31 STEP 1
```



```
print "Month = ", monthIndex, " Day = ", dayIndex
```

```
NEXT
```

```
NEXT
```

15.5 IF

If/then statements conditionally execute a group of commands, depending on the value of an expression.

Syntax

```
IF condition THEN statement [ELSE else statement]
```

Alternatively, you can use the multi-line syntax:

```
IF condition THEN
    statements
ELSE
    else statements
ENDIF
```

condition Any expression that evaluates to TRUE or FALSE

statement, Statement(s) executed IF condition is TRUE .

else statements Statement(s) executed IF condition is FALSE .

When executing the IF statement, condition is tested. If condition is TRUE , the statements following THEN are executed. If condition is FALSE , the statements following ELSE are executed. After executing the statements following THEN or ELSE, execution continues with the statement following ENDIF.

Examples

```
IF ( a = 0 ) THEN print "OK"

IF ( a = 0 ) or ( b = 0 ) THEN c = ( c - 1 ) ELSE c = ( c + 1 )

IF ( a = 0 ) THEN
    ( b = 0 )
    ( c = 0 )
ELSE
    ( b = 1 )
    ( c = 1 )
ENDIF
```

You can nest IF statements by placing one IF within another:

```
IF ( a = 0 ) THEN
    IF ( b = 0 ) THEN
        ( c = 0 )
    ELSE
        ( c = 1 )
    ENDIF
ENDIF
```

ENDIF

15.6 PRINT

Prints values to the log window (found under the Debug menu) and the print log files (see below).

Syntax

```
PRINT [expression, expression,...]
```

expression An expression to be printed, separated by commas or semicolons

PRINT without any parameters writes out a blank line.

Example

```
VARIABLES: variableOne, variableTwo TYPE: String  
  
variableOne = "Hello I am "  
variableTwo = "Bob"  
  
PRINT variableOne, variableTwo  
PRINT "Don't blame me, blame ", variableTwo
```

You can also print mathematical expressions which will be evaluated:

```
PRINT ( AbsoluteValue ( random ( 10 ) - 100 ) )
```

Printing is extremely useful for debugging. For instance, if you are trying to figure out why something won't work, can be useful to put several print statements like:

```
PRINT "The date: ", instrument.date, "Close: ", instrument.close  
PRINT "The variable in question: ", entryRiskOrSomeOtherVariable  
PRINT ""
```

You can print any expression, variable, parameter, indicator, or object.

Print Log Files

PRINT sends output to the log window and to two additional files:

```
C:\Program Files\TradingBlox\Results\PrintOutput.csv  
C:\Program Files\TradingBlox\Log Files\Normal.log
```

PrintOutput.csv is re-written every time a test is run. Normal.log stores up to 1 MB of data, so it has results from as many tests as it can hold. After it is full, it will create Normal.log.1, Normal.log.2, etc. up to 10 MB of print results.

These files are comma delimited for use in other programs.

15.7 WHILE

Executes a series of statements as long as a given condition is **TRUE**.

Syntax

```
WHILE condition
    [statements]
ENDWHILE
```

condition Expression that evaluates to **TRUE** or **FALSE**.

statements One or more statements executed while condition is True.

If *condition* is **TRUE**, all *statements* are executed until the **ENDWHILE** statement is encountered. Control then returns to the **WHILE** statement and *condition* is again checked. If *condition* is still **TRUE**, the process is repeated. If it is not **TRUE**, execution resumes with the statement following the **ENDWHILE** statement.

Unlike the **FOR** statement, the **WHILE** statement evaluates *condition* on every loop pass.

Example

```
VARIABLES: a TYPE: Integer
WHILE ( a > 0 )
    print "a = ", a
    a = ( a - 1 )
ENDWHILE
```

Section 16 – Trouble Shooting Script Problems


Section will discuss various methods for discovering problems in Trading Blox Basic script statements, and the tools Trading Blox provides to make the process of problem resolution possible.

"Work In Progress"

16.1 Debugger

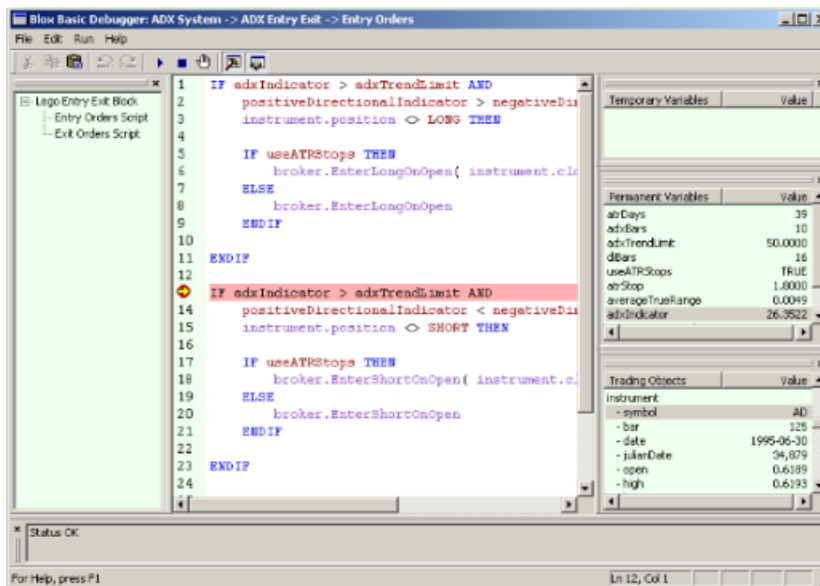
The Debugger is a powerful tool of Trading Blox. It allows you to set breakpoints in your code, and look at all the variables at that point. You set a breakpoint by clicking on a particular line of code in your Script, and pressing the Set Breakpoint button at the bottom right.

```
11  END IF
```






To clear that breakpoint you can click on the same line of code and press Clear Breakpoint. Or you can clear all breakpoints by pressing the Clear Breakpoints button. When you set a breakpoint in this manner, the default is to break on every instrument on every day.

If your breakpoint gets hit during the execution of the system, you will see a debugger window which shows you all the local, global, and object variables available to that script. It is useful in determining whether things are operating normally.

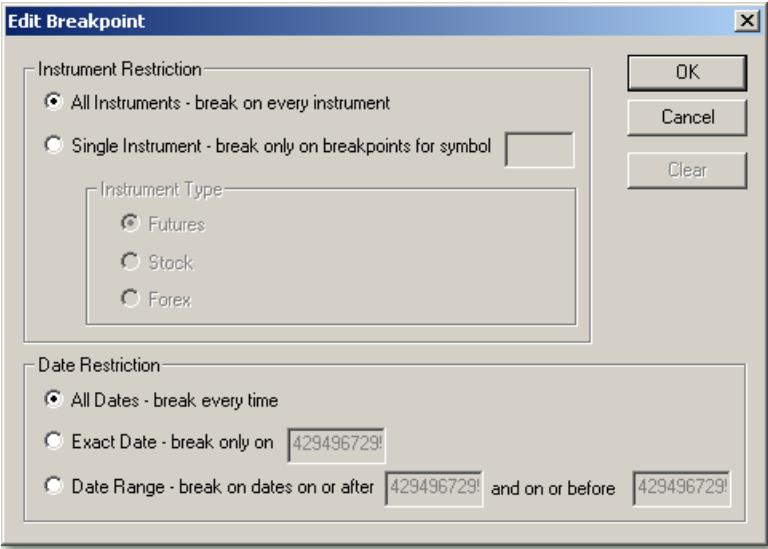


The debugger includes several different buttons on the toolbar which aid you in making sure that your code is doing what you intended. Starting from left to right we have:

-  The Run button runs the script.
-  The Stop button stops a script that is being debugged, terminating its execution.
-  The Breakpoint button toggles a breakpoint at the current line.

Use F11 to step from one line to the next.

You can also set or edit a breakpoint by double clicking on the line number. The line number is the number just in front of each line of code in your script. When you set or edit the breakpoint this way, you have more options.



You can break for all instruments or a single instrument. To enter a single instrument use the symbol. For Soybeans the symbol would be S. Use upper case. Indicate the Instrument Type, whether it is a future, stock, or forex.

You can also filter by date:

- All Dates: Break every time
- Exact Date: Breaks only on date entered (enter in format YYYYMMDD e.g. 20050704 for July 4, 2004)
- Date Range: Breaks on dates between the two dates entered

16.2 Auto-Keyword Changes

Software advancements create a need to change older features and add new abilities. Tables in this section will list the keywords changed or removed and inform which new keywords and methods can be used in their place where it is possible.

Historical information on Object properties and function changes will be listed in each of the object's changed keyword tables. Language functions and keyword changes will be listed the Trading Blox Basic Language keyword change table.

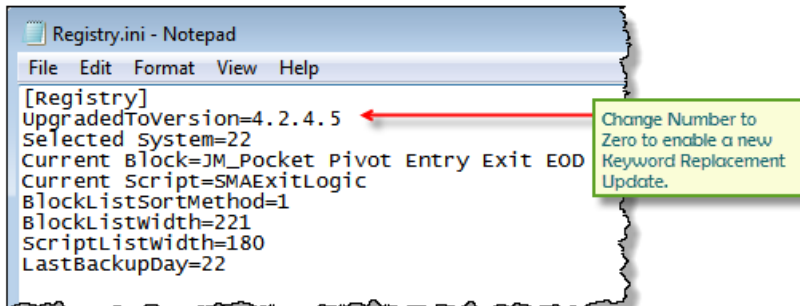
Automatic Keyword Changes

Trading Blox updates all the scripts with any scripting changes that it can easily identify. From a user perspective this means any program keyword changes discovered in the Blox modules will be replaced with a the keyword assigned as the obsoleted keyword replacement.

When a keyword is replaced the user will be prompted to save the changes. If the user declines to save changes after the first startup of a new version displays its "Do you want to save changes?" those automatic scripting updated keywords will be lost.

Modules with keywords that are no longer viable will create an error condition when that module is run because the keyword replacement process will not run again until a new version is used. However, if the user is willing to go into the Trading Blox directory and use Windows Notepad to view the "Registry.ini" file in the Trading Blox directory they can reset the control setting so the automatic keyword replacement process will run the next time Trading Blox is executed.

To reset the automatic keyword replacement process locate the "**UpgradedToVersion**" control word and reset the value to zero.



Automatic Keyword Replacement Update Control Registry Item.

When the value of the control word is below the version value of Trading Blox the automatic keyword process will execute and the Registry.ini setting will be updated regardless of whether the keyword changes were saved.

Keyword Replacement Example:

When this keyword is found:

```
instrument.futuresMonth
```

Trading Blox will remove the above keyword AND insert this keyword:

```
instrument.deliveryMonth
```

2.1 Previous KeyWord	Replacement KeyWord
----------------------	---------------------

<code>instrument.futuresMonth</code>	<code>instrument.deliveryMonth</code>
<code>instrument.totalUnits</code>	<code>instrument.currentPositionUnits</code>
<code>instrument.barsSinceEntry</code>	<code>instrument.unitBarsSinceEntry</code>
<code>instrument.totalPositionSize</code>	<code>instrument.currentPositionQuantity</code>
<code>instrument.totalPositionProfit</code>	<code>instrument.currentPositionProfit</code>
<code>instrument.totalPositionRisk</code>	<code>instrument.currentPositionRisk</code>
<code>instrument.tradeOrder</code>	<code>instrument.priorityIndex</code>
<code>test.equityDrawdown</code>	<code>test.currentDrawdown</code>
<code>test.dayNumber</code>	<code>test.currentDay</code>
2.2 Previous KeyWord	Replacement KeyWord
<code>.LoadPortfolioInstrument</code>	<code>.LoadSymbol</code>
<code>test.currentParameterRun</code>	<code>test.currentParameterTest</code>
<code>test.totalParameterRuns</code>	<code>test.totalParameterTests</code>
<code>test.AbortParameterRun</code>	<code>test.AbortTest</code>
<code>broker.EnterLongStopOpenOnly</code>	<code>broker.EnterLongOnStopOpen</code>
<code>broker.EnterShortStopOpenOnly</code>	<code>broker.EnterShortOnStopOpen</code>
2.3 Previous KeyWord	Replacement KeyWord
<code>test.generatingOrders</code>	<code>test.orderGenerationBar</code>
<code>test.totalInstruments</code>	<code>test.instrumentCount</code>
<code>system.sortInstruments</code>	<code>system.rankInstruments</code>
3.3 Previous KeyWord	Replacement KeyWord
<code>instrument.totalEquity</code>	<code>instrument.testTotalEquity</code>
3.5.5 Previous KeyWord	Replacement KeyWord
<code>system.openEquity</code>	<code>system.currentOpenEquity</code>
4.0.3 Previous KeyWord	Replacement KeyWord
<code>chart.xAxis</code>	<code>chart.setxAxisLabels</code>
<code>chart.addLine</code>	<code>chart.addLineSeries</code>

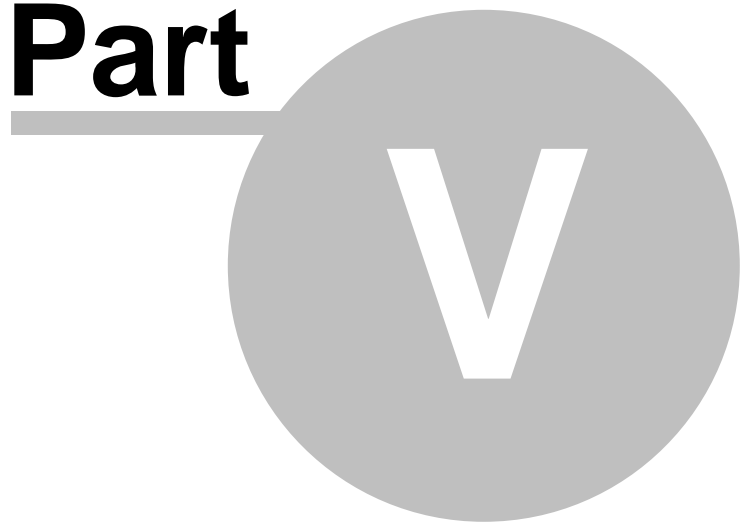
4.0.10 Previous KeyWord	Replacement KeyWord
<code>system.positionInstruments</code>	system.totalPositions
<code>system.tradingBars</code>	system.dataLoadedBars
<code>script.SetStringReturnValue</code>	script.SetReturnValue

Links:

See Also:

Trading Objects Reference

Part



Part 5 – Trading Objects Reference

Trading Objects represent real-world objects used in trading, the instrument object represents a tradeable market, the broker represents the broker who takes your orders, etc.

Blox Basic scripts use the Trading Objects to access information and to affect the trading simulation. The Trading Objects used in Trading Blox Builder are:

Object Names:	Description:
<u>Block</u>	represents the current Trading Block and is generally only used for debugging purposes
<u>Broker</u>	Broker methods are used to enter orders with their stops when protective exit prices are used, and exiting positions. Broker Entry order call the Unit Sizing script, which is followed by the Can Add Unit script. Both entry and exit orders are processed by the Can Fill Order script.
<u>Chart</u>	used to create custom charts
<u>Email Manager</u>	used to send emails from scripting
<u>FileManager</u>	used to read and write files
<u>Instrument</u>	Represents a given market, or a tradeable instrument to access pricing, position, and other information that is useful for influencing system orders and positions.
<u>Order</u>	contains information about the order used in the Can Fill Order script
<u>Script</u>	used to access custom user scripts
<u>System</u>	represents the system itself and is used to access system-level information such as the total equity
<u>Test</u>	represents the test and is used to obtain test-level information like the start and end dates

Section 1 – Alternate Objects

Alternate Objects are created for accessing data outside of the range of a system's normal object range.

Object:	Description:
AlternateBroker	AlternateBroker object is used to place orders in another system. Set the system of the alternateBroker with SetAlternateSystem. Then pass the symbol into the alternateBroker function call.
AlternateOrder	
AlternateSystem	

1.1 AlternateBroker Object

The [AlternateBroker](#) is also a broker object with all the same functions and properties.

The [AlternateBroker](#) is also a broker object with all the same functions and properties. It is brought into context by use of the [test.SetAlternateSystem](#) function. When the [alternateSystem](#) object is set, the [alternateBroker](#) object is also set to the same alternate system. In this way orders can be placed for any system, from any system, including from a Global Suite System (GSS). In a GSS the system can be looped over, and orders placed for any or all systems in the suite. From a GSS, the [instrument.symbol](#) needs to be the first parameter, as the GSS has no default instrument context.

Here is an example of placing an order for Gold (GC) in System_1. This function call can be done from a GSS or any other system in the test, and it assumes GC is in the current portfolio for system 1.

Example:

```
' Example loads Gold Future contract
If instrument.LoadSymbol( "F:GC", 1 ) THEN
  ' Set system data access to System-index 1
  test.SetAlternateSystem( 1 )
  ' When the Instrument is primed, and
  ' when the instrument's position is Flat,...
  If instrument.isPrimed AND instrument.position = OUT THEN
    ' Use the alternate Broker Object function
    ' to create an Long Entry On_Open order for Gold
    alternateBroker.EnterLongOnOpen( instrument.symbol )
    ' If the Long Entry order is created successfully,
    ' and if it didn't get rejected,...
    If alternateSystem.OrderExists() THEN
      ' Set the order quantity to 10-contracts
      order.SetQuantity( 10 )
    ENDIF
  ENDIF ' inst.isPrimed AND inst.position = OUT
ELSE
  ' Show the Gold Futures file failed to load.
  PRINT "Unable to load symbol"
ENDIF ' i.LoadSymbol
```

1.2 AlternateOrder Object

Trading Blox allows users to access order information in the script sections where the Object Order does not automatically have context by using the System's [SetAlternateOrder](#) function where each order is brought into context using the order's index value.

Once an order is in context, information about that order is made available using the Order Object's [alternateOrder](#) object prefix to access the properties and functions. This next example shows a simple approach to accessing order information in script section where order don't normally exist:

Example:

```
' Loop over the open orders setting the order sort
' value with a secret computation.
FOR orderIndex = 1 to system.totalOpenOrders STEP 1
  ' Bring order index by the 'orderIndex' Integer value.
  system.SetAlternateOrder( orderIndex )
  ' Change the order's sort value property to a random
  ' number between 1 and 100
  alternateOrder.SetSortValue( Random( 100 ) )
NEXT ' orderIndex
```

Once the order is in context other information can be accessed and changed as needed.

Notes:

Always check to be sure the order is available after a Broker function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and functions are available to reference and changes.

1.3 AlternateSystem Object

Alternate System Object is used when referencing other system that are in the same suite.

Section 2 – Block

All Block properties provide information about the module and the system in which the module is applied.

Most often these properties are used in a debugging operation so the programmer will know the source from where the debugging output is being generated.

Properties	Descriptions
Group	Group name of the Blox that was assigned by the Blox Editor
Name	Block module name being executed. When name is printed during debugging operation it shows the current script execution locations.
ScriptName	Name of the script currently being executed. Useful when printed .
System	System name where block is physically assigned during testing.
SystemIndex	System index number of the system. When only one system is located in the Suite, the system index value is always 1. When multiple systems are assigned to the Suite, the index values are assigned based upon the order in which they were first assigned to the Simulation Suite.

Within a system where these script name are listed more than once, all the scripts with the following names can execute when a new order is created, and an entry or exit order is filled:

Script Name:	Executes Timing:
CAN ADD UNIT	All New Entry Orders
CAN FILL ORDER	ALL FILLED ORDERS
ENTRY ORDER FILLED	ALL FILLED ENTRY ORDERS
EXIT ORDER FILLED	ALL FILLED EXIT ORDERS

When modules within a system have multiple instances of any of the above script names, and there is programming code in each of the multiple instances of these scripts it might be necessary to filter which orders can be processed by which module's script section using filtering logic similar to this:

Example -- CAN FILL ORDER :

```

! ~~~~~
! Reject Orders Generated by other System Modules
If order.systemBlockName = system.name + "." + block.name THEN
! Allow this module's CAN FILL ORDER script to only apply
! to orders that are generated by this Blox to do
! something in this area
ENDIF
! ~~~~~

```

Common debugging block object properties:

Example -- BEFORE ORDER EXECUTION:

```
| ~~~~~  
PRINT "Block Name:                ", block.name  
PRINT "Block Group Name:          ", block.group  
PRINT "Block Script Section Name: ", block.scriptName  
PRINT "Blox applied System Name:  ", block.system  
PRINT "System Suite Index Number: ", block.systemIndex  
| ~~~~~
```

PRINT OUTPUT:

```
Block Name:                _Max_Position_Limiter 3  
Block Group Name:         _Dev  
Block Script Section Name: Before Order Execution  
Blox applied System Name: Bollinger Breakout Plus  
System Suite Index Number: 2
```

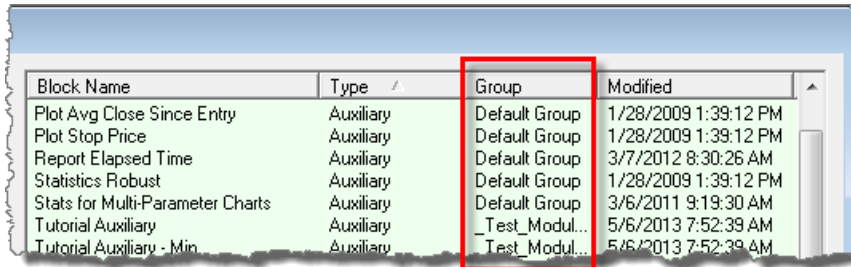
Links:

[lineNumber](#)

See Also:

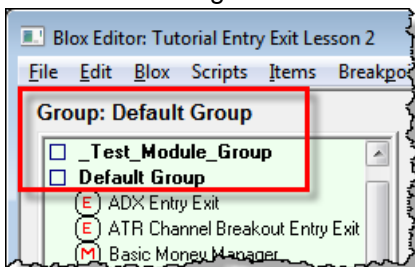
2.1 Group

Blox are assigned a **Group-Name** when they are created, or changed. Name returned is the name shown in the **Group Name** column when shown in the **System Editor** listing of all blox modules.



Block Name	Type	Group	Modified
Plot Avg Close Since Entry	Auxiliary	Default Group	1/28/2009 1:39:12 PM
Plot Stop Price	Auxiliary	Default Group	1/28/2009 1:39:12 PM
Report Elapsed Time	Auxiliary	Default Group	3/7/2012 8:30:26 AM
Statistics Robust	Auxiliary	Default Group	1/28/2009 1:39:12 PM
Stats for Multi-Parameter Charts	Auxiliary	Default Group	3/6/2011 9:19:30 AM
Tutorial Auxiliary	Auxiliary	_Test_Modul...	5/6/2013 7:52:39 AM
Tutorial Auxiliary - Min	Auxiliary	_Test_Modul...	5/6/2013 7:52:39 AM

It is also the same name shown in the **Blox Editor** groups section where the blox module is made available for editing.



Syntax:

`block.group`

Parameter:

<none>

Description:

Example:

```
~~~~~
PRINT "Block Group Name           ", block.group
~~~~~
```

Returns:

Block Group Name: _dev

Links:

See Also:

2.2 Name

Actual name assigned to the block that is in the system.

Syntax:

`block.name`

Parameter:

<none>

Description:**Example:**

```
| ~~~~~  
| PRINT "Block Name:           ", block.name  
| ~~~~~
```

Returns:

Block Name: Max_Position_Limiter 3

Links:**See Also:**

2.3 ScriptName

Name of script section where this `scriptName` property is being used.

When block properties are placed in a custom script section, the name returned by the `scriptName` property will be the script name in which the `script.Execute("CustomScript")` is called.

Syntax:

```
block.scriptName
```

Parameter:

<none>

Description:

Example:

```
! ~~~~~  
PRINT "Block Script Section Name: ", block.scriptName  
! ~~~~~
```

Returns:

```
Block Script Section Name: Before Order Execution
```

Links:

[Execute](#), [Script](#)

See Also:

2.4 System

Name of the [System](#) to which the blox module is attached and displayed in the **System Editor** module listing.

Syntax:

```
block.system
```

Parameter:

<none>

Description:**Example:**

```
! ~~~~~  
PRINT "Blox applied System Name  ", block.system  
! ~~~~~
```

Returns:

```
Blox applied System Name  Bollinger Breakout Plus
```

Links:

[System](#)

See Also:

2.5 SystemIndex

[System](#) index reports the index assignment of the system in a suite.

Systems are assigned an index value based upon the order in which they are selected to be included in a suite. Where this is only 1 system in a suite, the system index will always be one. When there are more than one system in a suite, the first system selected to be included the suite will be given the index value of 1, the second system selected will be given an index value of 2, and the last system selected will be given the index value that matches the number of systems attached to the suite.

Suites can become a Global System (GSS) when there is a system name that is the same as the suite's name. Modules attached to a GSS will be reported as having a system index of 0.

Scripts within the GSS modules execute ahead or behind the system index sequencing order. Review the details here to understand when GSS are executed: [Global Script Timing](#)

Syntax:

```
block.systemIndex
```

Parameter:

<none>

Description:

Example:

```
' ~~~~~
PRINT "System Suite Index Number ", block.systemIndex
' ~~~~~
```

Returns:

```
System Suite Index Number 2
```

Links:

See Also:

Section 3 – Broker

Broker object contains three classes of functions:

Class Type:	Description:
Entry Order	Required to create a new position, add a unit to an existing position, or reverse the direction of position.
Exit Order	Required to close out a position, remove a unit from a position, remove a set quantity of a position.
Position Adjustment	Position adjustment can add quantity by adding a unit, remove a unit, or reduce the quantity of a position by removing some of the quantity in a unit, some of the units when a larger quantity needs to be removed than is available in a unit, and this function can also apply a quantity removal percentage that will terminate a position when the remaining quantity is less than the smallest possible trade size of that instrument.

Each Broker function selection determines an order's type and execution requirements:

Type-of-Execution:	Type-of-Order:
at Market	Long Entry
on Open	Short Entry
on Close	Long Exit
on Stop	Short Exit
on Stop Open	
on Stop Close	
on Limit	
on Limit Open	
at Limit Close	

Notes:

When a Broker function is executed, it starts the process of assembling the information the software will need to determine if the order can succeed, or fail, with the current market information.

All Trading Blox orders are "Day-Orders." This means that once they are tested against the market's information they must be enabled, or rejected. Rejected orders are canceled and removed from access.

When an order is rejected it is no longer available and it must be recreated with another Broker function execution so a new order will be available for the next test-date market information. Recreating the order assume the system requires an order for the next market record.

Entry orders to reverse a position will exit a position trading in the opposite direction to the new entry order. In testing this works well, but in brokerage most electronic orders require an order to exit and a different order to enter in the opposite direction. This means that orders reversing direction in systems that will be traded should generate an exit order and a new entry order to ensure the brokerage follows the system's intent.

It is important at this stage to get a solid understanding of the order creation process. Click on this link [Order Object](#), if you have not studied how orders are created and processed.

Risk and Broker Orders

Protective exit price Stop-orders are used to calculate the risk of trades. Risk is determined by difference between the order price and the protective exit stop price of an active position. This difference creates a position point spread that is part of the risk/reward calculations, and is sometimes used with position sizing in the Money Manager. For instance, the Fixed Fractional Money Manager uses the entry risk provided in a new entry order to determine the unit's quantity. If you have no stops, undefined risk is assumed, and there is insufficient information to complete a Fixed Fractional calculation required for this sizing method.

When new entry orders are generated without any risk point spread, a different method for determining size must be used. For example, the "Multi Money Manager" money manager blox allows the trader to select volatility sizing approach so as to create an estimate of the new entry order risk in order for a fractional sizing calculation to be successful. There are many other methods for determining size that can be found in the postings in the Trading Blox forum.

Example:

```
' Typical broker statement to enter
' long at next market open without
' any protection price.
broker.EnterLongOnOpen           ' Places an order to buy at
                                  ' the open with no stop.

' Typical broker statement to enter
' long at next market open with a
' protection price.
broker.EnterLongOnOpen( stopPrice ) ' Places an order to buy at
                                      ' the open with a protective
                                      ' stop at stopPrice.
```

In the second broker example where the order's execution type is to enter on the next open, the risk basis price uses the difference from the current date Close price to the protective "stopPrice" to determine the risk of a single contract or share purchase. This is also the same process used with active positions that use protective exit prices to determine the risk amount of each contract in the position. When multiple units are used, the point spread between the Close-price to each unit's protective price is used to estimate each unit's risk. Multiple unit positions then sum the total risk of all units to create the instrument's [currentPositionRisk](#).

The quantity of each order is determined by the Money Manager. The Broker object calls the Money Manager, and the Money Manager sets the quantity for each order. If there is no Money Manager Block in the system, the unitSize defaults to 0. You must set the order quantity using [order.SetQuantity\(\)](#) in the Unit Size script, or the size will be 0, which will result in a trade that has no effect in your testing.

When a protective stop price is used in the entry order, that value is saved with the instrument and can be accessed using the instrument's [unitExitStop](#) property. The stop order itself, however, is only placed for the entry price bar. To 'hold' the stop and place it in the market every day, use the following code in your Exit Orders script:

Example:

```
' Protective Single-Unit Exit Order
broker.ExitAllUnitsOnStop( instrument.unitExitStop )
```

Many of our built-in systems use the above method to "hold" stops (keep a protective order in the market).

Multiple Units require an order for each unit when units are treated independently:

Example:

```
' . . . . .
' Update Long Position's Multiple Unit Protection
For UnitNum = 1 TO instrument.currentPositionUnits
' Use Max of New-High Offset, Previous Unit Exit,
' Or Position's Initial Entry Protection Price
Exit_Price = Max( TestExitPrice, _
                  instrument.unitExitStop[1], _
                  Money_Stop )

' Update Unit's Protective Exit Price Property
instrument.SetExitStop( UnitNum, Exit_Price )

' Generate an Protective Exit Order for this Unit
broker.ExitUnitOnStop( UnitNum, Exit_Price )
Next ' UnitNum
' . . . . .
```

Price-Record Processing:

Use the Entry Day Retracement parameter to adjust how entry day stops are processed. A setting of 100% is the most conservative, a setting of 0% is the least conservative, and a setting of -1 will disable entry day stops.

Symbol as the First Parameter:

All broker functions will execute an order for the current default instrument in context. When an order is intended for an instrument that isn't by default in the script at that time, the broker function's first parameter can be an instrument **symbol**. Applying a symbol that is different than the instrument in context will apply the order to instrument specified in the broker function. When no symbol is provided the broker function will apply the order to the current instrument in context.

When specifying any instrument out of its normal context assignments it is important to check the [instrument.tradesOnTradeDate](#) property to know if there is a new record available, or a holiday omission in the instrument's data. This practice is just as important when manually looping over instruments using the [LoadSymbol](#) function, be sure new calculations with missing data are not distorting previously correct instrument information.

Example:

```
' Create an Entry On_Open order using symbol
' when Instruments are out of context
broker.EnterLongOnOpen( useThisSymbol )
```

AlternateBroker:

When the [Broker Object](#) is out of the context of its default context scripts, the [AlternateBroker](#)

object should be used to execute any of the [Broker Object](#) Entry and Exit Functions. [AlternateBroker](#) has the same functions and properties as the [Broker Object](#) and will work in the same way.

Links:

[AlternateObject Object](#), [AlternateBroker Object](#), [Order Object](#)

See Also:

[Data Groups and Types](#)

3.1 Entry Order Functions

Entry Order functions are most often executed from within the Entry Orders script section of the system's [Entry Block](#). These Broker functions are the methods that generate the orders that create new positions.

Market On Open Orders	Descriptions:
EnterLongOnOpen	Buy on the open
EnterShortOnOpen	Short on the open
Stop Open Only Orders	
EnterLongOnStopOpen	Buy on the open if market is above/equal specified price
EnterShortOnStopOpen	Short on the open if market is below/equal specified price
Limit Open Only Orders	
EnterLongAtLimitOpen	Buy on the open if the market is below specified price
EnterShortAtLimitOpen	Short on the open if the market is above specified price
Stop Orders	
EnterLongOnStop	Buy any time if the market hits specified price
EnterShortOnStop	Short any time if the market hits specified price
Limit Orders	
EnterLongAtLimit	Buy any time if the market dips below specified price
EnterShortAtLimit	Short any time if the market climbs above specified price
Market On Close Orders	
EnterLongOnClose	Buy on the close
EnterShortOnClose	Short on the close
Stop Close Only Orders	
EnterLongOnStopClose	Buy on the close if market is above/equal specified price
EnterShortOnStopClose	Short on the close if market is below/equal specified price
Limit on Close Orders	

EnterLongAtLimitClose	Buy on close if close is below the specified price
EnterShortAtLimitClose	Short on close if close is above the specified price

AlternateBroker:

When the [Broker Object](#) is out of the context of its default context scripts, the [AlternateBroker](#) object should be used to execute any of the [Broker Object](#) Entry and Exit Functions. [AlternateBroker](#) has the same functions and properties as the [Broker Object](#) and will work in the same way.

EnterLongOnOpen

Enters a long position on the next open. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterLongOnOpen( [protectStopPrice] )
```

Parameter:

protectStopPrice

Description:

Value of the protect Exit Stop price to be used in case the market goes against the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
if system.OrderExists() THEN
'   Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF '   s.OrderExists
' ~~~~~
```

Example:

```
' Enter the market on the next open.
broker.EnterLongOnOpen( protectStopPrice )
OR
' Enter the market on the next open with no stop
broker.EnterLongOnOpen
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterShortOnOpen

Enters a short position on the next open. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterShortOnOpen( [protectStopPrice] )
```

Parameter:

```
protectStopPrice
```

Description:

Value of the protect Exit Stop price to be used in case the market goes against the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF '   s.OrderExists
' ~~~~~
```

Example:

```
' Enter the market on the next open.
broker.EnterShortOnOpen( protectStopPrice )
```

OR

```
' Enter the market on the next open with no stop
broker.EnterShortOnOpen
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterLongOnStopOpen

Enters a long position on the next open if the open is greater than or equal to the specified price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterLongOnStopOpen( stopPrice [, protectStopPrice] )
```

Parameter:	Description:
stopPrice	Stop order price
protectStopPrice	Value of the protect Exit Stop price to be used in case the market goes against the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
broker.EnterLongOnStopOpen( entryPrice, protectStopPrice )
```

OR

```
broker.EnterLongOnStopOpen( entryPrice )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterLongAtLimitOpen

Enters a long position on the next open if it is lower than the specified price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterLongAtLimitOpen( limitPrice [, protectStopPrice] )
```

Parameter:

limitPrice

Description:

Price which the market must be lower in order to trigger this order

protectStopPrice

Value of the protect Exit Stop price to be used in case the market goes against the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Enter the market on the next open if below the entry price.
broker.EnterLongAtLimitOpen( entryPrice, protectStopPrice )
```

OR

```
' Enter the market on the next open if below entry price with no stop
broker.EnterLongAtLimitOpen( entryPrice )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterShortOnStopOpen

Enters a short position on the next open if it is lower than or equal to the specified price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterShortOnStopOpen( stopPrice [, protectStopPrice] )
```

Parameter:	Description:
stopPrice	Stop order price
protectStopPrice	Value of the protect Exit Stop price to be used in case the market goes against the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
if system.orderExists() THEN
'   Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.orderExists
' ~~~~~
```

Example:

```
' Enter the market on the next open if at or below the entry price.
broker.EnterShortOnStopOpen( entryPrice, protectStopPrice )
```

OR

```
' Enter the market on the next open if at or below entry price with no s
broker.EnterShortOnStopOpen( entryPrice )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterShortAtLimitOpen

Enters a short position on the next open if it is higher than the specified price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterShortAtLimitOpen( limitPrice [, protectStopPrice] )
```

Parameter:	Description:
limitPrice	Price which the market must be higher in order to trigger this order
protectStopPrice	Value of the protect Exit Stop price to be used in case the market goes against the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
if system.orderExists() THEN
'   Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.orderExists
' ~~~~~
```

Example:

```
' Enter the market on the next open if above the entry price.
broker.EnterShortAtLimitOpen( entryPrice, protectStopPrice )
```

OR

```
' Enter the market on the next open if above entry price with no stop
broker.EnterShortAtLimitOpen( entryPrice )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterLongOnStop

Enters a long position if the next bar's high is greater than or equal to the order price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterLongOnStop( stopPrice [, protectStopPrice] )
```

Parameter:	Description:
stopPrice	Stop order price
protectStopPrice	Value of the protect Exit Stop price to be used in case the market goes against the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
' Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

' Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
broker.EnterLongOnStop( entryPrice, protectStopPrice )
```

OR

```
broker.EnterLongOnStop( entryPrice )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterShortOnStop

Enters a short position if the next bar's low is lower than or equal to the order price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterShortOnStop( stopPrice [, protectStopPrice] )
```

Parameter:

stopPrice

Description:

Stop order price

protectStopPrice

Value of the protect Exit Stop price to be used in case the market goes against the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
broker.EnterShortOnStop( entryPrice, protectStopPrice )
```

OR

```
broker.EnterShortOnStop( entryPrice )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterLongAtLimit

Enters a long position if the next bar's low is lower than the order price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterLongAtLimit( limitPrice [, protectStopPrice] )
```

Parameter:

limitPrice

Description:

Limit order price

protectStopPrice

Value of the protect Exit Stop price to be used in case the market goes against the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
broker.EnterLongAtLimit( priceTarget, protectStopPrice )
```

OR

```
broker.EnterLongAtLimit( priceTarget )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterShortAtLimit

Enters a short position if the next bar's high is greater than the order price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterShortAtLimit( limitPrice [, protectStopPrice] )
```

Parameter:	Description:
limitPrice	Limit order price
protectStopPrice	Value of the protect Exit Stop price to be used in case the market goes against the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
' Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

' Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
broker.EnterShortAtLimit( priceTarget, protectStopPrice )
```

OR

```
broker.EnterShortAtLimit( priceTarget )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterLongOnClose

Enters a long position on the next close. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterLongOnClose( [ protectStopPrice] )
```

Parameter:

```
protectStopPrice
```

Description:

This sets the protective stop price of the order, and the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Note:

Orders with a price that is close are excluded from same day exits, so this protective stop is not used on the day of entry.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
' Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

' Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Enter the market on the next close.
broker.EnterLongOnClose
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterShortOnClose

Enters a short position on the next close. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterShortOnClose( [ protectStopPrice] )
```

Parameter:

protectStopPrice

Description:

This sets the protective stop price of the order, and the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Note:

Orders with a price that is close are excluded from same day exits, so this protective stop is not used on the day of entry.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
' Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

' Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:**Links:**

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterLongOnStopClose

Enters a long position if the close is at or above the specified price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterLongOnStopClose( stopPrice, [ protectStopPrice] )
```

Parameter:

stopPrice

Description:

Entry stop price of the order

protectStopPrice

This sets the protective stop price of the order, and the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Note:

Orders with a price that is close are excluded from same day exits, so this protective stop is not used on the day of entry.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Enter the market if the close is at or above the entry price
broker.EnterLongOnStopClose( entryPrice )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterShortOnStopClose

Enters a short position if the close is at or below than the specified price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterShortOnStopClose( stopPrice, [ protectStopPrice] )
```

Parameter:

stopPrice

Description:

Entry stop price of the order

protectStopPrice

This sets the protective stop price of the order, and the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Note:

Orders with a price that is close are excluded from same day exits, so this protective stop is not used on the day of entry.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Enter the market if the close is at or below the entry price
broker.EnterShortOnStopClose( entryPrice )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterLongAtLimitClose

Enters a long position if the close of the next bar trades through the specified price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterLongAtLimitClose( limitPrice, [ protectStopPrice] )
```

Parameter:

limitPrice

Description:

Entry limit order price

protectStopPrice

This sets the protective stop price of the order, and the position (optional parameter when a protective exit price isn't wanted for the bar of entry).

Note:

Orders with a price that is close are excluded from same day exits, so this protective stop is not used on the day of entry.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Enter the market if the close trades through the price target
broker.EnterLongAtLimitClose( priceTarget )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

EnterShortAtLimitClose

Enters a short position if the close trades through the specified price. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:

```
broker.EnterShortAtLimitClose( limitPrice, [ protectStopPrice] )
```

Parameter:	Description:
limitPrice	Entry limit order price
protectStopPrice	This sets the protective stop price of the order, and the position (optional parameter when a protective exit price isn't wanted for the bar of entry). Note: Orders with a price that is close are excluded from same day exits, so this protective stop is not used on the day of entry.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Enter the market if the close trades through the price target
broker.EnterShortAtLimitClose( priceTarget )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

3.2 Exit Order Functions

Exit Order functions are most often used in the Exit Orders script section in a system's [Exit Block](#). These Broker functions are the primary method that generate the exit orders for reducing the number of units in a position, reducing the quantity in a unit, providing in the market protective order prices, or creating orders that provide target price exits.

Market On Open Orders	Descriptions:
ExitAllUnitsOnOpen	Exit all units on the open
ExitUnitOnOpen	Exit the specified unit on the open
Stop Open Only Orders	
ExitAllUnitsOnStopOpen	Exit all units on the open if market hits specified price
ExitUnitOnStopOpen	Exit the specified unit on the open if hits specified price
Limit Open Only Orders	
ExitAllUnitsAtLimitOpen	Exit all units on the open if market exceeds price
ExitUnitAtLimitOpen	Exit specified unit on the open if the market exceeds price
Stop Orders	
ExitAllUnitsOnStop	Exit all units any time if the market hits specified price
ExitUnitOnStop	Exit the specified unit any time if the market hits specified price
Limit Orders	
ExitAllUnitsAtLimit	Exit all units any time if the market exceeds specified price
ExitUnitAtLimit	Exit the specified unit any time if the market exceeds specified price
Market On Close Orders	
ExitAllUnitsOnClose	Exit all units on the close
ExitUnitOnClose	Exit the specified unit on the close
Stop Close Only Orders	
ExitAllUnitsOnStop	Exit all units on the close if market hits specified price

pClose	
ExitUnitOnStopClose	Exit the specified unit on the close if market hits specified price
Limit on Close Orders	
ExitAllUnitsAtLimitClose	Exit all units on close if the market exceeds the specified price
ExitUnitAtLimitClose	Exit the specified unit on close if the market exceeds the specified price

AlternateBroker:

When the [Broker Object](#) is out of the context of its default context scripts, the [AlternateBroker](#) object should be used to execute any of the [Broker Object](#) Entry and Exit Functions. [AlternateBroker](#) has the same functions and properties as the [Broker Object](#) and will work in the same way.

ExitAllUnitsOnOpen

Exits all units for the current instrument on the next open. This function is generally used by an [Exit Block](#) to close out a position.

Syntax:

```
broker.ExitAllUnitsOnOpen
```

Parameter:

none

Description:

Function does not take any parameters.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
if system.orderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.orderExists
' ~~~~~
```

Example:

```
' Exit the market on the next open.
broker.ExitAllUnitsOnOpen
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitUnitOnOpen

Exits the specified unit for the current instrument on the next open. This function is generally used by an [Exit Block](#) to lighten up a position.

Syntax:

```
broker.ExitUnitOnOpen( unitNumber, [ quantity ] )
```

Parameter:

unitNumber

Description:

Unit # to exit.

quantity

Optional quantity for a partial exit of the unit

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
' Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

' Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Exit the first unit on the next open.
broker.ExitUnitOnOpen( 1 )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitAllUnitsOnStopOpen

Exits all units for the current instrument on the next open if it is lower than the stop price for long positions or higher than the stop price for short positions. This function is generally used by an [Exit Block](#) to close out a position.

Syntax:

```
broker.ExitAllUnitsOnStopOpen( stopPrice )
```

Parameter:

stopPrice

Description:

Protective Price at which the open must exceed a price (above/below) to trigger this order.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Exit all units on the next open if it hits our stop.
broker.ExitAllUnitsOnStopOpen( exitStop )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitAllUnitsAtLimitOpen

Exits all units for the current instrument on the next open if it is higher than the limit price for long positions or lower than the limit price for short positions. This function is generally used by an [Exit Block](#) to close out a position.

Syntax:

```
broker.ExitAllUnitsAtLimitOpen( limitPrice )
```

Parameter:

limitPrice

Description:

A price at which the open must exceed (above/below) to trigger this order.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
if system.orderExists() THEN
'   Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.orderExists
' ~~~~~
```

Example:

```
' Exit all units on the next open if it trades through our limit.
broker.ExitAllUnitsAtLimitOpen( limitPrice )
```

Links:

[Broker, Exit Order Functions](#)

See Also:

ExitUnitOnStopOpen

Exits the specified unit for the current instrument on the next open if it is lower than the stop price for long positions or higher than the stop price for short positions. This function is generally used by an [Exit Block](#) to lighten up a position.

Syntax:

```
broker.ExitUnitOnStopOpen( unitNumber, stopPrice )
```

Parameter:

unitNumber

Description:

Unit number to exit.

stopPrice

A price at which the open must exceed (above/below) to trigger this order.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Exit the first unit on the next open if it hits our stop.
broker.ExitUnitOnStopOpen( 1, exitStop )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitUnitAtLimitOpen

Exits the specified unit for the current instrument on the next open if it is higher than the limit price for long positions or lower than the limit price for short positions. This function is generally used by an [Exit Block](#) to lighten up a position.

Syntax:

```
broker.ExitUnitAtLimitOpen( unitNumber, limitPrice )
```

Parameter:

unitNumber

Description:

Unit number to exit

limitPrice

A price at which the open must exceed (above/below) to trigger this order.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
  ' Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

  ' Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Exit the first unit on the next open if it trades through our limit.
broker.ExitUnitAtLimitOpen( 1, exitLimit )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitAllUnitsOnStop

Exits all units for the current instrument if the price during the next bar goes lower than the stop price for long positions or higher than the stop price for short positions. This function is generally used by an [Exit Block](#) to close out a position.

Syntax:

```
broker.ExitAllUnitsOnStop( stopPrice )
```

Parameter:

stopPrice

Description:

A price at which the next bar must exceed (above/below) to trigger this order

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Exit all units during the next bar if it hits our stop.
broker.ExitAllUnitsOnStop( exitStop )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

WWW.TRADING-SOFTWARE-DOWNLOAD.COM

ExitUnitOnStop

Exits the specified unit for the current instrument on the next bar if the market goes lower than the stop price for long positions or higher than the stop price for short positions. This function is generally used by an [Exit Block](#) to lighten up a position.

Syntax:

```
broker.ExitUnitOnStop( unitNumber, stopPrice, [ quantity ] )
```

Parameter:	Description:
unitNumber	Unit to exit
stopPrice	A price at which the next bar must exceed (above/below) to trigger this order
quantity	Exit option to remove a partial quantity from the unit. When left blank, entire unit will be removed.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
' Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

' Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Exit the first unit if the market hits our stop.
broker.ExitUnitOnStop( 1, exitStop )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitAllUnitsAtLimit

Exits all units for the current instrument if the price trades through the limit price. This function is generally used by an [Exit Block](#) to close out a position.

Syntax:

```
broker.ExitAllUnitsAtLimit( limitPrice )
```

Parameter:

limitPrice

Description:

A price at which the market must trade through to trigger this order.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
if system.orderExists() THEN
'   Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.orderExists
' ~~~~~
```

Example:

```
' Exit all units during the next bar if it trades through our target.
broker.ExitAllUnitsAtLimit( limitPrice )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitUnitAtLimit

Exits the specified unit for the current instrument on the next bar if the market trades through the specified limit price. This function is generally used by an [Exit Block](#) to lighten up a position.

Syntax:

```
broker.ExitUnitAtLimit( unitNumber, limitPrice, [ quantity ] )
```

Parameter:**Description:**

unitNumber	Unit number to exit
limitPrice	A price a which the next bar must trade through to trigger this order.
quantity	Exit option to remove a partial quantity from the unit. When left blank, entire unit will be removed.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
' Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

' Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Exit the first unit if the market trades through our target.
broker.ExitUnitAtLimit( 1, limitPrice )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitAllUnitsOnClose

Exits all units for the current instrument on the next close. This function is generally used by an [Exit Block](#) to close out a position.

Syntax:

```
broker.ExitAllUnitsOnClose
```

Parameter:

none

Description:

This function does not take any parameters.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
if system.orderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.orderExists
' ~~~~~
```

Example:

```
' Enter the market on the next open.
broker.ExitAllUnitsOnClose
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitUnitOnClose

Exits the specified unit for the current instrument on the next close. This function is generally used by an [Exit Block](#) to lighten up a position.

Syntax:

```
broker.ExitUnitOnClose( unitNumber )
```

Parameter:

unitNumber

Description:

Unit number to exit on next trade day close.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Exit the first unit on the next close.
broker.ExitUnitOnClose( 1 )
```

Links:

[Broker, Exit Order Functions](#)

See Also:

ExitAllUnitsOnStopClose

Exits all units for the current instrument on the next close if it is lower than the stop price for long positions or higher than the stop price for short positions. This function is generally used by an [Exit Block](#) to close out a position.

Syntax:

```
broker.ExitAllUnitsOnStopClose( stopPrice )
```

Parameter:

stopPrice

Description:

A price at which the next bar must exceed (above/below) to trigger this order

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
  ' Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

  ' Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Exit all units on the next close if it hits our stop.
broker.ExitAllUnitsOnStopClose( exitStop )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitUnitOnStopClose

Exits the specified unit for the current instrument on the next close if it is lower than the stop price for long positions or higher than the stop price for short positions. This function is generally used by an [Exit Block](#) to lighten up a position.

Syntax:

```
broker.ExitUnitOnStopClose( unitNumber, stopPrice )
```

Parameter:

unitNumber

Description:

Unit number to exit.

stopPrice

A price at which the next bar must exceed (above/below) to trigger this order

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
if system.orderExists() THEN
'   Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.orderExists
' ~~~~~
```

Example:

```
' Exit the first unit on the next close if it hits our stop.
broker.ExitUnitOnStopClose( 1, exitStop )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitAllUnitsAtLimitClose

Exits all units for the current instrument if the close trades through the limit price. This function is generally used by an [Exit Block](#) to close out a position.

Syntax:

```
broker.ExitAllUnitsAtLimitClose( limitPrice )
```

Parameter:

limitPrice

Description:

A price a which the next bar must trade through to trigger this order.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
if system.orderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.orderExists
' ~~~~~
```

Example:

```
' Exit all units during the next bar if it trades through our limit price
broker.ExitAllUnitsAtLimitClose( limitPrice )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

ExitUnitAtLimitClose

Exits the specified unit for the current instrument on the next close if it trades through the specified limit price. This function is generally used by an [Exit Block](#) to lighten up a position.

Syntax:

```
broker.ExitUnitAtLimitClose( unitNumber, limitPrice )
```

Parameter:	Description:
unitNumber	Unit number to exit.
limitPrice	A price a which the next bar must trade through to trigger this order.

Returns:

When a broker function succeeds it will place a True in the `system.orderExists()` and when it fails to this property will return a False. Before attempting to access any order information expected from a broker order, consider checking to confirm an order was created with a conditional statement similar to this:

```
' ~~~~~
' When New Order is Created,...
If system.OrderExists() THEN
' Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel)

' Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel)
ENDIF ' s.OrderExists
' ~~~~~
```

Example:

```
' Exit the first unit if the close trades through our target.
broker.ExitUnitAtLimitClose( 1, limitPrice )
```

Links:

[Broker](#), [Exit Order Functions](#)

See Also:

3.3 Position Adjustment Functions

Position adjustment functions are most often used in the **Adjust Instrument Risk** script section of a [Risk Manager Block](#).

When these functions are called they will reduce or increase the quantity in a position by reducing the quantity in a single unit, or in a group of units with multiple unit positions when a larger quantity is required to be removed than what is available in a single unit.

As quantities in any of the units are reduced to zero those unit are terminated.

As the quantities of a position are increased a unit will be added to a position to contain the added quantity, and an incremental unit number will appear.

Adjust Size On Open Order	Descriptions:
AdjustPositionOnOpen	Adjusts the position on the open
Adjust Size On Stop Order	
AdjustPositionOnStop	Adjusts the position any time if the market hits specified price
Adjust Size At Limit Order	
AdjustPositionAtLimit	Adjusts the position on any time if the market goes beyond specified price
Adjust Size On Close Order	
AdjustPositionOnClose	Adjusts the position on the close

AlternateBroker:

When the [Broker Object](#) is out of the context of its default context scripts, the [AlternateBroker](#) object should be used to execute any of the [Broker Object](#) Entry and Exit Functions. [AlternateBroker](#) has the same functions and properties as the [Broker Object](#) and will work in the same way.

AdjustPositionOnClose

Increases or decreases an existing position by the specified factor as of the close.

- Increasing a position size will result in the addition position units since the contract/share additions will have a different entry date than any of the existing units.
- Decreasing a position size will remove contracts/shares starting with the last unit on, and working back through the remaining units until enough quantity has been removed.

For instance:

```
broker.AdjustPositionOnClose( 1.4 )
```

would increase a position by 40%, while

```
broker.AdjustPositionOnClose( .8 )
```

would decrease the position by 20%.

This function is generally used by a [Risk Manager Block](#) to lighten a position to meet certain risk restrictions.

Syntax:

```
broker.AdjustPositionOnClose( adjustmentFactor )
```

Parameter:

adjustmentFactor

Description:

Factor by which the current position quantity will be multiplied. Result will determine position size after adjustments have been processed.

Example:

```
' Reduce the position size by our computed adjustment.
broker.AdjustPositionOnClose( riskAdjustment )
```

Links:

[Broker](#)

See Also:

[Risk Manager Block](#)

AdjustPositionOnOpen

Increases or decreases an existing position by the specified factor as of the next bar's open.

For instance:

```
broker.AdjustPositionOnOpen( 1.4 )
```

would increase a position by 40%, while

```
broker.AdjustPositionOnOpen( .8 )
```

would decrease the position by 20%.

Increasing a position size will result in adding units since the contract/share additions will have a different entry date than any of the existing units. Decreasing a position size will remove contracts/shares starting with the last unit on, and working back to the first if necessary.

This function is generally used by a [Risk Manager Block](#) to lighten a position to meet certain risk restrictions.

Syntax:

```
broker.AdjustPositionOnOpen( adjustmentFactor )
```

Parameter:	Description:
adjustmentFactor	Factor by which the current position quantity will be multiplied. Result will determine position size after adjustments have been processed.

Example:

```
' Reduce the position size by our computed adjustment.
broker.AdjustPositionOnOpen( riskAdjustment )
```

Links:

[Broker](#)

See Also:

[Risk Manager Block](#)

AdjustPositionOnStop

Increases or decreases an existing position by the specified factor, if the market hits the stop price. See: [AdjustPositionOnOpen](#) for a more complete description of the adjustment factor.

Increasing a position size will result in adding units since the contract/share additions will have a different entry date than any of the existing units. Decreasing a position size will remove contracts/shares starting with the last unit on, and working back to the first if necessary.

This function is generally used by a [Risk Manager Block](#) to lighten a position to meet certain risk restrictions.

Syntax:

```
broker.AdjustPositionOnStop( [symbol], adjustmentPercent, stopPrice )
```

Parameter:	Description:
[symbol]	Symbol is optional when intended broker order is for the instrument naturally in context.
adjustmentPercent	the factor which will be multiplied by the existing position quantities to arrive at the new unit sizes.
stopPrice	the price which the market must hit to trigger an adjustment.

Example:

```
' Adjust the position size by our computed adjustment when Stop price :
broker.AdjustPositionOnStop( 0.75, stopPrice )
```

Links:

[Broker](#)

See Also:[AdjustPositionOnOpen](#)**AdjustPositionAtLimit**

Increases or decreases an existing position by the specified factor if the market trades through the limit price. See: [AdjustPositionOnOpen](#) for a more complete description of the adjustment factor.

Increasing a position size will result in adding units since the contract/share additions will have a different entry date than any of the existing units. Decreasing a position size will remove contracts/shares starting with the last unit on, and working back to the first if necessary.

This function is generally used by a [Risk Manager Block](#) to lighten a position to meet certain risk restrictions, or by an [Exit Block](#) to take profits on a portion of a position at a specified profit target.

Syntax:

```
' Change the positions quantity using the adjustment percent
' when limit price is traded through.
broker.AdjustPositionAtLimit([symbol], adjustmentPercent, limitPrice )
```

Parameter:	Description:
[symbol]	Symbol is optional when intended broker order is for the instrument naturally in context.
adjustmentPercent	Percentage rate multiplier applied the existing position quantity to determine new position size.
limitPrice	Trade through limit price required to change position size.

Returns:

Adjust Position will add a unit when percentage factor increases position quantity, and it will reduce units greater than one unit when reducing position quantity.

Example:

```
' Take profits on a portion of our position at our target.
broker.AdjustPositionAtLimit( riskAdjustment, profitTarget )
```

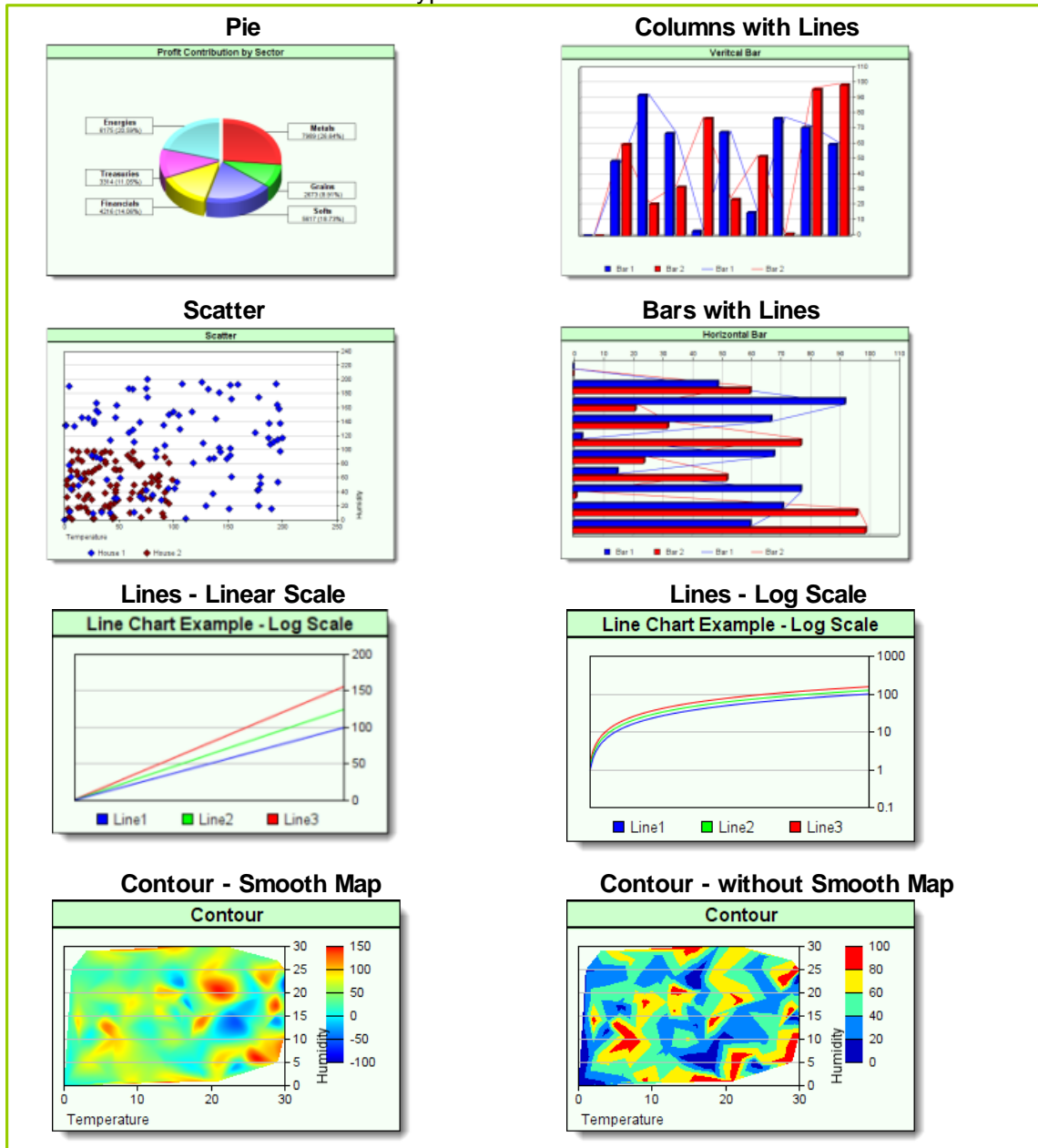
Links:[Broker](#)**See Also:**[AdjustPositionOnOpen](#)

Section 4 – Chart

Custom Charts

At the end of a Simulation Test it is possible to have various custom data graphs display test information on chart not possible previously. These new chart are created with scripts using Trading Blox Basic statements, and they can be directed appear in the **Trading Blox Summary Performance Reports**, or as individual images in a browser window.

Custom Chart images are in addition to the standard BPV Custom Graph images. These new Custom Charts can create new chart types shown here:



New custom charts provide users with an ability to analyze trading ideas with new visual data displays by collecting and analysis data and displaying the results in new ways. These new

chart types are intended to expand the ways that Trading Blox Basic can display data as independent images or as an expanded information in its end of simulation testing report. All the needed new functions and properties needed to create, store and include the custom charts in an expanded report, or in a separate browser page are explained in the topics of this Chart Object subordinate pages.

Chart Creation:

Programming new charts starts with the process of deciding which of the six new chart types will be used.

Five of the new chart use the same basic creation function [NewXY](#), but Pie charts require the [NewPie](#) function designed specifically for Pie charts to start the custom chart process.

Once the chart type, name and image size dimensions have been created, other functions can be added to change from a standard chart display to a more tailor image by apply control over the plotted area size and location, adding overlays that change how a chart appears, and then adding data.

When all the chart creation scripts have been executed, the [Make](#) function is executed so that it creates a file image that can be accessed and loaded into a summary report or a browser page.

When a custom chart is directed to appear in the **Trading Blox Summary Performance Report**, it will be placed in the area below the **Custom Graphs** section. If the custom chart is intended to provide information after a stepped simulation test, it will appear just below the stepped optimization table and chart area of a **Summary Performance Report**.

Charts can also be directed to automatically appear as an images displayed by browsers, or by the user imported the chart image into documents. Regardless of where the new custom chart is directed to appear, the chart images are files that can preserved in any folder named in the chart creation scripting process.

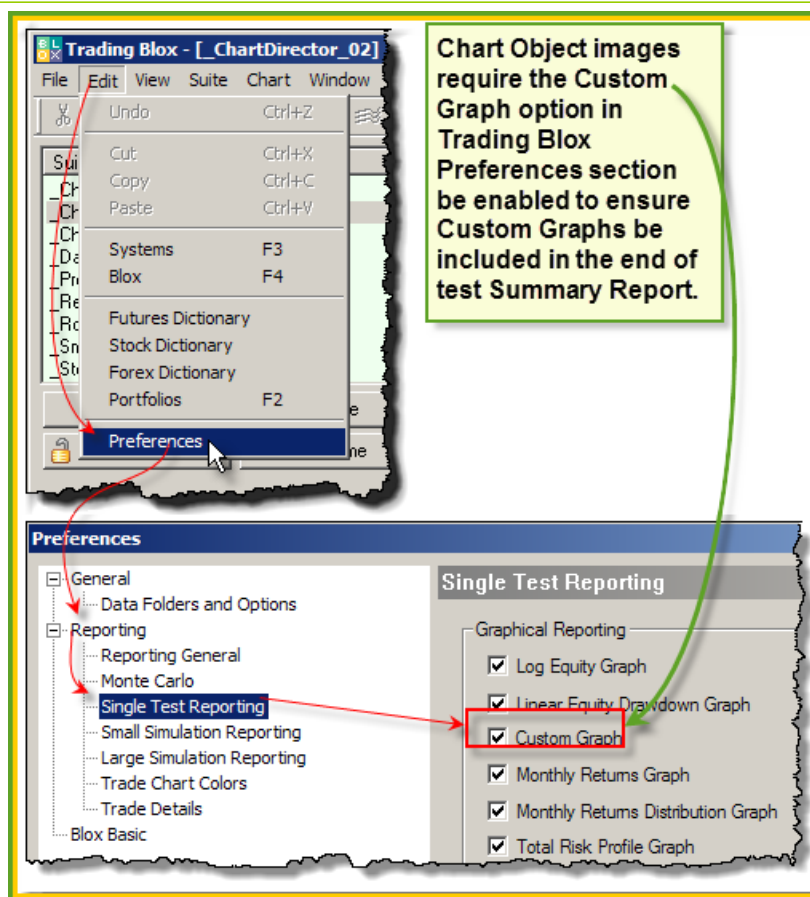
Chart Creation Step:

Custom charts only need a few simple steps to create a chart:

- Collect and analyze the data into BPV numeric series, and BPV String series if labels are needed.
- Determine the size of the chart, and then execute either the [NewXY](#) or [NewPie](#) functions to create chart image space .
- Adjust the plotting area within the chart image so the data plots, scales, labels and legends will all display properly.
- Add all the data series needed for plotting, and then decide if dates, axis labels are needed to improve chart information.
- Make the chart into a finished image by executing the [Make](#) function that directs the file to active report folder.
- Create the simple HTML image display code when images are to be displayed in a browser, or displayed in a performance report.

Custom Chart Requirement:

Before charts can be displayed they must be saved as an image file so they can be accessed and displayed in performance reports or browsers. Creating an image file is made easy with the [Make](#) function that must be present at the end of all chart script sections. Once the file is saved, it can be displayed when the Trading Blox Preference setting is enabled:



Preference settings to enable Custom Graphs and Custom Charts.

Display Custom Charts in a Simulation Report:

Images in the simulation report use a default width of 830 pixels. By using that width, or a smaller value the report's display width will be preserved.

Custom chart images displays in the summary performance report are supported by two new test-object functions. Each function places custom charts in different locations to support where the charts can be found.

Display custom charts in the area at the bottom of where BPV custom graphs are displayed:

Example:

BEFORE TEST SCRIPT

```

' =====
' This task will load the chart SystemEquity.jpg image into
' the simulation report:
' ~~~~~
' This statement creates a single chart displaying task.
test.SetChartTestHtml("<img src=' " _
                      + test.resultsReportPath _
                      + "\SystemEquity" _
                      + AsString(test.currentParameterTest) _
                      + ".gif" _
                      + "' width=830 height=500>")
' =====

```

OR

```

' =====
' This task will load two chart images in the
' simulation report:
' ~~~~~
' Next two lines assign the full path and file name to two BPV
' variables:
chartHtml1 = "<img src=' " _
            + test.resultsReportPath _
            + "\Winning Trades" _
            + AsString( test.currentParameterTest ) _
            + ".gif" _
            + "' width=415 height=400>"

chartHtml2 = "<img src=' " _
            + test.resultsReportPath _
            + "\Losing Trades" _
            + AsString( test.currentParameterTest ) _
            + ".gif" _
            + "' width=415 height=400>"

' This statement creates a task to display two charts
' side by side.
test.SetChartTestHtml( chartHtml1 + chartHtml2 )
' =====

```

OR

```

' =====
' This task will load the same two chart images defined above
' simulation report, but it will place the first image above
' the second image:
' ~~~~~

' This statement creates a task to display two charts
' one above the other.
test.SetChartTestHtml( chartHtml1 + "<br>" + chartHtml2 )
' =====

```

Display charts just below Multi-Parameter Table in the Stepped Parameter Summary Performance table:

Example:

BEFORE TEST SCRIPT

```
' =====
' This task will load the chart SystemEquity.jpg image into
' the simulation report:
' ~~~~~
' This statement creates a single chart displaying task.
test.SetChartSimulationtHtml("<img src=' " _
                             + test.resultsReportPath _
                             + "\SystemEquity" _
                             + AsString
(test.currentParameterTest) _
                             + ".gif" _
                             + "' width=830 height=500>")
' =====

OR
' =====
' This task will load two chart images in the
' simulation report:
' ~~~~~
' Next two lines assign the full path and file name to two BPV
' variables:
chartHtml1 = "<img src=' " _
             + test.resultsReportPath _
             + "\Winning Trades" _
             + AsString( test.currentParameterTest ) _
             + ".gif" _
             + "' width=415 height=400>"

chartHtml2 = "<img src=' " _
             + test.resultsReportPath _
             + "\Losing Trades" _
             + AsString( test.currentParameterTest ) _
             + ".gif" _
             + "' width=415 height=400>"

' This statement creates a task to display two charts
' side by side.
test.SetChartSimulationtHtml( chartHtml1 + chartHtml2 )
' =====

OR
' =====
' This task will load the same two chart images defined above
' simulation report, but it will place the first image above
' the second image:
' ~~~~~

' This statement creates a task to display two charts
```

```
' one above the other.
test.SetChartSimulationHtml( chartHtml1 + "<br>" + chartHtml2 )
' =====
```

Display Custom Charts in default browser:

Usually, the default program is the computer's default browser.

Example:

AFTER TEST SCRIPT

Place this code section below the area where the custom chart script creation has saved the chart using the [chart.Make](#) function

```
' =====
' Show Custom Chart as a HTML image page.
' =====
' Assign custom chart image name to BPV variable
CustomChartName = "SectorPerformancePieChart.png"

' Create Full Path and HTML file name where Custom Chart images
' will be stored and displayed.
SummaryFileLocation = test.resultsReportPath + "\SummaryCharts.htm"

' Open the newly created HTML file name in the new
' test result data folder to get a file number for writing reference
iFileNum = fileManager.OpenWrite( SummaryFileLocation )

' If file is created and opened successfully,...
If iFileNum THEN
' Write the HTML Header & Body tags
fileManager.WriteLine( iFileNum, "<HTML><BODY>" )

' Create the image links for the Scatter Chart Image
fileManager.WriteLine( iFileNum, sTD_Prefix + CustomChartName

' Close HTML Body structure
fileManager.WriteLine( iFileNum, "</BODY></html>" )
ENDIF ' iFileNum

' Close the HTML image display file.
fileManager.Close( iFileNum )

' =====
' Open the new HTML Custom Chart with the Default Browser
OpenFile( SummaryFileLocation )
' =====
```

Links:

[Make](#), [NewPie](#), [NewXY](#), [OpenFile](#), [SetChartSimulationHtml](#), [SetChartTestHtml](#)

4.1 AddBarLayer

AddBarLayer can modify the appearance of bars and enhance the 3D effect of an [XYChart](#) created to show bars or columns.

Syntax:

```
chart.AddBarLayer( [BarMethodEffect], [3D_Depth])
```

Parameters:	Data Information:	
BarMethodEffect	Bar Method Effect:	Use Value:
	Overlay	0
	Stack	1
	Depth	2
	Side (Default)	3
	Percentage	4
	Note: Optional parameter, unless there is a need to change the 3D bar effect. When only a 3D effect change is needed, enter a value of 3 to use the default side-by-side bar display, or use the value of another effects when needed.	
3D_Depth	Optional: No value is required in this parameter location when the 3D effect doesn't need to be changed.	
	Bar Depth Effect:	Use Value:
	Auto Adjust 3D Effect	-1
	Control 3D Depth Size	Pixels depth size

NOTE:

This method is not required with bar charts, but when it is used the first parameter provides five methods that change the way bar are displayed on a chart. Its second parameter provides control over the chart's 3-dimensional appearance. Applying a bar display modification doesn't require the user to change the 3D effect of the chart, but both parameters can be applied at the same time.

This method must be executed ahead of when any of the chart's data is applied so that all the series are handled correctly.

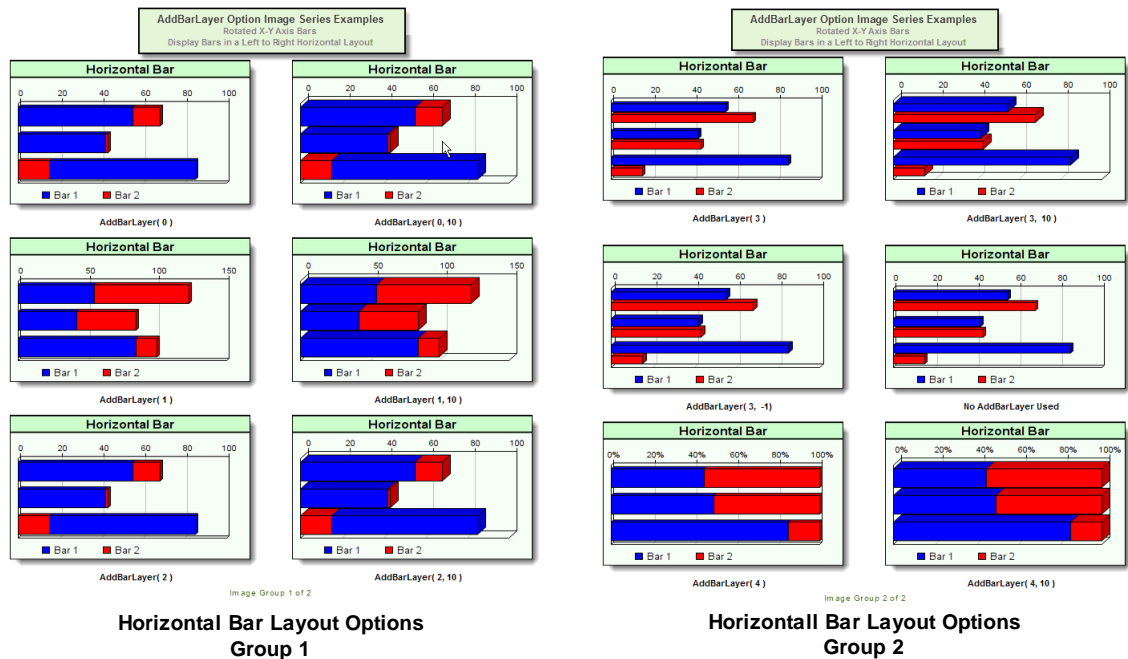
Examples:

```

' ~~~~~
' Rotated XY-Axis - Horizontal Bar Chart Code
' ~~~~~
' Create with rotated X & Y Axis so bars lay horizontal
' "Vertical" option creates horizontal bars
chart.NewXY( "Horizontal Bar", 300, 200, "Vertical" )
' 3D Plot Area Adjustment - See SetPlotArea Notes
chart.SetPlotArea( 10, 30, 60, 30 )
' See Table Notes for BarMethodEffect & 3D_Depth values
chart.AddBarLayer( [BarMethodEffect], [3D_Depth] ) ' Examples ->
' Add 3 element Bar series data "bar1"
chart.AddBarSeries( ASeries( bar1 ), 3 )
' Add 3 element Bar series data "bar2"
chart.AddBarSeries( ASeries( bar2 ), 3 )

' Create & Save an image of the chart with
' this file name. BackSlash Character is Required
' when using ResultsReportPath
chart.Make( test.resultsReportPath + "\" + hbar.png" )

```



```

' ~~~~~
' Standard XY-Axis - Vertical Bar/Column Chart Code
' ~~~~~
' Create with Standard X & Y Axis Columns
chart.NewXY( "Horizontal Bar", 300, 200 )
' 3D Plot Area Adjustment - See SetPlotArea Notes
chart.SetPlotArea( 10, 40, 60, 30 ) '( 10, 50, 60, 30 ) <- % Bars
' See Table Notes for BarMethodEffect & 3D_Depth values
chart.AddBarLayer( [BarMethodEffect], [3D_Depth] ) 'Examples ->

```



```

' Add 3 element Bar series data "bar1"
chart.AddBarSeries( AsSeries( bar1 ), 3 )
' Add 3 element Bar series data "bar2"
chart.AddBarSeries( AsSeries( bar2 ), 3 )

' Create & Save an image of the chart with
' this file name. BackSlash Character is Required
' when using ResultsReportPath
chart.Make( test.resultsReportPath + "\" + vbar.png" )

```

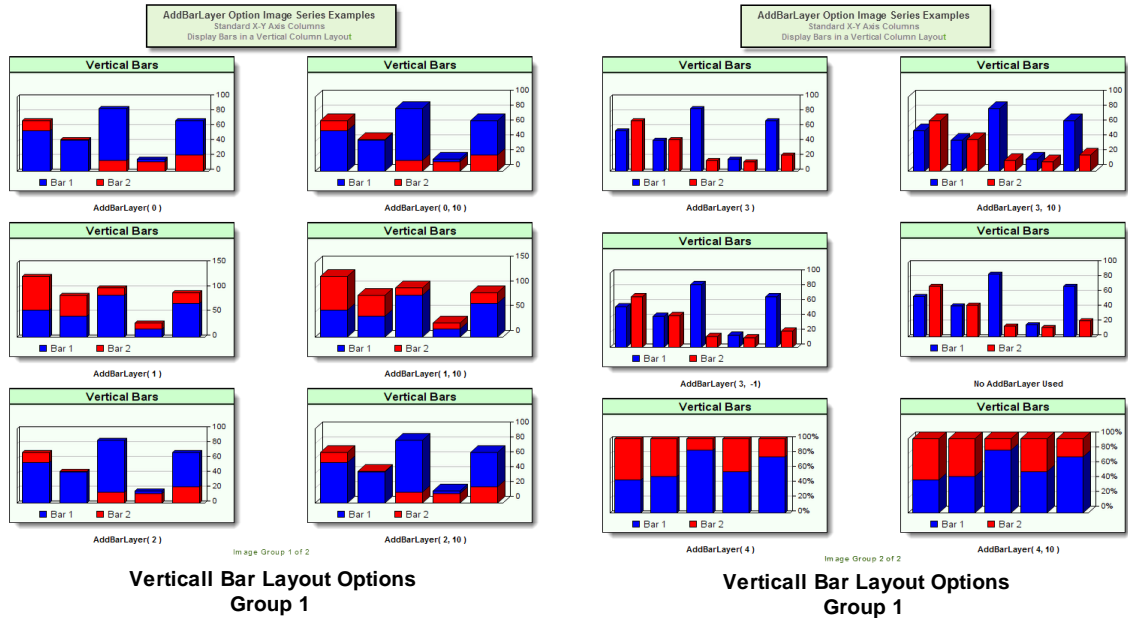
**Note:**

Chart example images have a drop shadow effect so they would appear above the background. [NewXY](#) can automatically add a similar drop-shadow effect when the "Shadow" option is added as an option.

Links:

[AddBarSeries](#), [AsSeries](#), [Make](#), [NewXY](#), [resultsReportPath](#), [SetPlotArea](#)

See Also:

4.2 AddBarSeries

AddBarSeries adds a different series of data on a chart display.

Syntax:

```
chart.AddBarSeries( AsSeries( BarSeries ), Elements )
```

Parameter:	Data Information:
BarSeries	BPV Numeric series intended to represent a group of bars. Note: Use with all BPV Numeric or String series that are passed to any Chart parameter. AsSeries(...) function conditions the series so the chart function will be able to use the information contained within each of the series elements.
Elements	Count of the numeric elements in the series. Note: Manually Sized Series: GetSeriesSize function provides the element count. Auto-Index series: BPV: test.currentDay property reports last series element index. IPV: instrument.bar property reports last series element index.

NOTE:

To add an addition bar group to a chart, call **AddBarSeries** again with a different data series.

Example:

```
' ~~~~~
' CREATE a Column / Vertical Bar Chart
' Create graphing space for a horizontal chart 300-Pixel wide,
' & 200-Pixels tall with chart title: "Vertical Columns"
chart.NewXY( "Line Chart", 300, 200 )

' Size Plotting Area to these values
chart.SetPlotArea( 10, 35, 60, 30 )

' Use Side-by-Side Bar/Column display
chart.AddBarLayer( 3 )

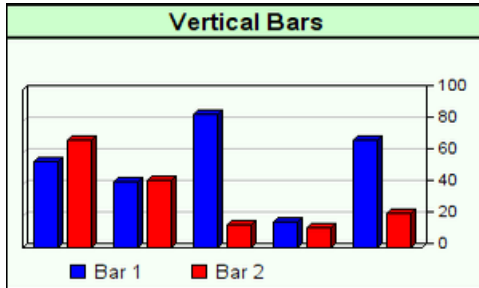
' Add 5 element values to represent "bar1"
chart.AddBarSeries( AsSeries( bar1 ), 5 )

' Add 5 element values to represent "bar2"
chart.AddBarSeries( AsSeries( bar2 ), 5 )

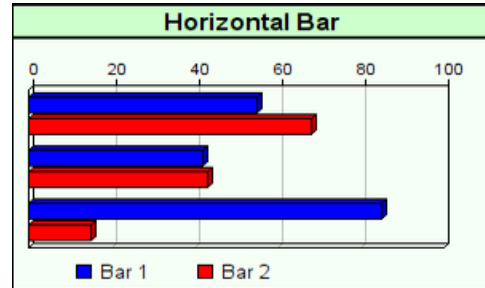
' Create & Save an image of the chart with
' this file name. BackSlash Character is Required
```

```
' when using ResultsReportPath  
chart.Make( Test.ResultsReportPath + "\" + "vbar.png" )
```

NewXY Option "Vertical" Not Enabled.



NewXY Option "Vertical" Enabled.



Links:

[AddBarLayer](#), [AsSeries](#), [Make](#), [NewXY](#), [ResultsReportPath](#), [SetPlotArea](#)

See Also:

4.3 AddContourLayer

Applies a 3-Dimensional contour layer color map onto a newly created **NewXY** chart.

Syntax:

```
chart.AddContourLayer( AsSeries(XAxisSeries), _
                      AsSeries(YAxisSeries), _
                      AsSeries(ZAxisSeries), _
                      SeriesCount, _
                      [Smooth] )
```

Parameter:	Description:
XAxisSeries	X-Axis Series data.
YAxisSeries	Y-Axis Series data.
ZAxisSeries	Z-Axis Series data.
	Note: Use with all BPV Numeric or String series that are passed to any Chart parameter. AsSeries (...) function conditions the series so the chart function will be able to use the information contained within each of the series elements.
SeriesCount	Integer value specifies the number of data element values in of all three of the data series specified. Note: Manually Sized Series: GetSeriesSize function provides the element count. Auto-Index series: BPV: test.currentDay property reports last series element index. IPV: instrument.bar property reports last series element index.
Smooth	Optional: When left out, chart will draw contours with clear line definitions between the contour colors levels creating levels that change with clear level lines. When the optional "Smooth" is added the color changes showing the different contour levels will have a blended color change transition indicating the landscape level change have a smooth contoured slope.

NOTE:

This function must be executed after the **NewXY** statement has sized the chart.

AddContourLayer requires data for the X,Y and Z data series, which are the first three parameters. Fourth parameter requires the count of the number elements in the Z-Axis series.

An optional "Smooth" parameter can be applied so that the color transitions between each of the different levels displayed are shown as blended color gradient transition between area colors.

Contour maps display the X-Axis scale below the plotting areas lower chart boundary. Y-Axis scale is displayed just outside the plot areas right side boundary. Z-Axis scale steps are

displayed in the area between the outside boundary Y-Axis scale. White space on the right side of a contour chart needs to provide enough pixel space to enable the Z-Scale color legend to display so that it doesn't interfere with the Y-Scale display.

Creating the extra space around plotted areas is created by the use of the [SetPlotArea](#) function.

Example:

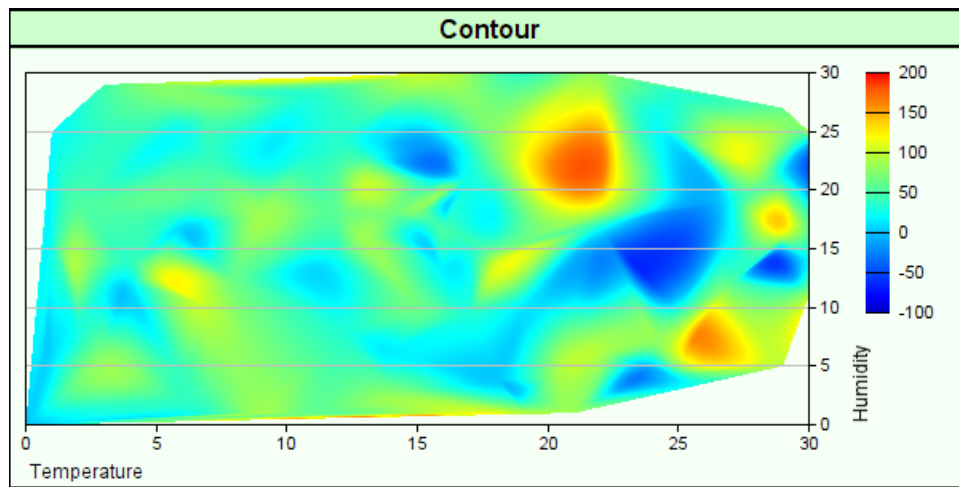
```
' ~~~~~
' CONTOUR COLOR MAP CHART
' ~~~~~
' Create graphing space that is 600-Pixels wide, & 300-Pixel high.
' Place the name "Contour Map" in the chart's window Title Bar space.
chart.NewXY( "Contour Map", 600, 300 )

' Size the Bar Plotting area within the boundaries of the graphing
' image area
' (x-Left, x-Right, y-Top, y-Bottom )
chart.SetPlotArea( 10, 100, 40, 40 )

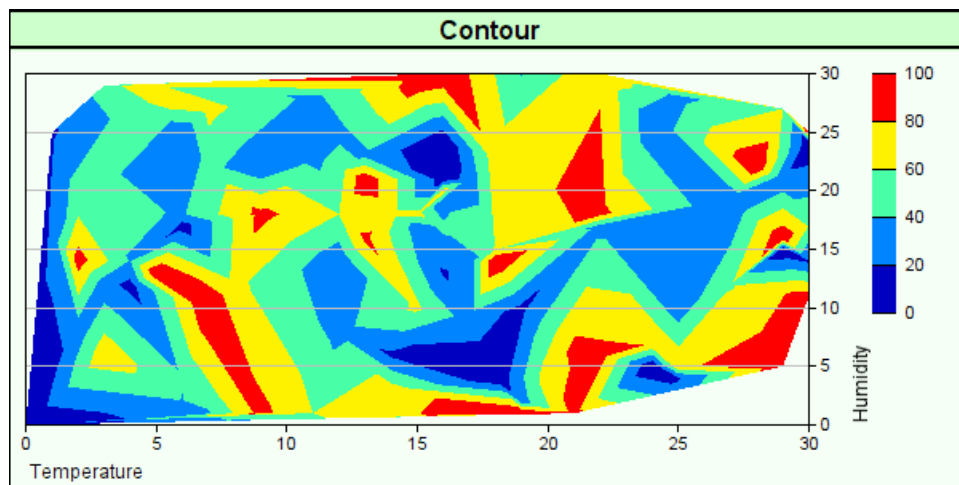
' Create a contour map with BPV numeric series for x, y, & z axis
chart.AddContourLayer( AsSeries( randomx ), _
                      AsSeries( randomy ), _
                      AsSeries( house2x ), iContourLevels, sSurface )

' Add Titles for X & Y Axis Scales
' Function places: "Temperature" near the x-Axis bottom left-side
' and it places "Humidity" at the bottom right as vertical text
chart.SetAxisTitles( "Temperature", "Humidity" )

' Create & Save this graph as a chart image file.
' Note: a BackSlash "\" Character is Required
' when using ResultsReportPath <-- Click Link below for Details
chart.Make( test.resultsReportPath + "\" + "Contour.png" )
' ~~~~~
```



Contour XY & Z Axis with Smooth Enabled



Contour Smooth option omitted AddContourLayer

Links:

[AsSeries](#), [Make](#), [NewXY](#), [resultsReportPath](#), [SetAxisTitles](#), [SetPlotArea](#)

See Also:

4.4 AddLineLayer

Function allows the chart's Linear scale to be converted to a Log scale.

Syntax:

```
chart.AddLineLayer( [LineMethod], [Options] )
```

Parameter:	Description:
LineMethod	Not used. However, when the Log option is enabled, enter a Zero or 1 in this field.
Options	Converts linear scale to a Log scale.

Example:

```
' ~~~~~
' Log Scale Line Chart Example
' ~~~~~
' Establish Scatter Chart image size
iChartWidth = 500      ' X-Axis Width
iChartHeight = 300    ' Y-Axis Height

' Create a image
chart.NewXY( "Log Scale Line Chart Example", iChartWidth, iChartHeight
' Size the Scatter Dot Plotting area
chart.SetPlotArea( 30, 30, 40, 50 )

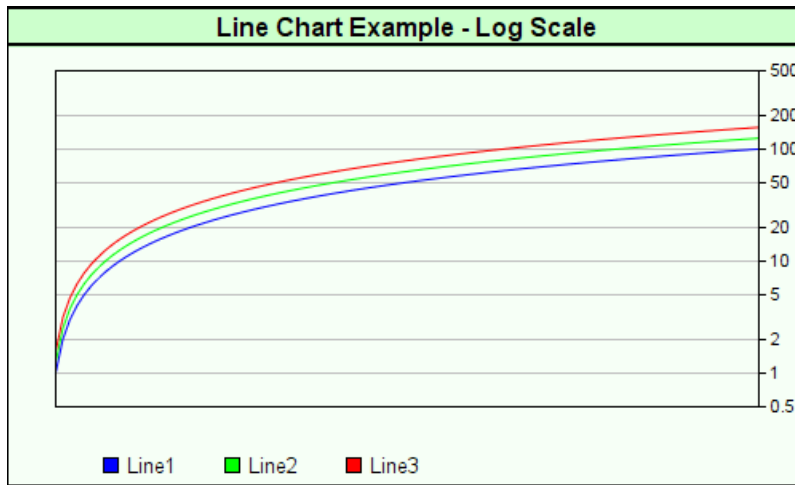
' Use Log Scale
chart.AddLineLayer( 0, "Log" )    ' <-- Converts Linear Scale to Log Scale

' Generate Blue Color Value
ColorValue1 = ColorRGB( 255, 0, 0 )
' Generate Green Color Value
ColorValue2 = ColorRGB( 0, 255, 0 )
' Generate Red Color Value
ColorValue3 = ColorRGB( 0, 0, 255 )

' Add Line data series 1
chart.AddLineSeries( AsSeries( Line1 ), 100, "Line1", ColorValue1 )
' Add Line data series 2
chart.AddLineSeries( AsSeries( Line2 ), 100, "Line2", ColorValue2 )
' Add Line data series 3
chart.AddLineSeries( AsSeries( Line3 ), 100, "Line3", ColorValue3 )

' Create & Save this new chart as a file.
' Always add a backSlash Character after "resultsReportPath"
chart.Make( test.resultsReportPath + "\" + "LogChartExample.png" )
```

Code Script Output:

**Links:**

[ASeries](#), [AddLineSeries](#), [ColorRGB](#), [Make](#), [NewXY](#), [resultsReportPath](#), [SetPlotArea](#)

See Also:

4.5 AddLineSeries

Adds a chart line to a [NewXY](#) chart. Function has four parameters, but only the first two parameters are required.

First parameter is a BPV data series, Second parameter is the element count of the first parameter's data series.

Two optional parameters are, Title and Color. Title will be the name displayed with the series, and Color will determine the color if its value isn't assigned the default value of "-1". Leaving the Color parameter blank will lets the chart automatically designate an unused color.

Syntax:

```
chart.AddLineSeries( AsSeries( LineSeries ), SeriesElements, [Title], [Color]
```

Parameter:	Description:
LineSeries	<p>Name of BPV Series containing data to be plotted.</p> <p>Note: Use with all BPV Numeric or String series that are passed to any Chart parameter. AsSeries(. . .) function conditions the series so the chart function will be able to use the information contained within each of the series elements.</p>
SeriesElements	<p>Number of data elements in the series.</p> <p>Note: Manually Sized Series: GetSeriesSize function provides the element count. Auto-Index series: BPV: test.currentDay property reports last series element index. IPV: instrument.bar property reports last series element index.</p>
Title	Parameter is a text field. Name entered will be used as the name for the line series label. If the name option is not used, the name of the data series name is used.
Color	A color value of: -1 - will uses the chart's automatic color assignment. See ColorRGB and the Colors for information on how to use other colors.

Example:

```

' ~~~~~
' LINE CHART EXAMPLE
' ~~~~~
' Establish Scatter Chart image size
iChartWidth = 500      ' X-Axis Width
iChartHeight = 300    ' Y-Axis Height

' Create a image
chart.NewXY( "Line Chart Example", iChartWidth, iChartHeight )

' Size the Scatter Dot Plotting area
chart.SetPlotArea( 30, 30, 40, 50 )

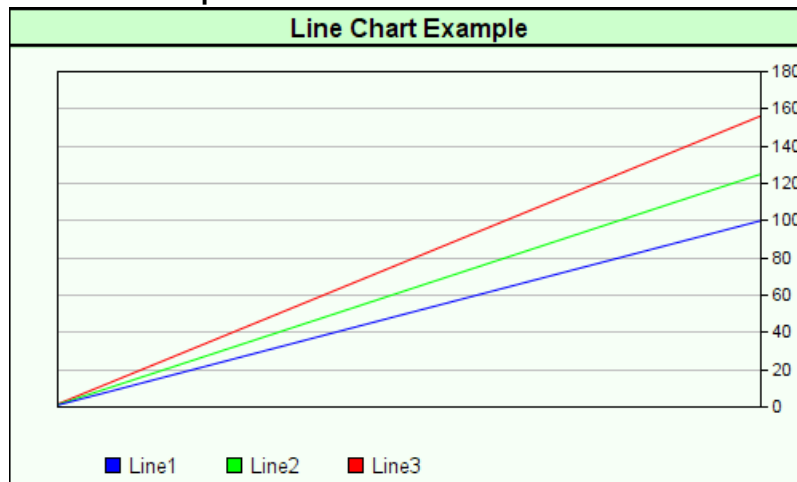
' Generate Blue Color Number
ColorValue1 = ColorRGB( 255, 0, 0 )
' Generate Green Color Number
ColorValue2 = ColorRGB( 0, 255, 0 )
' Generate Red Color Number
ColorValue3 = ColorRGB( 0, 0, 255 )

' Add Line data series 1
chart.AddLineSeries( ASeries( Line1 ), 100, "Line1", ColorValue1 )
' Add Line data series 2
chart.AddLineSeries( ASeries( Line2 ), 100, "Line2", ColorValue2 )
' Add Line data series 3
chart.AddLineSeries( ASeries( Line3 ), 100, "Line3", ColorValue3 )

' Create & Save this new chart as a file.
' Always add a backSlash Character after "resultsReportPath"
chart.Make( test.resultsReportPath + "\" + "LineChartExample.png" )
' ~~~~~

```

3-Line Chart Example:

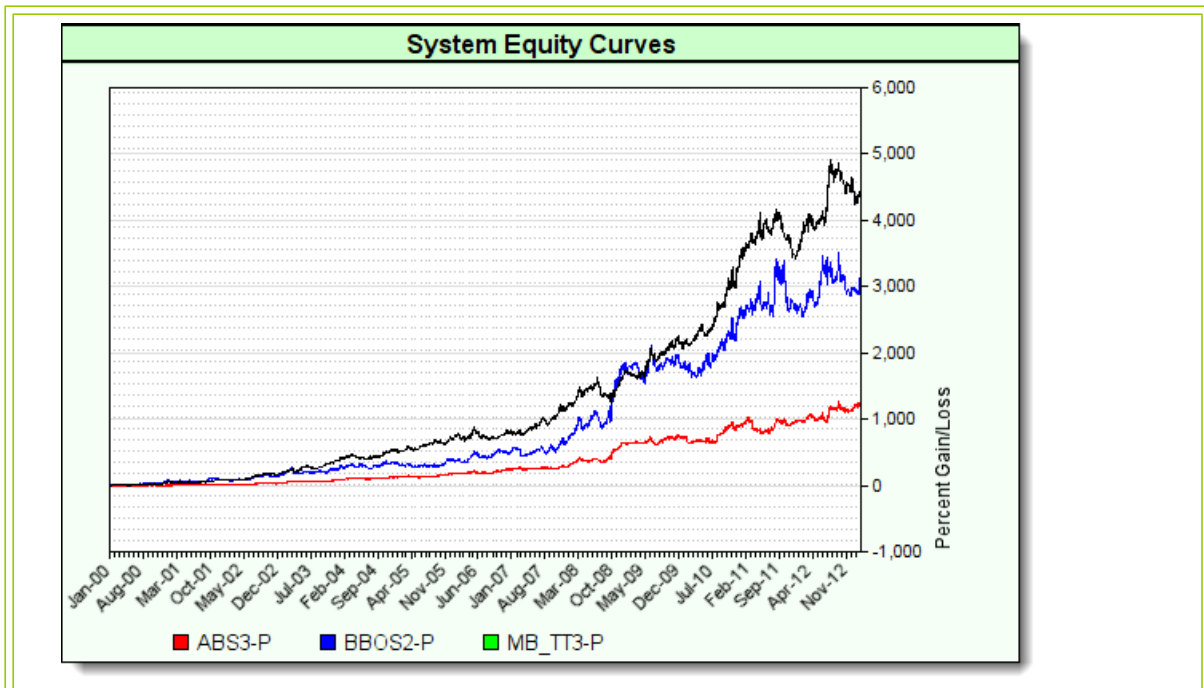


Multi-Line Chart Example:

```

' ~~~~~
' Create a chart area that is 80-Pixels wide, and 500-Pixels high
chart.NewXY( "System Equity Curves", iChartWidth, iChartHeight )
' Number of data points to place on the chart
elementCount = test.currentDay
' Place a scale label identifying the vertical scale information
chart.SetAxisTitles( "", "Percent Gain/Loss" )
' Set X-Axis Dates to use beginning of month
chart.SetxAxisDates(AsSeries(DateSeries), elementCount, 4)
' Plot a Black line at the chart's Y-Axis Zero location
chart.AddLineSeries( AsSeries( systemEquity ), elementCount, "", 0 )
' ~~~~~
' Examine each system in the Simulation Suite
For systemIndex = 1 TO test.systemCount
' Access each system in the suite
test.SetAlternateSystem( systemIndex )
' Capture the daily exchange net equity rate change of '
' each system in the Suite
For i = 1 TO elementCount
' Calculate the total equity net percentage change between '
' Test dates and store that information in a BPV system equity se
systemEquity[i] = ( alternateSystem.totalEquity[ elementCount - i
/ alternateSystem.totalEquity[ elementCount - 1 ] - 1 )
Next ' i
' ~~~~~
' Assign each system net rate change to a specific color
plotColor = ColorItem[ systemIndex ]
'OR
' Consider a random number color assignment
' BLUE GREEN RED
'plotColor = ColorRGB( Random(255), Random(255), Random(255) )
' Place this system's test-date total equity percentage net change
' value in the chart space using the new color
chart.AddLineSeries( AsSeries(systemEquity), elementCount, _
alternateSystem.name, plotColor )
Next ' systemIndex
' ~~~~~
' When all the system's equity changes are display as new chart
' lines, save the image information as a file.
chart.Make( sFilePath2 )
' ~~~~~

```

**Links:**

[AsSeries](#), [ColorRGB](#), [Make](#), [NewXY](#), [resultsReportPath](#), [SetPlotArea](#)

See Also:

4.6 AddScatter

Scatter charts show unconnected shapes that reflect the two-dimensional values of each data point intersection on supplied data series.

Syntax:

```
chart.AddScatter(AsSeries(xSeries),AsSeries(ySeries),ElementCount,[Symbol
```

Parameter:	Description:																
xSeries	An array of numbers representing the x values of the data points. If no explicit x coordinates are used in the chart (eg. using an enumerated x-axis), an empty array may be used for this argument.																
ySeries	An array of numbers representing the y values of the data points. Note: Use with all BPV Numeric or String series that are passed to any Chart parameter. AsSeries(...) function conditions the series so the chart function will be able to use the information contained within each of the series elements.																
ElementCount	Number of data elements in the series. Note: Manually Sized Series: GetSeriesSize function provides the element count. Auto-Index series: BPV: test.currentDay property reports last series element index. IPV: instrument.bar property reports last series element index.																
Symbol	<table border="1"> <thead> <tr> <th>Enter #:</th> <th>Shape Description:</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Square shape</td> </tr> <tr> <td>2</td> <td>Diamond shape</td> </tr> <tr> <td>3</td> <td>Triangle pointing up</td> </tr> <tr> <td>4</td> <td>Triangle pointing right</td> </tr> <tr> <td>5</td> <td>Triangle pointing left</td> </tr> <tr> <td>6</td> <td>Triangle pointing down</td> </tr> <tr> <td>7</td> <td>Circle</td> </tr> </tbody> </table>	Enter #:	Shape Description:	1	Square shape	2	Diamond shape	3	Triangle pointing up	4	Triangle pointing right	5	Triangle pointing left	6	Triangle pointing down	7	Circle
Enter #:	Shape Description:																
1	Square shape																
2	Diamond shape																
3	Triangle pointing up																
4	Triangle pointing right																
5	Triangle pointing left																
6	Triangle pointing down																
7	Circle																
Size	Optional parameter will create symbols the at size entered. Default symbol size is 12-Pixels.																

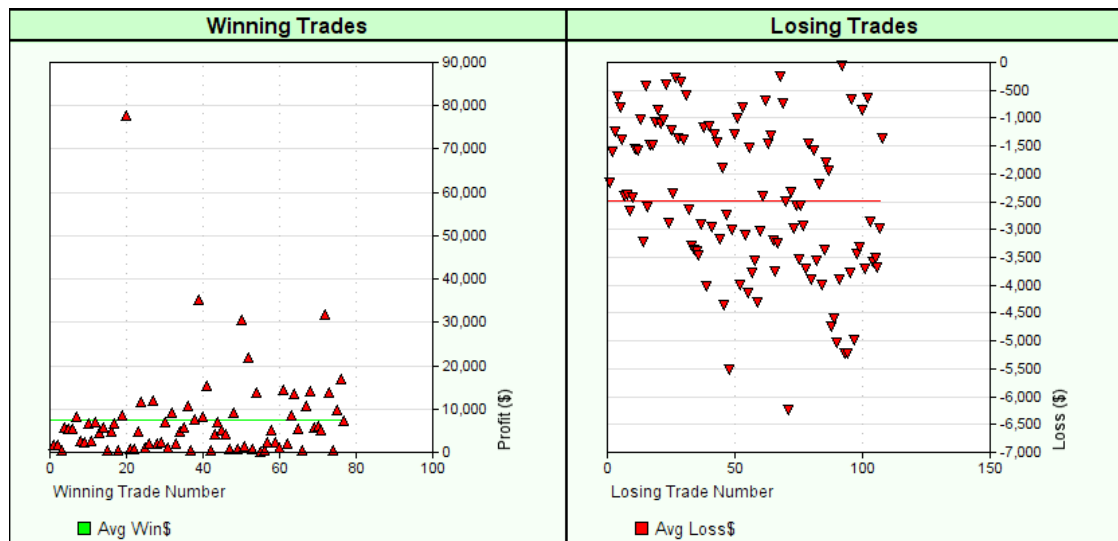
Example:

```
' ~~~~~  
' Win Trade Code Example - More code available in blox "Trade Charts"
```

```

' -----
' Create a scatter chart of Winning Trades
chart.NewXY( "Winning Trades", 415, 400 )
' Display each trade's profit as a chart dot
chart.AddScatter( AsSeries( labelSeries), AsSeries( dataSeries), count,
' Display the average of the sum of all winning trades as a line
chart.AddLineSeries( AsSeries( dataAverage ), count, "Avg Win$", ColorR
' Display the Chart's labels for X & Y Axis scales
chart.SetAxisTitles( "Winning Trade Number", "Profit ($)" )
' Create the chart as an image file in the current test folder.
chart.Make( test.resultsReportPath _
+ "\Winning Trades" _
+ AsString( test.currentParameterTest ) _
+ ".gif" )
' ~~~~~

```



Trade Charts Blox - Win Loss Report Image Example

Click Image to enlarge, Click again to reduce

Links:

[AddLineSeries](#), [AsSeries](#), [Make](#), [NewXY](#), [resultsReportPath](#),
[SetxAxisLabels](#), [SetAxisTitles](#), [SetPlotArea](#)

See Also:

4.7 Make

Saves scripted charts as image files in the names of the location and file name specified.

Syntax:

```
chart.Make( FileSavingDetails )
```

Parameter:	Description:
FileSavingDetails	<p>This parameter must contain the path, folder, file name and the image type name being created.</p> <p>String variable or Text contained with quotation marks required: <Drive>:\<Full path and folder name>\<filename>.<image-format></p> <p>Supported Image Formats: PNG, JPG, GIF, BMP</p>

Notes:

This function only creates the file so that other methods can access the graph and then display it in a report, and or browser.

When specifying destination paths where the file is to be saved, it is recommended the path be created using the test-object property: `resultsReportPath`. This property identifies the the current test folder path and folder name where the other test report images and data are being saved, and it provides a convenient method for referencing the destination of where to place a file, and then later in the script where to access the file when it is time to display it in a report or browser.

Example:

```
' Create & Save this new chart as a file.
' Always add a backSlash Character after "resultsReportPath"
chart.Make( test.resultsReportPath + "\" + "Scatter.png" )
```

Results:

A drive, path, including folder-name, will be appended to your file name, and will appear similar to this next line as long you have added a backslash character '\' between the folder name and the file name:

```
"C:\Trading Blox\Results\Test 2013-01-08_08_47_55\Scatter.png"
```

File path information folder name reflects the 'Suite Name' and the date and time in YYYY-MM-DD_HH_MM_S format of the test executed and the time execution began.

Click on any of the these links to review chart creation code examples to see how all the chart methods use this `Make` function:

- [AddBarSeries](#)
- [AddContourLayer](#)
- [AddLineLayer](#)
- [AddLineSeries](#)
- [AddScatter](#)
- [NewPie](#)

Links:

[Chart](#), [General Properties](#), [resultsReportPath](#)

See Also:

4.8 NewPie

Creates a sectioned Pie chart that can show legends, and pie section values connected to each segment of the pie chart.

Syntax:

```
Chart.NewPie( ChartTitle, xAxisWidth, yAxisHeight, _
              AsSeries(pieChartValues), AsSeries(pieLabels), PieSections,
```

Parameter:	Data Information:
ChartTitle	String variable. Name to display in pie chart's title bar area.
xAxisWidth	Horizontal pixel external width of chart.
yAxisHeight	Vertical pixel external height of chart.
pieChartValues	BPV numeric series of pie section value.
pieLabels	BPV String series of pie section names. Note: Use with all BPV Numeric or String series that are passed to any Chart parameter. AsSeries(...) function conditions the series so the chart function will be able to use the information contained within each of the series elements.
PieSections	pieChartValue sectopm and pieLabels series.
Option	Option create an upper left corner light source shadow behind the lower right area of the pie chart image so that is it appears to stand above the background area.

Notes:

Do not use the [SetPlotArea](#) function with PIE charts because they don't support adjustable overlays.

Example:

```

' ~~~~~
' PIE CHART - Script Example from the Pie Charts Blox
' ~~~~~
' Establish Chart image size
iChartWidth = 500      ' X-Axis Width
iChartHeight = 300    ' Y-Axis Height

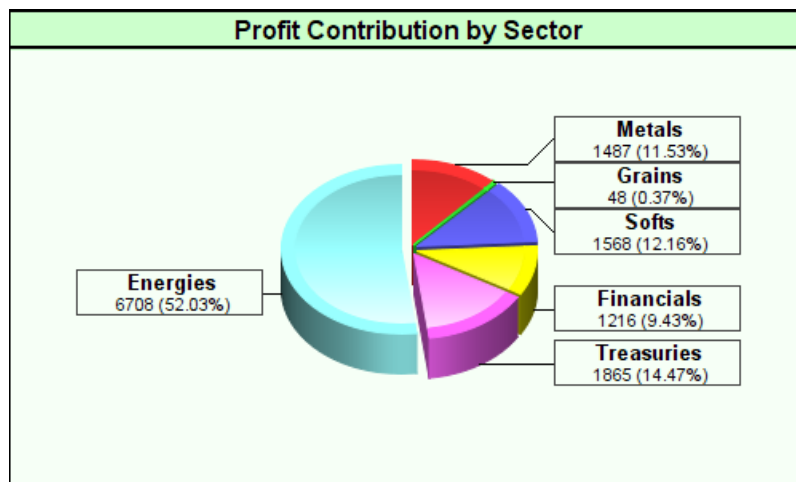
' Create Random Pie Segment Values
pieChartValues[1] = Random(10000)
pieChartValues[2] = Random(10000)
pieChartValues[3] = Random(10000)
pieChartValues[4] = Random(10000)
pieChartValues[5] = Random(10000)
pieChartValues[6] = Random(10000)

' Pie Label Values
pieLabels[1] = "Metals"
pieLabels[2] = "Grains"
pieLabels[3] = "Softs"
pieLabels[4] = "Financials"
pieLabels[5] = "Treasuries"
pieLabels[6] = "Energies"

' Create a Pie graph, use 6 numeric pieChartValues values
' add 6 segment pieLabel names, do not use "shadow" option.
chart.NewPie( "Profit Contribution by Sector", iChartWidth, iChartHeight
             AsSeries(pieChartValues), AsSeries(pieLabels), 6 )

' Create & Save this new chart as a file.
' Always add a backSlash Character after "resultsReportPath"
chart.Make( test.resultsReportPath + "\" + "PieChart.png" )

```



Shadow styling around image was added later by graphics editing program

Links:

[AsSeries](#), [Make](#), [NewPie](#), [Random](#), [resultsReportPath](#)

See Also:

4.9 NewXY

NewXY is the starting method used to create Bar, Scatter, Line, linear and, or Logarithmic charts.

Syntax:

```
Chart.NewXY( ChartTextName , ChartPixelWidth , ChartPixelHeight , [ChartOptions]
```

Parameter:	Descriptions:										
ChartTextName	Text assigned to this parameter will be placed in the title bar area above the chart image.										
ChartPixelWidth	Chart width is determined by the number of pixel entered into this parameter.										
ChartPixelHeight	Chart height is determined by the number of pixel entered into this parameter.										
ChartOptions	There are three optional words that can change a chart:										
	<table border="1"> <thead> <tr> <th>Methods:</th> <th>Descriptions:</th> </tr> </thead> <tbody> <tr> <td>Log</td> <td>Using the Log option in this parameter will convert the Y-Axis scale of the chart display from a Linear-scale to a Log-scale presentation</td> </tr> <tr> <td>Shadow</td> <td>This option adds a drop shadow effect to the chart's appearance. When used the shadow effect reduces the plot area of the image by the size of the added drop shadow effect. Color used for the shadow will be different from the normal white color of the background to make its appearance noticeable.</td> </tr> <tr> <td>Vertical</td> <td>Bar, Contour, Dot, and Line charts are oriented according the placement of the X & Y axis locations. In most charts the orientation of the X-Axis is placed at the bottom of the chart, and the Y-Axis is place along the side, or vertical axis. This option allows the X & Y Axis locations to be rotated so that the X-Axis is positioned along the side, or vertical axis, and it places the Y-Axis along the bottom axis. When "Vertical" is used in this optional parameter location, the chart's data axis locations will show its visual information rotated 90-degrees to the right. Vertical bars which show their value by their height orientation along an image's side or vertical axis will change to displaying horizontal bars that show their value in a left to right orientation along the rotated bottom axis. Contour, dot and line information will also be rotated when those type of displays are added to the chart image space.</td> </tr> <tr> <td>Combined Methods</td> <td>All three methods can be combined as an additional instruction to the NewXY chart method.</td> </tr> </tbody> </table>	Methods:	Descriptions:	Log	Using the Log option in this parameter will convert the Y-Axis scale of the chart display from a Linear-scale to a Log-scale presentation	Shadow	This option adds a drop shadow effect to the chart's appearance. When used the shadow effect reduces the plot area of the image by the size of the added drop shadow effect. Color used for the shadow will be different from the normal white color of the background to make its appearance noticeable.	Vertical	Bar, Contour, Dot, and Line charts are oriented according the placement of the X & Y axis locations. In most charts the orientation of the X-Axis is placed at the bottom of the chart, and the Y-Axis is place along the side, or vertical axis. This option allows the X & Y Axis locations to be rotated so that the X-Axis is positioned along the side, or vertical axis, and it places the Y-Axis along the bottom axis. When "Vertical" is used in this optional parameter location, the chart's data axis locations will show its visual information rotated 90-degrees to the right. Vertical bars which show their value by their height orientation along an image's side or vertical axis will change to displaying horizontal bars that show their value in a left to right orientation along the rotated bottom axis. Contour, dot and line information will also be rotated when those type of displays are added to the chart image space.	Combined Methods	All three methods can be combined as an additional instruction to the NewXY chart method.
	Methods:	Descriptions:									
	Log	Using the Log option in this parameter will convert the Y-Axis scale of the chart display from a Linear-scale to a Log-scale presentation									
	Shadow	This option adds a drop shadow effect to the chart's appearance. When used the shadow effect reduces the plot area of the image by the size of the added drop shadow effect. Color used for the shadow will be different from the normal white color of the background to make its appearance noticeable.									
Vertical	Bar, Contour, Dot, and Line charts are oriented according the placement of the X & Y axis locations. In most charts the orientation of the X-Axis is placed at the bottom of the chart, and the Y-Axis is place along the side, or vertical axis. This option allows the X & Y Axis locations to be rotated so that the X-Axis is positioned along the side, or vertical axis, and it places the Y-Axis along the bottom axis. When "Vertical" is used in this optional parameter location, the chart's data axis locations will show its visual information rotated 90-degrees to the right. Vertical bars which show their value by their height orientation along an image's side or vertical axis will change to displaying horizontal bars that show their value in a left to right orientation along the rotated bottom axis. Contour, dot and line information will also be rotated when those type of displays are added to the chart image space.										
Combined Methods	All three methods can be combined as an additional instruction to the NewXY chart method.										

Example:

"Vertical Shadow" will create a rotated axis chart with a drop shadow effect

"Log Shadow" will create a rotated axis with a drop shadow effect.

"Log Vertical Shadow" will create a rotated axis Log chart with a drop shadow effect.

Notes:

After this method has been executed, the type of features available on a chart are determined by adding other methods to the chart to generate the kind of chart needed.

When **NewXY** is executed:

- First parameter is a text-value that places the characters into the chart's title label area.
- Second parameter is a numeric value that determines the chart's outside boundary width in pixels.
- Third parameter is value that determines the chart's outside vertical height in pixels.
- Fourth parameter is an optional modification string value that can rotate the orientation of the X & Y axis, the scaling of the data from a Linear to a Log scaled display, and it can add a drop shadow effect to the resulting image.

Examples:

NewXY function is required in the code scripts to start the creation of all the custom charts except the Pie Charts:

- [AddBarSeries](#)
- [AddContourLayer](#)
- [AddLineLayer](#)
- [AddLineSeries](#)
- [AddScatter](#)

Links:

[Chart](#), [General Properties](#)

See Also:

4.10 SetAxisTitle

`SetAxisTitles` function adds two text axis titles to X & Y scale locations.

Syntax:

```
chart.SetAxisTitles( xAxisLabel, yAxisLabel )
```

Parameter:

Description:

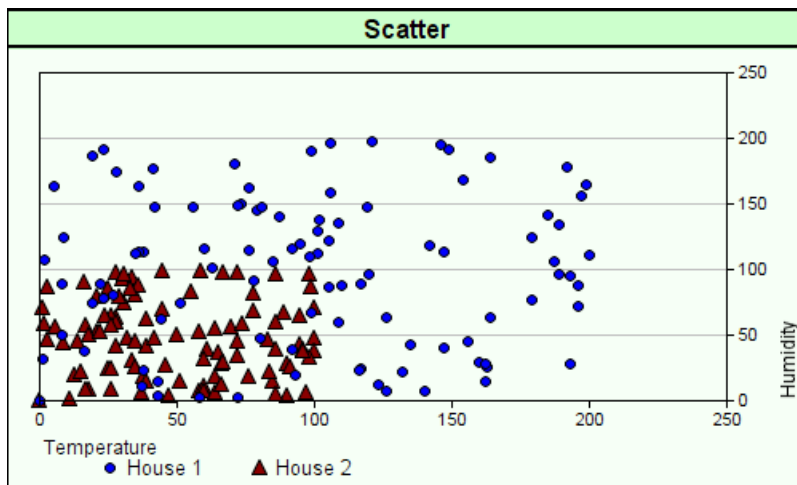
xAxisLabel	String - Name of x-Axis title.
yAxisLabel	String - Name of y-Axis title.

Note:

Plotting area in this image used the `SetPlotArea` function to control how much space around the image would be provided for scale values and the addition of the axis titles. Axis titles are displayed without a problem, but if the x-axis scale values are converted to dates, the additional space needed by the data labels might cause some or all of the the x-axis title name to be obscured. When that happens, increase the plot area pixel size of the last value parameter on the right. Rotating scales using the "Vertical" option will also require plot area size location changes.

Example:

```
' ~~~~~
' SCATTER CHART - x-Axis At Chart Bottom - Axis Not Rotated
' ~~~~~
' Establish Scatter Chart image size
iChartWidth = 500      ' X-Axis Width
iChartHeight = 300    ' Y-Axis Height
' Create a image
chart.NewXY( "Scatter", iChartWidth, iChartHeight )
' Size the Scatter Dot Plotting area
chart.SetPlotArea( 20, 50, 40, 55 )
' Place the X-Axis scale label "Temperature" in the left corner.
' Place the Y-Axis label "Humidity" in the lower right corner.
chart.SetAxisTitles( "Temperature", "Humidity" )
' Place the series of 100 dots positioned at element coordinates
' x & y using the BPV "Randomx" element values.
chart.AddScatter( AsSeries( randomx ), AsSeries( randomy ), iNumber, 7,
' Place the series of 100 dots positioned at element coordinates
' x & y using the BPV "House2x" element values.
chart.AddScatter( AsSeries( house2x ), AsSeries( house2y ), iNumber, 3,
' Create & Save this new chart as a file.
' Always add a backSlash Character after "resultsReportPath"
chart.Make( test.resultsReportPath + "\" + "Scatter.png" )
```

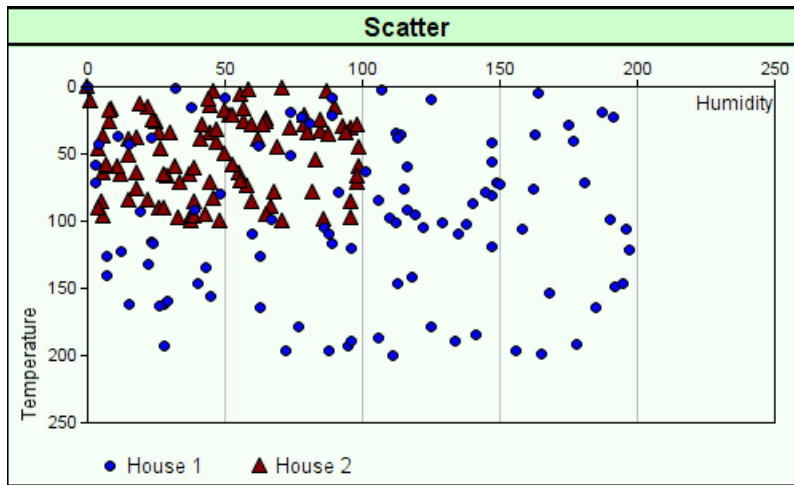


Standard x-Axis position places x-Axis below x-Scale
and y-Axis scale is placed along y-Scale on right side of plotted area.

```

' ~~~~~
' SCATTER CHART - x-Axis At Chart Left Size - Axis Rotated
' ~~~~~
' Establish Scatter Chart image size
iChartWidth = 500      ' X-Axis Width
iChartHeight = 300    ' Y-Axis Height
' Create a image
chart.NewXY( "Scatter", iChartWidth, iChartHeight, "Vertical" )
' Size the Scatter Dot Plotting area
chart.SetPlotArea( 50, 20, 50, 40 )
' Place the X-Axis scale label "Temperature" in the left corner.
' Place the Y-Axis label "Humidity" in the lower right corner.
chart.SetAxisTitles( "Temperature", "Humidity" )
' Place the series of 100 dots positioned at element coordinates
' x & y using the BPV "Randomx" element values.
chart.AddScatter( AsSeries( randomx ), AsSeries( randomy ), iNumber, 7,
' Place the series of 100 dots positioned at element coordinates
' x & y using the BPV "House2x" element values.
chart.AddScatter( AsSeries( house2x ), AsSeries( house2y ), iNumber, 3,
' Create & Save this new chart as a file.
' Always add a backSlash Character after "resultsReportPath"
chart.Make( test.resultsReportPath + "\" + "Scatter.png" )

```



Code for this image is using the "Vertical" option with the NewXY function to rotate the x-Axis scale. When axis locations are rotated, x-axis label is placed along left side of image, and y-axis label is p

Links:

[AddScatter](#), [AsSeries](#), [Make](#), [NewXY](#), [resultsReportPath](#), [SetPlotArea](#)

See Also:

4.11 SetBarGapShape

Function must be used with the [AddBarLayer](#) function and it must only be executed after the [AddBarLayer](#) has executed.

When used this function can change the shape of the bar, and optionally it can the gap between bars, and as an additional option it can change the space between bar groups.

Syntax:

```
chart.SetBarGapShape( [Shape], [Gap], [Subgap] )
```

Parameter:	Description:																
Shape	<table border="1"> <thead> <tr> <th>Enter #:</th> <th>Shape Description:</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Square shape</td> </tr> <tr> <td>2</td> <td>Diamond shape</td> </tr> <tr> <td>3</td> <td>Triangle pointing up</td> </tr> <tr> <td>4</td> <td>Triangle pointing right</td> </tr> <tr> <td>5</td> <td>Triangle pointing left</td> </tr> <tr> <td>6</td> <td>Triangle pointing down</td> </tr> <tr> <td>7</td> <td>Circle</td> </tr> </tbody> </table>	Enter #:	Shape Description:	1	Square shape	2	Diamond shape	3	Triangle pointing up	4	Triangle pointing right	5	Triangle pointing left	6	Triangle pointing down	7	Circle
Enter #:	Shape Description:																
1	Square shape																
2	Diamond shape																
3	Triangle pointing up																
4	Triangle pointing right																
5	Triangle pointing left																
6	Triangle pointing down																
7	Circle																
Gap	<p>Sets the gap between the bars in a bar chart layer.</p> <p>Gap between the bars is expressed as the portion of the space between the bars. For example, a bar gap of 0.2 means 20% of the distance between two adjacent bars is the gap between the bars.</p> <p>The portion of the space between the bars, or between bar groups for multi-bar layers, uses a default bar gap = 0.2 or 20%</p>																
Subgap	<p>This argument only applies to multi-bar charts. It is the portion of the space between the bars in a bar group. Gap uses the same decimal process to represent the size of the Subgap spacing between groups</p>																

NOTE:

A Gap is the space between each bar value of the same series. Subgap spacing is the distance between the bars of different series, which is also of a different color. For multi-bar layers (bar layers using the Side data combine method, or for stacked bar layers with multiple data groups), Gap refers to the portion of the space between bar groups, while SubGap refers to the portion of the space between bars within the same bar group.

Example:

```
' ~~~~~
' CREATE a Column / Vertical Bar Chart
' Create graphing space for a horizontal chart 300-Pixel wide,
' & 200-Pixels tall with chart title: "Vertical Columns"
chart.NewXY( "Line Chart", 300, 200 )
```

```
' Size Plotting Area to these values
chart.SetPlotArea( 10, 35, 60, 30 )

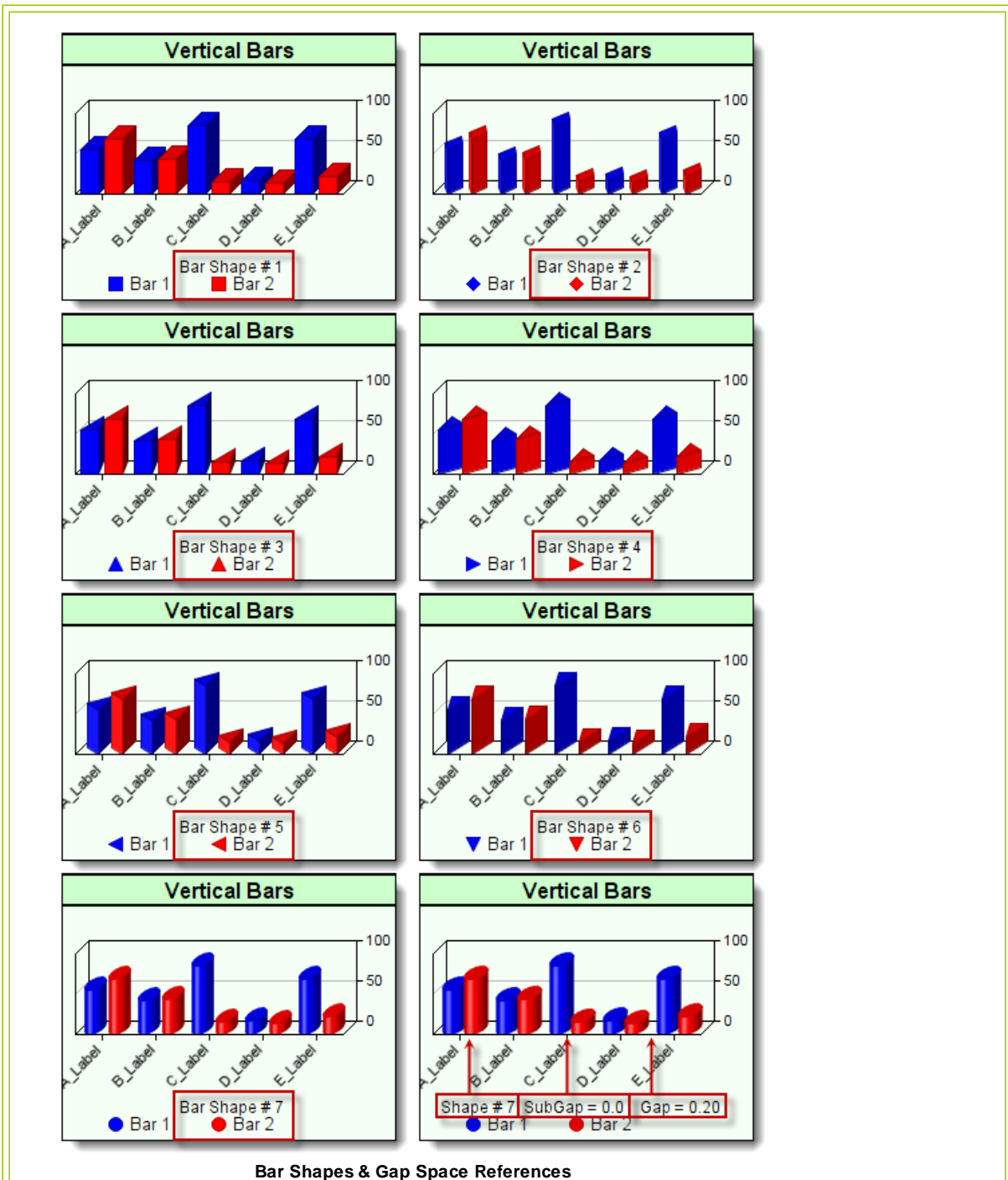
' Use Side-by-Side Bar/Column display
chart.AddBarLayer( 3 )

' Add 5 element values to represent "bar1"
chart.AddBarSeries( AsSeries( bar1 ), 5 )

' Add 5 element values to represent "bar2"
chart.AddBarSeries( AsSeries( bar2 ), 5 )

' Create & Save an image of the chart with
' this file name. BackSlash Character is Required
' when using ResultsReportPath
chart.Make( Test.ResultsReportPath + "\" + "vbar.png" )
```

This Image is a collection of column chart images showing the various shapes. It also shows the Gap and Subgap locations where their size changes the bar spacing:



Links:

[AddBarLayer](#), [AddBarSeries](#), [AsSeries](#), [Make](#), [NewXY](#), [ResultsReportPath](#), [SetPlotArea](#)

See Also:



4.12 SetPlotArea

SetPlotArea is used to control the positioning and size of the plotting area within the boundaries of the image. Each of the four parameters provides the offset distance in pixels from the image edge to graph's plotting area around which space for the axis scales, labels, and legends is provided.

Syntax:

```
chart.SetPlotArea( xLeftOffset, xRightOffset, yTopOffset,
yBottomOffset )
```

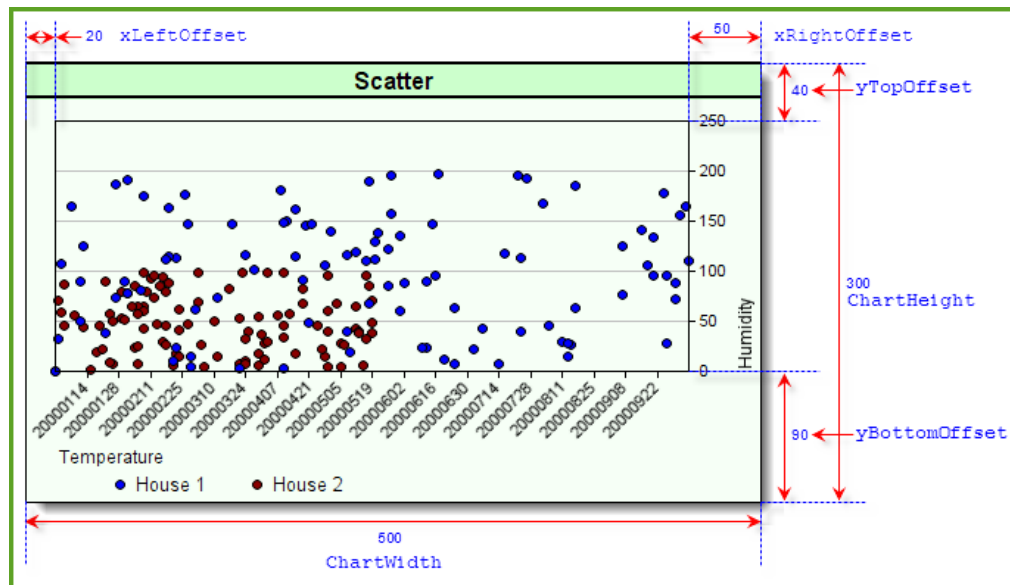
Parameter:	Description:
xLeftOffset	Size of space to offset the graph's left side boundary plotting area from the image's left side edge.
xRightOffset	Pixel size positioning where the graph's right edge graphing boundary border area is placed from the right edge of the graphing image boundary.
yTopOffset	Pixel size positioning of the graph's top edge graphing boundary border area from the top edge of the graphing image boundary.
yBottomOffset	Pixel size positioning of the graph's bottom edge graphing boundary border area from the bottom edge of the graphing image boundary.

Note:

Do not use **SetPlotArea** function with a **NewPie** chart because the plot area is not adjustable.

When trying to determine how many pixels are needed to make room for the characters and numbers in the scales and labels, there is no simple rule available. It isn't available because different Font sizes and different fonts with a different styles require different pixel widths and heights of space to allow the label or scale information to be legible. Pixel spacing is also influenced by the screen's pixel density settings. For example, in this image its size is 500 pixel wide by 300 pixel high. Image was created created in Windows 7 64-Bit using a screen resolution of 1680 x 1050 pixels with 32-bit color. Screen density is 96 pixels per inch set in Landscape mode to fit a 21-inch monitor.

NewXY & SetPlotArea Dimension Locations

**Example:**

```

' ~~~~~
' SCATTER CHART - Script Example from the Scatter Charts Blox
' ~~~~~

' Establish Scatter Chart image size
iChartWidth = 500      ' X-Axis Width
iChartHeight = 300    ' Y-Axis Height
' Create a image
chart.NewXY( "Scatter", iChartWidth, iChartHeight )

' Size the Scatter Dot Plotting area
chart.SetPlotArea( 20, 50, 40, 90 ) <-- Reference Chart Diagram Detail

' Use a BPV stepped string series to send date labels to X-Axis
chart.SetxAxisLabels( AsSeries( LabelSeries ), iRandomRange )
' Place the X-Axis scale label "Temperature" in the left corner.
' Place the Y-Axis label "Humidity" in the lower right corner.
chart.SetAxisTitles( "Temperature", "Humidity" )
' Place the series of 100 dots positioned at element coordinates
' x & y using the BPV "Randomx" element values.
chart.AddScatter( AsSeries( randomx ), AsSeries( randomy ), iNumber, 7,
' Place the series of 100 dots positioned at element coordinates
' x & y using the BPV "House2x" element values.
chart.AddScatter( AsSeries( house2x ), AsSeries( house2y ), iNumber, 7,
' Create & Save this new chart as a file.
' Always add a backSlash Character after "resultsReportPath"
chart.Make( test.resultsReportPath + "\" + "Scatter.png" )

```

Links:

[AddScatter](#), [AsSeries](#), [Make](#), [NewXY](#), [SetxAxisLabels](#), [SetAxisTitles](#)

See Also:

4.13 SetxAxisDates

Function changes the numbered X-Axis scale labels to Dates at the major tick-mark locations. Date range displayed on the chart will cover the entire date range between the simulation Test-Start date and Test-End date.

Syntax:

```
chart.SetxAxisDates( AsSeries(DateSeries), [ElementCount], [Filter], [LabelStep], [Format] )
```

Parameters:	Description:														
DateSeries	Numeric date series, converted using ChartTime. Or if 0 is passed in, the default internal test date/time will be used. Note: Use with all BPV Numeric or String series that are passed to any Chart parameter. AsSeries(...) function conditions the series so the chart function will be able to use the information contained within each of the series elements.														
ElementCount	Number of elements in the date series.														
Filter	Option is dependent upon date data in the supplied series, and chart axis display area. <table border="1" data-bbox="516 919 1448 1234"> <thead> <tr> <th>Use #:</th> <th>Filter & Label Stepping Description:</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Start Of Hour</td> </tr> <tr> <td>2</td> <td>Start Of Day</td> </tr> <tr> <td>3</td> <td>Start Of Week</td> </tr> <tr> <td>4</td> <td>Start Of Month</td> </tr> <tr> <td>5</td> <td>Start Of Year</td> </tr> <tr> <td>6</td> <td>None</td> </tr> </tbody> </table>	Use #:	Filter & Label Stepping Description:	1	Start Of Hour	2	Start Of Day	3	Start Of Week	4	Start Of Month	5	Start Of Year	6	None
Use #:	Filter & Label Stepping Description:														
1	Start Of Hour														
2	Start Of Day														
3	Start Of Week														
4	Start Of Month														
5	Start Of Year														
6	None														
LabelStep	Default is zero, or one, and it will show all the axis labels the scale area allows. A values of 2 will hide two months of a quarter, a value of 6 shows a date label every 6-months.														
Format	"{value mmm-yy}"														

Notes:

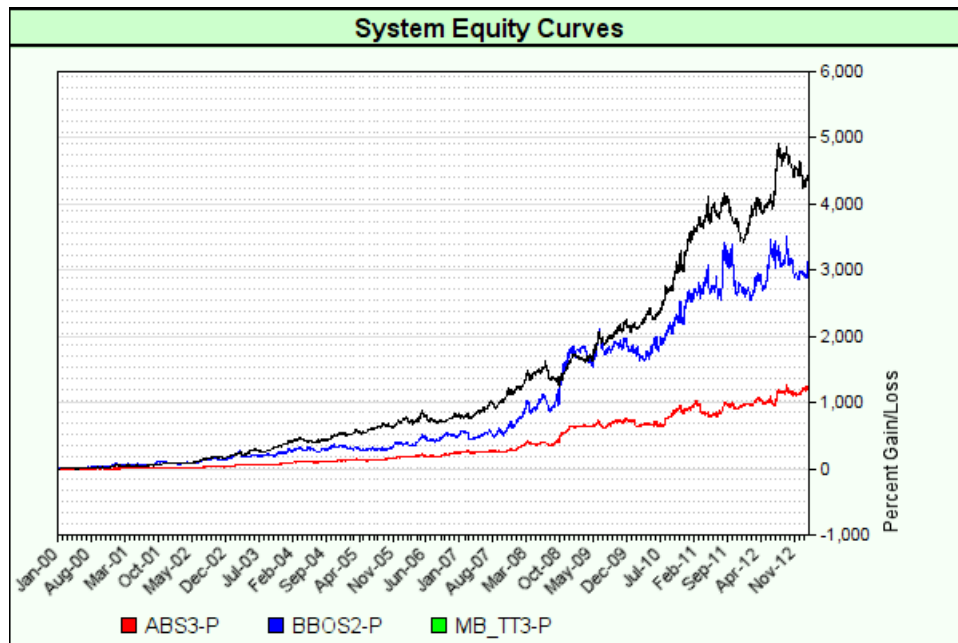
Controlling the stepping of dates is highly dependent upon the amount of area available to display dates.

If using a custom date series, convert each element to ChartTime. As an example, in the After Trading Day script: DateSeries = chartTime(test.currentdate, test.currentTime)

To use the internal default date/time chartTime series, pass in 0 as the date series. Further parameters can still be used for formatting and such.

Example:

```
! ~~~~~
! Multi-Line Chart Example in AddLineSeries Examples:
! ~~~~~
! Set X-Axis Dates to use beginning of month
chart.SetxAxisDates(AsSeries(DateSeries), elementCount, 4)
```

Multiple System Equity Curves.tbx

Links:

[AddLineSeries](#), [AsSeries](#)

See Also:

[ChartTime](#), [Colors](#)

4.14 SetxAxisLabels

This function will place a label along the X-Axis of a chart. It works with charts that have the X-Axis in its normal location below the plot lower boundary area, or along the left side of the chart when the `NewYX` function's optional parameter `"Vertical"` option is applied.

Syntax:

```
chart.SetxAxisLabels( AsSeries(LabelSeries), LabelCount )
```

Parameter:	Description:
LabelSeries	BPV String Series. Note: Use with all BPV Numeric or String series that are passed to any Chart parameter. AsSeries(...) function conditions the series so the chart function will be able to use the information contained within each of the series elements.
LabelCount	Number of labels contained in the BPV String series.

NOTE:

Custom labels are located at the chart's tick-marks, and they are passed to this function using a BPV String series. String series can be auto-indexed, or manually sized and manually indexed, but both indexing types of series must be passed using the `AsSeries` function.

Label names are created as part of the chart's creation code. Their placement on the chart and the space between each label is handled by the logic used to create the label names that the script places into the elements of the series. Space between label when there are lot of data points along the X-Axis is required to prevent the placement of subsequent labels from covering the previous label because the tick marks where the labels are place are too close together.

Label can be applied the chart's X-Axis when it is in its standard position below the chart's plotting area, or when the X-Axis is rotated. When the X-Axis is rotated, the labels will be placed near the tick-mark just outside of the chart's left vertical plotting boundary area.

Example:

```

' ~~~~~
' HORIZONTAL BAR & COLUMN CHARS with Axis Labels
' ~~~~~
' Establish Contour map image size
iChartWidth = 600      ' X-Axis Width
iChartHeight = 300     ' Y-Axis Height

' Create graphing space for a bar chart wide,
' tall with chart title "vertical" option creates
' horizontal bars.
chart.NewXY( "Vertical Bars", iChartWidth, iChartHeight, "" )
'chart.NewXY( "Horizontal Rotated Axis Bars", _
'           iChartWidth, iChartHeight, "Vertical" )

' Size the Bar Plotting area
chart.SetPlotArea( 10, 40, 60, 70 )

' Values entered into each parameter location can change the
' appearance of the layout and 3D depth effect.
chart.AddBarLayer( 3 )

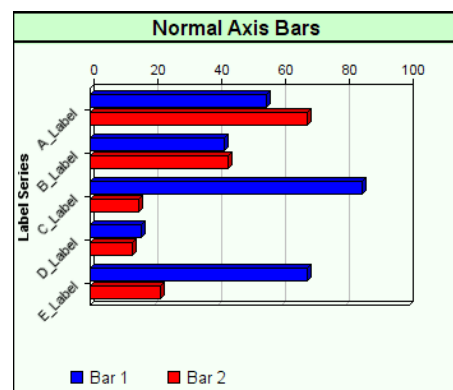
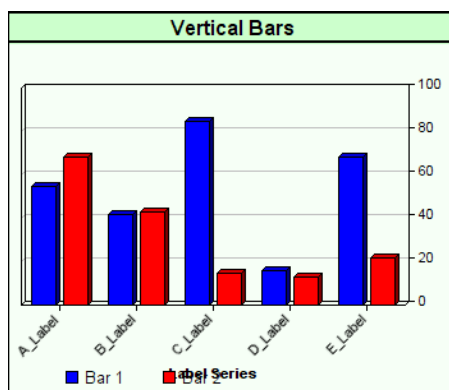
' Add x-Axis Bar Labels
chart.SetxAxisLabels( AsSeries(LabelSeries), iBarCount)

' Add 10 element Bar series data "bar1"
chart.AddBarSeries( AsSeries( bar1 ), iBarCount )

' Add 10 element Bar series data "bar2"
chart.AddBarSeries( AsSeries( bar2 ), iBarCount )

' Create an image of the contour map named:  contour.png
' & save that image into the test results folder
chart.Make( test.resultsReportPath + "\" + "HVbars.png" )

```

**Links:**

[AddBarLayer](#), [AddBarSeries](#), [AsSeries](#), [Make](#), [NewXY](#), [resultsReportPath](#)

See Also:

Section 5 – Email Manager

Trading Blox provides the functions needed to send unsecured, and secured emails. Unsecured email servers are not very popular any longer, so you might be forced to only use the SSL (Secure Socket Layer) encryption connection function to establish a connection. SSL method emails are more secure because of the authentication processes involved that are used with unsecured emails.

To understand what is required, spend time reviewing the functions and the examples, and get the information your email client is required to use when you send an email using a specific email service.

When you have a working email process functioning with Trading Blox you will be able to send text messages and files like the standard Trading Blox files, or a customer position and order file created for how you communicate brokerage orders.

Function Name:	Description:
EmailConnect	connects to the email server
EmailConnectSSL	connects using stunnel SSL
EmailSend	sends an email
EmailSendHTML	sends an html formatted email
EmailDisconnect	disconnects from the email server

5.1 Email Connect

The EmailConnect function connects to the email server.

Syntax:

```
connected = EmailConnect( serverName, [returnEmail], [replyEmail], [userNa
```

Parameter:	Description:
serverName	name of the email server
[returnEmail]	return address
[replyEmail]	reply address
[userName]	username for the email account
[password]	password for the email account
[port]	port

Example:**Returns:**

```
connected =
```

Links:**See Also:**

5.2 EmailConnectSSL

The EmailConnectSSL function connects to the email server using SSL over Stunnel. Required for sending mail through gmail and yahoo mail.

```
// that the only differences between them are the server names
// and the SMTP ports, which is either 465 or 587.
//
// Provider   Port   Server Name
// -----   ----   -
//   GMAIL    465   smtp.gmail.com
//           995   pop.gmail.com
//           993   imap.gmail.com
//
//   HotMail  587   smtp.live.com
//           995   pop3.live.com
//
//   Yahoo    465   plus.smtp.mail.yahoo.com
//           995   plus.pop.mail.yahoo.com
//
//   MS Online 587   smtp.mail.microsoftonline.com
//           995   pop.mail.microsoftonline.com
//
//   Sympatico 587   smtphm.sympatico.ca
//           995   pophm.sympatico.ca
```

Syntax:

```
connected = EmailConnectSSL( serverName, [returnEmail], [replyEmail],
[userName], [password], [port] )
```

Parameter:	Description:
serverName	Name of user's email server
[returnEmail]	Return email address
[replyEmail]	Reply to email address
[userName]	User's "username-ID" for their email account
[password]	Password for the user's email account
[port]	Out-Going SMTP port number

Example:

```
' ~~~~~
' Connect to user's Out-Going mail server
If EmailConnectSSL( "smtp.gmail.com", _
                   "tradingblox@gmail.com", _
                   "tradingblox@gmail.com", _
                   "tradingblox@gmail.com", _
```

```
        "password", 465 ) THEN
    ' When Connection is reported as TRUE,...
    ' Send a HTML email message here:
    EmailSendHTML( "tim@tradingblox.com", _
                  "TradingBlox Orders", "@" + test.orderReportPath )
    ' Send the same HTML email message here:
    EmailSendHTML( "tim@tradingblox.com", "Trading Blox Reports", _
                  "<IMG SRC='cid:message-root.1'>", "", "", "", _
                  "Images/TradingBloxLogo.jpg" )
    ' Send a regular Text-Only email message here:
    EmailSend( "tim@tradingblox.com", "Trading Blox Orders", _
              "Order Report Attached", "", "", test.orderReportPath )
ELSE
    ' Create an Error condition that generates a message
    ERROR ( "Unable to connect to email server" )
ENDIF

    ' Disconnect from the user's out-going email server
    EmailDisconnect()
    ' ~~~~~
```

Returns:

connected = value of true if the connection was successful

Links:**See Also:**

5.3 Email Send

This function sends email using the connection to the email server setup with [EmailConnect](#).

Syntax:

```
EmailSend( toList, subject, message, [cclist], [bcclist], [attachments]
```

Parameter:

Description:

toList	list of email address to send the email to. To send to more than one email address put <> around each address and separate using a semi colon.
subject	subject of the email
message	message body of the email
[cclist]	email cc list
[bcclist]	email bcc list
[attachments]	list of file names to attach. Full path name required. Separate multiple files by a semicolon only -- no spaces.

Example:

```

| ~~~~~
EmailSend( "thewebmaster@tradingblox.net", _
           "Trading Blox Order Message", outputString )

EmailSend( "<thewebmaster@tradingblox.net>;<myBroker@tradingblox.net>", _
           "Trading Blox Order Message", outputString )

EmailSend( "thewebmaster@tradingblox.net", _
           "Trading Blox Order Message", outputString, _
           "", "", "c:\myResults.pdf" )

EmailSend( "thewebmaster@tradingblox.net", "Trading Blox Order Message"
           outputString, "", "", "c:\myResults.pdf;c:\myOrders.csv" )
| ~~~~~

```

Returns:

Links:

See Also:

5.4 EmailSendHTML

This function sends html formatted email using the connection to the email server setup with [EmailConnect](#).

Additional EmailSendHTML Notes:

If the first character of the message (6th argument) or alternate text (8th argument) is a '@', then it is considered as the filename which contains the message to send.

The 'Images' field contains the filenames of images that are to be embedded in the email message. The first image must be referenced in the text of the HTML encode email message as

```
<IMG SRC="cid:message-root.1">
```

The second image (if any) must be referenced as

```
<IMG SRC="cid:message-root.2">
```

Continue in this way for all embedded images.

'AltText' is used to provide a plain [ASCII](#) text equivalent of the message for those email clients that cannot decode HTML.

Syntax:

```
EmailSendHTML( toList, subject, message, _
               [ccList], [bccList], [attachments], [images],
               [alternate text] )
```

Parameter:	Description:
toList	the list of email address to send the email to. To send to more than one email address put <> around each address and separate using a semi colon.
subject	subject of the email
message	message body of the email
[ccList]	email cc list
[bccList]	email bcc list
[attachments]	list of file names to attach. Full path name required. Separate multiple files by a semicolon only -- no spaces.
[images]	Image files to send
[alternate text]	alternate text for plain text viewers

Example:

Returns:

Links:

See Also:

5.5 EmailDisconnect

This function disconnects from the server setup with [EmailConnect](#).

This function should be called before the test completes, in order to close the connection with the email server

Syntax:

```
EmailDisconnect()
```

Parameter:

<none>

Description:**Example:**

```
' ~~~~~  
' Disconnect from out-going SMTP mail server.  
EmailDisconnect()  
' ~~~~~
```

Returns:**Links:****See Also:**

Section 6 – File Manager

Object functions and properties allow the importing and exporting of data by way of external files. Multiple files can be opened at the same time, and there are additional [File & Disk Functions](#) functions available that support task associated with working with data on the disk and its various sub-directory folders.

Functions:	Description:
Close	Closes an open file.
DefaultFolder	Returns the default folder used by the file manager for locating files. This is also the main Trading Blox folder
EndOfFile or EOF	Returns TRUE if the end of the file is reached
CountLines	Returns the number of lines in the file. Takes the file number as a parameter
OpenRead	Opens a file for reading
OpenWrite	Opens a file for writing
PartialLine	Returns TRUE if the entire line was not read fully by ReadLine
ReadLine	Reads a line from a file into a string variable
WriteLine	Writes a string to the file appending a new line character
WriteString	Writes a string to the file without a new line character

See string functions [GetField](#) for use in parsing the input from files. See [GetFieldCount](#) for discovering the number of comma separated fields in the record.

NOTE:

If you are opening multiple files be sure to save the File Numbers into BPV Integer Variables for use in other areas of the module.

If you save a File-Number in a temp Local variable, then it will be lost once you leave the script section.

Normally one would open the files in the Before Test script and close in the After test script. The BPV Integer File Number variables can then be used to read and write from the various files at the same time. When using the OpenAppend function the file can be closed and opened multiple times during the test, since the writing will always be appended to the end of the file.

Links:

[File & Disk Functions](#), [GetField](#), [GetFieldCount](#)

See Also:

Functions:	Description:
Close	closes the file
CopyFile	
CountLines	returns the number of lines in the file. Takes the file number as a parameter
CreateDirectory	
DeleteFile	
EditFile	
FileExists	
FileSize	
MoveFile	
OpenAppend	
OpenFile	
OpenFileDialog	
OpenRead	opens a file for reading
OpenWrite	opens a file for writing
PartialLine	returns TRUE if the entire line was not read fully by ReadLine
ReadLine	reads a line from a file into a string variable
WriteLine	writes a string to the file appending a new line character
WriteString	writes a string to the file without a new line character

Functions:[OpenRead](#)[OpenWrite](#)[Close](#)[WriteLine](#)[WriteString](#)[ReadLine](#)[EndOfFile](#) or [EOF](#)

PartialLine

Description:

opens a file for reading

opens a file for writing

closes the file

writes a string to the file appending a new line character

writes a string to the file **without** a new line character

reads a line from a file into a string variable

returns TRUE if the end of the file is reached

returns TRUE if the entire line was not read fully by ReadLine

defaultFolder	returns the default folder used by the file manager for locating files. This is also the main Trading Blox folder.
CountLines	returns the number of lines in the file. Takes the file number as a parameter

6.1 Close

The Close function closes a text file that has previously been opened using [OpenRead](#) or [OpenWrite](#). Pass in the optional file number if more than one file has been opened.

Syntax:

```
fileManager.Close( fileNumber )
```

Parameter:

file number

Description:

File number of each file opened

Returns:

<none>

Example:

```
' ~~~~~
VARIABLES: lineString Type: String
VARIABLES: fileNumber Type: Integer
' ~~~~~
' Open the file.
fileNumber = fileManager.OpenWrite( "C:\FileToOpen.txt" )

' If file-number > 0, . . .
If fileNumber > 0 THEN
  ' Create text line using the symbol, a comma delimiter
  ' and the instrument's close price
  lineString = instrument.symbol + "," + instrument.close

  ' Write the line to the file identified by the file-number.
  fileManager.WriteLine( fileNumber, lineString )

  ' Close the file identified by the file-number.
  fileManager.Close( fileNumber )
ENDIF
' ~~~~~
```

Links:

[Close](#), [OpenWrite](#), [WriteLine](#)

See Also:

[File Manager](#)

6.2 CountLines

Function will return the number of records in a [FileManager](#) object opened file by using the file's file number to access the file.

Syntax:

```
iRecs = fileManager.CountLines( iFilNum )
```

Parameter:

iFilNum

Description:

Integer value provided by the FileManager's Open methods.

Returns:

Number of records contained in the file.

Example:

```
' ~~~~~
VARIABLES: sFullPathFileName, sPathName, sFileName   Type: String
VARIABLES: iFilNum, iRecs                          Type: Integer
' ~~~~~
' Combine Path + "\" + File-Name into a Full Path-File statement
sFullPathFileName = sPathName + sFileName

' If the file Exist, . . .
If FileExists( sFullPathFileName ) THEN
' Open File to obtain its File Number
iFilNum = fileManager.OpenRead( sFullPathFileName )
' If a File number is assigned, . . .
If iFilNum THEN
' Count the Records contained in the file
iRecs = fileManager.CountLines(iFilNum)
ENDIF ' iFilNum
ENDIF ' FileExists
' ~~~~~
```

Links:

[FileExists](#), [OpenAppend](#), [OpenRead](#), [OpenWrite](#)

See Also:

[FileManager](#), [File & Disk Functions](#)

6.3 DefaultFolder

Function provides the full path location of Trading Blox.

Syntax:

```
' Get Trading Blox Installation Path  
sTBPath = fileManager.DefaultFolder
```

Parameter:

<none>

Description:**Returns:**

Assigns the text containing the Full Path of the installed Trading Blox location.
i.e. C:\Trading Blox\

Example:

```
' ~~~~~  
VARIABLES: sTBPath      Type: String  
' ~~~~~  
' Get Trading Blox Installation Path  
sTBPath = fileManager.DefaultFolder
```

Links:**See Also:**

[File Manager](#)

6.4 EndOfFile

The EndOfFile function returns TRUE if the end of the file is reached after a sequence of [ReadLine](#) calls while reading a file previously opened with [OpenRead](#).

Syntax:

```
endReached = fileManager.EndOfFile( fileNumber )
```

Parameter:

fileNumber

Description:

File-number identified by the number value.

Returns:

endReached is set to TRUE when the End-of-File marker is reached.

Example:

```
' ~~~~~
VARIABLES: lineString      Type: String
VARIABLES: fileNumber      Type: Integer
' ~~~~~
' Open file for Read Only access.
fileNumber = fileManager.OpenRead( "C:\FileToOpen.txt" )

' When a file-number > 0 is available, . . .
If fileNumber > 0 THEN
  ' Loop reading lines until we reach the
  ' end of the file marker.
  Do UNTIL fileManager.EndOfFile( fileNumber )
    ' Read a line from the current file.
    lineString = fileManager.ReadLine( fileNumber )

    ' Print the line
    PRINT lineString
  LOOP

  ' Close the file.
  fileManager.Close( fileNumber )
ENDIF ' fileNumber > 0
' ~~~~~
```

Links:

[OpenRead](#), [ReadLine](#)

See Also:

[File Manager](#)

6.5 OpenAppend

The OpenAppend function opens a text file for writing. If the file already exists, the file will be opened for writing at the end of the file and any previous information will not be erased.

If the file cannot be opened, the function will return false.

To add more records to the file use the [WriteLine](#) and [WriteString](#) functions.

If a colon is included in the path, then the path is literal and used as is, so you can open any file in any folder. If a colon is not included, then the path is relative to the Trading Blox directory. If a // is included, then the path is literal so you can access any file on any folder on any server.

If the file name includes a colon, then the file name is considered a full path, and that full path will be used.

If the file name starts with \\ then the file name is considered a full path to a server location.

If the file name does not contain a colon or \\, then the file name is considered relative to the main Trading Blox folder.

Syntax:

```
fileNumber = fileManager.OpenAppend( fileName )
```

Parameter:

fileName

Description:

Full Path & File-Name of the file to open

Returns:

fileNumber = Trading Blox assigned file identification number of the file opened so additional information can be added.

Example:

```
' ~~~~~
VARIABLES: fileName      Type: String
VARIABLES: fileNumber   Type: Integer
' ~~~~~
' Open file named so additional information can be appended.
fileNumber = fileManager.OpenAppend( fileName )
```

Links:

[OpenRead](#), [OpenWrite](#), [WriteLine](#), [WriteString](#)

See Also:

[File Manager](#)

6.6 OpenRead

The `OpenRead` function opens a text file for reading. If the file is opened successfully, the File Number is returned. The file can then be read using the [ReadLine](#) function.

If the file does not exist, the function returns `FALSE` or Zero.

If a colon is included in the path, then the path is literal and used as is, so you can open any file in any folder. If a colon is not included, then the path is relative to the Trading Blox directory. If a `//` is included, then the path is literal so you can access any file on any folder on any server.

If the file name includes a colon, then the file name is considered a full path, and that full path will be used.

If the file name starts with `\\` then the file name is considered a full path to a server location.

If the file name does not contain a colon or `\\`, then the file name is considered relative to the main Trading Blox folder.

Syntax:

```
fileNumber = fileManager.OpenRead( fileName )
```

Parameter:

fileName

Description:

Name of the file to open

Returns:

fileNumber = File identification number when file is successfully opened for Read only access.

Example:

```
' ~~~~~
VARIABLES: fileNumber   Type: Integer
' ~~~~~
' Open the file.
If fileManager.OpenRead( "C:\FileToOpen.txt" ) THEN
  PRINT "File Opened Successfully"
ENDIF
```

OR

```
' Open the file.
fileNumber = fileManager.OpenRead( "C:\FileToOpen.txt" )

If fileNumber > 0 THEN
  PRINT "File Number", fileNumber, "Opened Successfully"
ENDIF
' ~~~~~
```

Links:

See Also:



6.7 OpenWrite

Function opens a text file for write only access using the [WriteLine](#) and [WriteString](#) functions.

If the file already exists, any information contained in the file will be erased.
If the file cannot be opened, the function will return FALSE or Zero.

If a colon is included in the path, then the path is literal and used as is, so you can open any file in any folder. If a colon is not included, then the path is relative to the Trading Blox directory. If a // is included, then the path is literal so you can access any file on any folder on any server.

If the file name includes a colon, then the file name is considered a full path, and that full path will be used.

If the file name starts with \\ then the file name is considered a full path to a server location.

If the file name does not contain a colon or \\, then the file name is considered relative to the main Trading Blox folder.

Syntax:

```
fileNumber = fileManager.OpenWrite( fileName )
```

Parameter:

fileName

Description:

Name of the file to open

Returns:

fileNumber = File identification number required to access the file when opened successful; Zero

Example:

```
' ~~~~~
VARIABLES: fileNumber   Type: Integer
VARIABLES: lineString Type: String
' ~~~~~
' Open the file.
fileNumber = fileManager.OpenWrite( "C:\FileToOpen.txt" )

If fileNumber > 0 THEN
  ' Construct the line.
  lineString = instrument.symbol + "," + instrument.close

  ' Write out the line to the file.
  fileManager.WriteLine( fileNumber, lineString )

  ' Another way to write the same thing as above.
  fileManager.WriteLine( fileNumber, instrument.symbol, instrument.close

  ' Close the file.
  fileManager.Close( fileNumber )
ENDIF ' fileNumber > 0
' ~~~~~
```

Links:

See Also:

6.8 PartialLine

Function will return TRUE when the line being read is longer than the maximum [String](#) length of 512 characters. This character length restriction requires the function to report a TRUE condition so the programming will know when there are more character that are available. Access to the additional characters is made available by performing additional [ReadLine](#) executions of the same record value.

Syntax:

```
iNotAllRecordData = fileManager.PartialLine
```

Parameter:

<none >

Description:**Returns:**

iNotAllRecordData is TRUE when record could NOT read the entire record's

Example:

```
' Read indexed record as a string
sRecord = fileManager.ReadLine( iFilNum, x )
' Did ReadLine Fail to get the entire record
iNotAllRecordData = fileManager.PartialLine
```

Links:

[ReadLine](#), [String](#)

See Also:

6.9 ReadLine

Function reads and returns all the characters from a file's record that was previously opened with the [OpenRead](#) function. When ReadLine will read each line in the file and return its text contents until it reaches the End-of-Line marker sequence.

This call can be used in conjunction with the [GetFieldCount](#) to identify how many fields are in a comma delimited text file, and use the [GetField](#) functions to use that field count information to extract values the text record in the `lineString` variable supplied by the [ReadLine](#) function's file access.

[ReadLine](#) will only read the first 512 characters of the line. If there are more characters then the `fileManager.PartialLine` flag will be set to true, and the next call to [ReadLine](#) will return the remaining characters.

When not using the optional `lineIndex` parameter, the file is read sequentially. When using the `lineIndex` parameter the file can be read random access. But note that this later method is much slower.

Syntax:

```
lineString = fileManager.ReadLine( fileNumber, [lineIndex] )
```

Parameter:

`fileNumber`

Description:

File identification number of open file to be read.

`[lineIndex]`

Optional line index to return, or when used to access a record, it will read the record at the index value assigned.

Returns:

`lineString` = Text contents of the record read from the file.

Example:

```
' ~~~~~  
VARIABLES: lineString Type: String  
VARIABLES: fileNumber Type: Integer  
' ~~~~~  
' Open the file.  
fileNumber = fileManager.OpenRead( "C:\FileToOpen.txt" )  
  
If fileNumber > 0 THEN  
' Loop reading lines until we reach the  
' end of the file.  
Do UNTIL fileManager.EndOfFile( fileNumber )  
' Read a line from the current file.  
lineString = fileManager.ReadLine( fileNumber )  
  
' Print the line  
PRINT lineString  
LOOP  
  
' Close the file.  
fileManager.Close( fileNumber )  
ENDIF  
' ~~~~~
```

Links:

[DO](#), [GetField](#), [GetFieldCount](#), [EndOfFile](#), [PartialLine](#), [OpenRead](#)

See Also:

[File Manager](#)

6.10 WriteLine

Function writes an optional list of string expressions to a file that was previously opened with [OpenWrite](#) function, and then writes the Windows End-of-Line character sequence after all the expressions have been written.

If no expressions, or numeric variables are supplied, then [WriteLine](#) just writes the End-of-Line character sequence.

For Windows, the End-of-Line sequence is [ASCII](#) character 10 for Line Feed. Windows text file editors like NotePad will interpret this sequence as the start of a new line. If more than one parameter is passed, they are separated by commas for ease of using CSV formatting to simplify file use in spreadsheet programs.

Syntax:

```
fileManager.WriteLine( fileNumber, [ expression , expressionTwo ... ] )
```

Parameter:

fileNumber

Description:

file number if more than one file has been opened.

[expression , expressionTwo ...]

the first string expression to write (optional) and second string expression to write (optional)

Returns:

nothing is returned

Example:

```
' ~~~~~
VARIABLES: lineString Type: String
VARIABLES: fileNumber Type: Integer
' ~~~~~
' Open the file.
fileNumber = fileManager.OpenWrite( "C:\FileToOpen.txt" )

If fileNumber > 0 THEN
  ' Construct the line.
  lineString = instrument.symbol + "," + instrument.close

  ' Write out the line to the current file.
  fileManager.WriteLine( fileNumber, lineString )

  ' Another format for the above.
  fileManager.WriteLine( fileNumber, instrument.symbol, instrument.close )

  ' Close the file.
  fileManager.Close( fileNumber )
ENDIF ' fileNumber > 0
' ~~~~~
```

Links:

[Close](#), [OpenWrite](#)

See Also:

[File Manager](#)

6.11 WriteString

The WriteString function writes a list of string expressions to a file that was previously opened with [OpenWrite](#). It does not write the windows end-of-line character like [WriteLine](#) does. If multiple parameters are passed, they are not separated by commas, so you can control the exact characters being output.

Syntax:

```
fileManager.WriteString( fileName, expression [, expressionTwo ... ] )
```

Parameter:	Description:
fileName	file number if more than one file has been opened
expression	the first string expression to write
[, expressionTwo...]	the second string expression to write (optional)

Returns:

Nothing is returned.

Example:

```
' ~~~~~
VARIABLES: lineString Type: String
VARIABLES: fileName Type: Integer
' ~~~~~
' Open the file.
fileName = fileManager.OpenWrite( "C:\FileToOpen.txt" )

If fileName > 0 THEN
  ' Write out the date to the current file.
  fileManager.WriteString( fileName, instrument.date, "," )

  ' Construct another line.
  lineString = instrument.symbol + "," + instrument.close

  ' Write out the line to the current file.
  fileManager.WriteString( fileName, lineString )

  ' Write out the end-of-line character sequence.
  fileManager.WriteLine( fileName )

  ' Close the file.
  fileManager.Close( fileName )
ENDIF ' fileName > 0
' ~~~~~
```

Links:

[Close](#), [OpenWrite](#)

See Also:

[File Manager](#)

Section 7 – Instrument

All instrument data is specific to each system's portfolio, and the instrument specific data that is generated as a result of the simulation.

Additional specific instrument information can be added by a user creating a new IPV variable name that will be in context when that specific instrument time arrives in the instrument rotation that happens in all instrument specific script sections. Added external data can be loaded into specific instruments by looping through the instruments before it is needed in the simulation. Scripts like Before Test and Before Trading Day are often used to load instrument information in a bulk processing of all instruments.

In most cases where the same symbol data is used in suites hosting multiple system, the actual file data in each instrument's file will be the same. Although prices are the same for an instrument over all systems, the positions and trades are often different over multiple systems. This means each instrument's object data collection is a specific instance of information that is only representative of that instrument's information in that system. When trying to make a decision with your scripting code to use an IPV or a BPV, the decision should be based upon whether the value being used is specific to the instrument, or universal for all instruments. When it is specific to an instrument, use an IPV; when the same value is to be applied to all instruments, use a BPV.

Added information can be assigned during execution, or it can be assigned at the initialization of a test. How to decide when added information is assigned to an instrument should be based upon how soon the information will be needed to affect system operations, or whether the repeated calculations done ahead of time to create a static test value make sense to do them once. In most cases the decision is fairly simple, and the process of assigning information ahead of time is easy using the System function [LoadSymbol](#).

Most of the time [LoadSymbol](#) is used in script where the instruments will not have automatic context. These script section, shown in the next table, only execute once for each test record, whereas a script section where instruments have automatic context are executed once for each instrument of each test record.

Scripts Without Automatic Instrument Context:

Script Name:	Usual Sequence:
Before Simulation	1
Before Test	2
Before Trading Day	3
Before Bar	4
Before Order Execution	5
After Bar	6
Initialize Risk Management	7
Compute Risk Adjustment	8
After Trading Day	9
After Test	10

After Simulation

11

In scripts where multiple instrument must be processed and where automatic instrument context is not supported must use [LoadSymbol](#) and then use a looping structure like a [For Next](#) area so that each instrument can be loaded and processed within the looping structure.

An instrument can be loaded into a special type of BPV object structure using the [LoadSymbol](#) function. In a BPV dialog there is a data type named '[instrument](#)'.

Block Permanent Variable

Name for Code: Mkt Any Name can be used OK

Name for Humans: Re-usable IPV Instrument Structure Cancel

Defined Externally in Another Block

Variable Type:

- Integer - whole number values e.g. 1, -2, 400, 5, etc.
- Floating Point - fractional number e.g. 2.5, 1.414, etc.
- Price - fractional number in the range of prices
- String - "Hello", "Goodbye", etc.
- Series - a series or list of numbers
- Series - a series or list of strings
- Instrument - used to load and access alternate markets

Variable Options:

Default Value: 0

Scope: Block (selected), System, Test, Simulation

Data assigned to any IPV variable will be still there when accessed using the IPV variable name. Information last assigned to this BPV name will be accessible as long as it hasn't been replaced.

This is why the [LoadSymbol](#) function can take a symbol and a system's information as parameters. When both are used, the reach of the [LoadSymbol](#) process can extend outside of the module where the work is being performed.

When a BPV Instrument Type variable is used to load an instrument's information, that name can be used elsewhere in the module to reference the values of the last loaded instrument. In some cases a series of BPV - Instrument types are created with recognizable names so they can each load a different instrument and provide simple access to that instrument's information whether it is in context or not where it is needed.

Scripts where instrument context is automatic execute each script once for each instrument for each test bar within the test period range. Instrument that don't have a trade data record with a test record location will allow access to the previous instrument record when it exists, or will just skip the instrument when there is no previous record is available. In these scripts, most often when an instrument that doesn't have context at the time of when the instrument brought into context must be accessed, then no looping structure is needed unless more than one instrument needs to be accessed.

Scripts with Automatic Instrument Context:

Script Name:	Usual Sequence:
Rank Instruments	1

Filter Portfolio	2
Before Instrument Day	3
Exit Orders	4
Entry Orders	5
Unit Size	6
Can Add Unit	7
Update Indicators	8
Can Fill Order	9
Exit Order Filled	10
Entry Order Filled	11
After Instrument Open	12
Adjust Stops	13
After Instrument Day	14
Compute Instrument Risk	15
Adjust Instrument Risk	16

For most common usage, in the Entry and Exit scripts for example, the instrument object is setup for the default instrument being processed, and for the test in which the script is executing.

BPV instrument objects can be defined in scripting, and will have access to the same functions and properties as the default instrument object.

- [Correlation Properties](#)
- [Correlation Functions](#)
- [Data Access Properties](#)
- [Data Functions](#)
- [Group Properties](#)
- [Historical Trade Properties](#)
- [Instrument Loading Functions](#)
- [Position Functions](#)
- [Position Properties](#)
- [Ranking Functions](#)
- [Ranking Properties](#)
- [Trade Control Properties](#)
- [Trade Control Functions](#)

If you want to save a value that is independent of instruments, use a Block Permanent Variable (see [Variables and Types](#)).

Instrument properties and functions can be accessed using the '.' syntax:

Example:

```
variable1 = instrument.date
```

or if myTempInstrument is defined as a BPV **Instrument** object:

```
variable1 = myTempInstrument.date
```

Links:**See Also:**

7.1 Data Properties

Bar Indexing

Properties listed with a '[']' following them may be indexed using a number which determines which day's data to access. There are also built in constants for 'today' and 'yesterday' which can be used. For example:

```
yesterdaysDate = instrument.date[ 1 ]
```

OR

```
yesterdaysDate = instrument.date[ yesterday ]
```

will access yesterday's date.

If no index is used (i.e. like instrument.date above) then the value will return the current bar. This value will be set to yesterday's bar for scripts which are run before the trading day and to today's bar for scripts run after the current date has been set.

Data Access Properties:

Property:	Description:
activeStatus	optional value for stocks, A for active and I for inactive
averageVolume	<p>Current 5 day EMA of the unAdjustedVolume. Used internally by Trading Blox for volume filters such as max volume per trade and minimum volume.</p> <p>This property is computed dynamically during the simulation run and can be used in scripting as needed. This property cannot be used in Calculated Indicators because all values for all calculated indicators are computed before the simulation starts running, and this property has not been computed yet.</p> <p>Test computed indicators do not support look-back references. However, their calculated result from each instrument bar can be stored in an IPV series that will support look-back referencing if that process is coded into the blox.</p>
bar	the bar number for the current date. Bar 1 is the first bar of data loaded. The start of the test is likely not Bar 1.
bigPointValue	usually 1.00 for Stocks, unless the Convert Profit by Stock Splits global is on. In that case the big point value is the unadjusted close divided by the adjusted close for any given day. For Futures, as set in the Futures Dictionary and adjusted by the currency conversion if there is one. Does not change for futures. For Forex, the current value in dollars for the pair. For Forex, this number can change each day.
brokerSymbol	the broker symbol as defined in the dictionary
close[]	the close for the specified bar
conversionRate	The conversion rate of foreign denominated futures and stocks. This respects the "reverse conversion" checkbox in the forex dictionary.
currency	The currency in which the future is denominated. Set in the Futures or Stock Dictionary.

currencyBorrowRate	The borrow rate of the currency.
currencyDate	The current date of the currency converter, if present.
currencyLendRate	The lending rate of the currency.
currencyTime	The current time of the currency converter, if present.
currentBar	the bar number minus the startBar plus one.
currentWeek	the weekIndex minus the startWeek plus one.
dataLoadedBars	total number of bars of data loaded and cached.
dataVendorID	optional value for stocks, the data vendor id
date[]	the date value for the specified bar. In YYYYMMDD format.
dayClose [dayIndex]	the close of the current day as of the current time
dayHigh [dayIndex]	the high of the current day as of the current time.
dayIndex	the current day index for use with the day series which is derived from intraday data
dayLow[dayIndex]	the low of the current day as of the current time
dayOpen [dayIndex]	the open of the current day
dayVolume [dayIndex]	the volume of the current day as of the current time
defaultAverageTrueRange	<p>Current internal computation of the 39 day average true range. Used internally for slippage calculations.</p> <p>This property is computed dynamically during the simulation run and can be used in scripting as needed. This property cannot be used in Calculated Indicators because all values for all calculated indicators are computed before the simulation starts running, and this property has not been computed yet.</p> <p>Test computed indicators do not support look-back references. However, their calculated result from each instrument bar can be stored in an IPV series that will support look-back referencing if that process is coded into the blox.</p>
deliveryMonth	the delivery month of the contract represented by the data -- format: YYYYMM (futures only)
deliveryMonthLetter	the delivery month letter (Z for December, etc)
description	the description from the appropriate dictionary for this symbol
displayDigits	the number of digits to the right of the decimal, as set in the dictionary.
dividend	the dividend for the current bar
endBar	the bar number for the last day of testing for this instrument. Not system specific.
endDate	the end date of testing for this instrument. Not system specific. Can be different than the lastDataLoadedDate if the end testing date changes to an earlier date after the data is loaded and cached.

exchange	the instrument's exchange
extraData1[] ... extraData8[]	the value of any optional extra data appended to the data file for the specified bar. There are up to 8 extra data fields you can use with this format.
fileName	the filename of the instrument
firstDataLoadedDate	The first date of the data loaded and cached for this instrument. Not system specific.
folder	the folder location of the file
forexBaseBorrowRate	the borrow interest rate of the Base side of the forex pair
forexBaseLendRate	the lending interest rate of the Base side of the forex pair
forexPipSize	the pip size of the forex market as set in the forex dictionary with 7-decimal maximum size
forexPipSpread	the pip spread as set in the forex dictionary
forexQuoteBorrowRate	the borrow interest rate of the Quote side of the forex pair
forexQuoteLendRate	the lend interest rate of the Quote side of the forex pair
high[]	the high for the specified bar
inPortfolio	returns TRUE if the instrument is in the system's portfolio. returns FALSE if the instrument is a supporting forex file or loaded using LoadSymbol and not in the portfolio.
intradayData	returns TRUE if the instrument is using intraday data
isForex	returns TRUE if the instrument is a forex
isFuture	returns TRUE if the instrument is a future
isPrimed	returns TRUE if the instrument is primed
isStock	returns TRUE if the instrument is a stock
julianDate[]	the number of days since 1900 for the current bar
lastBarOfDay	returns TRUE if the bar is the last bar in the day
lastDataLoadedDate	The last date of the data loaded and cached for this instrument. Not system specific.
lastDayOfMonth	returns TRUE if the bar is the last bar in the month
lastDayOfWeek	returns TRUE if the bar is the last day in the week
lastDayOfYear	returns TRUE if the bar is the last bar in the year
lastTradingInstrument	returns TRUE if the instrument is the last trading instrument for the trading day.
low[]	the low for the specified bar
margin	the margin requirement for a futures instrument as set in the Futures Dictionary. Not used for stocks or forex.
minimumTick	the amount of the minimum tick in points. For futures this is set in the Futures Dictionary. For stocks this is .01 divided by the stock split adjustment, which is calculated as the unadjusted close divided by the adjusted close. In this way, the actual minimum tick for the time period can be determined.

minimumVolume	the minimum volume setting from global parameters. Uses the stock minimum value for stocks, and the futures minimum volume for futures. Forex returns minimum volume of zero.
monthClose [monthIndex]	the close of the current calendar month, as of the current day
monthHigh [monthIndex]	the high of the current calendar month, as of the current day
monthIndex	the current month index for use with the month series
monthLow [monthIndex]	the low of the current calendar month, as of the current day
monthOpen [monthIndex]	the open of the current calendar month
nativeBPV	the native currency big point value, as set in the dictionary
negativeAdjustment	for back-adjusted data that goes below zero (eg CL) all prices are raised so that no price will be negative. This is the amount by which the prices are raised. Normally you don't need this since the debugger prices, trade prices, and order generation prices are all converted back to normal prices. But if you need the actual price for calculations in the script, or to print the value, then you would subtract this amount.
open[]	the open for the specified bar
openInterest[]	the open interest for the specified bar (if available)
orderSortValue	the order sort value as entered in the Futures Dictionary
referenceID	This property is a special object pointer that is used by custom DLL extension functions to access instrument object information.
priorityIndex	the numerical order of the instruments used in a system. For futures, the order is based on the order ranking in the futures dictionary. For stocks and Forex, the order is alphabetical. Each system will have its own ranking of instruments. All scripts that loop over instruments will do so in this order.
roundLot	returns the round lot of the instrument, as set in the dictionary
savedWFPProfit	The Walk Forward process saves open positions from one OOS test to the next. For Forex, the profit is saved as well to help compute the overall profit of the combined OOS tests. This value is available in the After Test script for debugging purposes.
startBar	the bar number of the first day of testing for this instrument taking into consideration the priming required for this instrument for this system.
startDate	the start date of testing for this instrument and system taking into consideration priming. Is usually different than the firstDataLoadedDate.
startWeek	the weekIndex of the startBar
stockSplitRatio	The ratio of the unadjusted close to the adjusted close. When Convert Profit by Stock Splits global is on, then the profit is multiplied using this ratio on trade entry date vs. trade exit date, to account for the increase in shares due to the splits during the course of the trade.
symbol	the instrument's trading symbol, e.g. S, IBM, CL
systemClosedEquity	the current system closed equity for the instrument
systemOpenEquity	the current system open equity for the instrument

<code>systemTotalEquity</code>	the current total system profit/loss for the instrument
<code>testClosedEquity</code>	the current test closed equity for the instrument
<code>testOpenEquity</code>	the current total test open equity for the instrument
<code>testTotalEquity</code>	the current total test profit/loss for the instrument
<code>time[]</code>	the time value for the specified bar. 0 if daily data. In HHMM format.
<code>tradeDayOpen</code>	the open for tomorrow. Useful in the Entry script to know the open for the trade day.
<code>tradesOnTradeBar</code> (New)	returns TRUE if the instrument trades on the current trading date/time. Works for intra day as well as daily systems to confirm if there is a bar of data for the current test date/time. Important to use when excluding holidays from a computation.
<code>tradesOnTradeDate</code> (Obsolete)	
<code>tradingMonths</code>	the trading months list defined in the Futures Dictionary. Used only for accounting for contract rolls estimation, when the data does not have the delivery month.
<code>unadjustedClose</code>	the actual close price unadjusted for contract merging (futures), splits, or dividends (stocks)
<code>unAdjustedVolume</code>	the volume for stocks, unadjusted by stock splits. Typically the the raw OHLC and V in the data series are all adjusted for stock splits.
<code>usedMargin</code>	the total margin used for the current open position. This is purchase equity for stocks and total margin for futures.
<code>volume[]</code>	the volume for the specified bar
<code>weekClose</code> [weekIndex]	the close of the current calendar week, as of the current day
<code>weekHigh</code> [weekIndex]	the high of the current calendar week, as of the current day
<code>weekIndex</code>	the current weekIndex used for the week series.
<code>weekLow</code> [weekIndex]	the low of the current calendar week, as of the current day
<code>weekOpen</code> [weekIndex]	the open of the current calendar week

Links:[Data Functions](#)**See Also:**

7.2 DataFunctions

The following functions can be used to round or format price data and to load external data:

Function Name:	Description:
AddCommission	adds the specified amount to the commission to the specified unit of the current position for the instrument.
Extract (fileName, startDate, endDate)	Extracts all indicators and IPV Series variable by day and by instrument to a file.
GetDateTimeIndex	Returns the time index within a date.
GetDayIndex	returns the day index for a date
PriceFormat	returns the price as a string formatted for printing. For example, a price for Soybeans of 6.25 will return "6 1/4". Does not round to tick.
RealPrice	returns the real price, converted back from being negative adjusted.
RoundTick	Rounds the specified price to the nearest tick
RoundTickDown	returns the price rounded down to the next tick
RoundTickUp	returns the price rounded up to the next tick
RoundTick	Rounded_Price = (price + minimumTick / 2)

Links:

[Data Properties](#)

See Also:

AddCommission

Adds commission per share/contract to the specified unit of the current position for the instrument. If there is no current position or the unit number is out of range, then it will return an error.

Syntax

```
instrument.AddCommission( [unitNumber], commission )
```

Parameters

unitNumber	the unit number to which the commission will be added
commission	the commission amount to add per share or contract, in base system currency

Example of adding commission when an order is first entered. Place in the Entry Order Filled Script

```
' Add $12 in commission per contract or share to this new unit.
' If there are 5 contracts or shares, the total commission added will
be $60.

instrument.AddCommission( instrument.currentPositionUnits, 12 )
```

Extract

Syntax:	
<code>Extract(fileName , startDate , endDate)</code>	
Parameter:	Description:
Returns:	
Example:	
Links:	
See Also:	

GetDateTimeIndex

Returns the bar index of the date and time. This bar index synchronizes with the instrument.bar. To find the close for a bar Index returned, subtract from instrument.bar and use as a lookback index.

Returns -1 if the date time is not found in the instrument series.

Syntax

```
barIndex = instrument.GetDayIndex( date, time )
```

Parameters

<i>date</i>	the date in YYYYMMDD format.
<i>time</i>	the time in HHMM

Example

```
barIndex = instrument.GetDayIndex( 20060101, 1130 )

IF barIndex <> -1 THEN
  barClose = instrument.close[ instrument.bar - barIndex ]
  barDate = instrument.date[ instrument.bar - barIndex ]
ENDIF

PRINT barClose, barDate
```

GetDayIndex

Returns the bar index of the date and optional time. This bar index synchronizes with the instrument.bar. To find the close for a bar Index returned, subtract from instrument.bar and use as a lookback index.

Returns -1 if the date is not found in the instrument series.

Syntax

```
barIndex = instrument.GetDayIndex( date, [time] )
```

Parameters

<i>date</i>	the date in YYYYMMDD format.
<i>time</i>	the optional time in HHMM format

Example

```
barIndex = instrument.GetDayIndex( 20060101 )

IF barIndex <> -1 THEN
  barClose = instrument.close[ instrument.bar - barIndex ]
  barDate = instrument.date[ instrument.bar - barIndex ]
ENDIF

PRINT barClose, barDate
```

PriceFormat

Returns the price formatted for printing. For example, a price for Soybeans of 6.25 will return "6 1/4".

Note that this function does not change the value of the price, and if a tick value is required RoundTickUp and/or RoundTickDown can be used to adjust the price prior to using this function.

This function returns a string value, and cannot then assigned to a floating point variable or used in any computations. The return value should only be used for Printing or Reporting purposes.

Syntax

```
roundedValue = PriceFormat( price )
```

Parameters

<i>price</i>	the price to be formatted
<i>return value</i>	Returns a STRING type variable

Example

```
' Print the price.
PRINT "Price = ", instrument.PriceFormat( entryPrice )
```

RealPrice

Returns the real price as read from the data file. This can be different from the price used in scripting if the data has been negative adjusted.

Syntax

```
rPrice = RealPrice( price )
```

Parameters

<i>price</i>	the price value to adjust
--------------	---------------------------

Example

```
' Print the real price for debugging
PRINT instrument.RealPrice( instrument.close )
```

If the data series had any negative values in it, the entire data series would be raised by the smallest factor or multiple of 10 possible. So if the most negative number was -.5 then the data series would be raised by 1.00.

If the data series was raised by 1.00, then the instrument.negativeAdjustment value would be 1.00, and the prices would be higher by 1.00 than the actual data file, and the prices in the chart.

So a close price of 54 would be represented as 55 in scripting, but charted as 54. To print the close price in real prices, use the instrument.RealPrice function. In this case the return value of instrument.RealPrice(55) would be 54, since the function subtracts the instrument.negativeAdjustment value from the input price.

RoundTick

Rounds the specified price to the nearest tick using the instrument's tick value. Behaves the same as `roundTickDown(price + minimumTick/2)`.

Syntax

```
roundedValue = RoundTick( price )
```

Parameters

price the price to be rounded

Example

```
' Round the price to the nearest tick.  
entryPrice = instrument.RoundTick( entryPrice )
```

RoundTickDown

Rounds the specified price rounded down to the nearest tick using the instrument's tick value.

Syntax

```
roundedValue = RoundTickDown( price )
```

Parameters

price the price to be rounded

Example

```
' Add one tick.  
entryPrice = entryPrice + instrument.minimumTick  
  
' Round the price up to the nearest tick.  
entryPrice = instrument.RoundTickDown( entryPrice )
```

RoundTickUp

Rounds the specified price rounded up to the nearest tick using the instrument's tick value.

Syntax

```
roundedValue = RoundTickUp( price )
```

Parameters

price the price to be rounded

Example

```
' Add one tick.  
entryPrice = entryPrice + instrument.minimumTick  
  
' Round the price up to the nearest tick.  
entryPrice = instrument.RoundTickUp( entryPrice )
```

7.3 Correlation Functions

The correlation functions allow you to control the correlations for instruments dynamically as a test runs.

Correlation Functions:	Descriptions:
ResetCloselyCorrelated	removes all instruments from the close correlation matrix for this instrument
ResetLooselyCorrelated	removes all instruments from the loose correlation matrix for this instrument
AddCloselyCorrelated	adds an instrument to the close correlation matrix for this instrument
AddLooselyCorrelated	adds and instrument to the loose correlation matrix for this instrument

Links:

[Correlation Properties](#)

See Also:

ResetCloselyCorrelated

Removes all the instruments from the close correlation matrix for this instrument. This is usually done as part of a dynamic correlation evaluation process where instruments are added based on their recent correlation using the [AddCloselyCorrelated](#) function.

Syntax

```
ResetCloselyCorrelated
```

Parameters

none

Example

```
' Remove all the close correlations.
instrument.ResetCloselyCorrelated
```

ResetLooselyCorrelated

Removes all the instruments from the loose correlation matrix for this instrument. This is usually done as part of a dynamic correlation evaluation process where instruments are added based on their recent correlation using the [AddLooselyCorrelated](#) function.

Syntax

```
ResetLooselyCorrelated
```


7.4 Correlation Properties

Correlation Property Names:	Description:
closelyCorrelated	String representation of all the closely correlated instruments. Symbols separated by colons.
looselyCorrelated	String representation of all the loosely correlated instruments. Symbols separated by colons.
closelyCorrelatedLongUnits	number of units long for all loosely correlated instruments
closelyCorrelatedShortUnits	number of units short for all closely correlated instruments
looselyCorrelatedLongUnits	number of units long for all loosely correlated instruments
looselyCorrelatedShortUnits	number of units short for all loosely correlated instruments
closelyCorrelatedLongInstruments	number of long closely correlated instruments
closelyCorrelatedShortInstruments	number of short closely correlated instruments
looselyCorrelatedLongInstruments	number of long loosely correlated instruments
looselyCorrelatedShortInstruments	number of short loosely correlated instruments
closelyCorrelatedLongQuantity	total quantity long for closely correlated instruments
closelyCorrelatedShortQuantity	total quantity short for closely correlated instruments
looselyCorrelatedLongQuantity	total quantity long for loosely correlated instruments
looselyCorrelatedShortQuantity	total quantity short for loosely correlated instruments
closelyCorrelatedLongRisk	total risk for closely correlated long instruments
closelyCorrelatedShortRisk	total risk for closely correlated short instruments
looselyCorrelatedLongRisk	total risk for loosely correlated long instruments
looselyCorrelatedShortRisk	total risk for loosely correlated short instruments
closelyCorrelatedLongMargin	total margin for closely correlated long instruments
closelyCorrelatedShortMargin	total margin for closely correlated short instruments

n	
looselyCorrelatedLongMargin	total margin for loosely correlated long instruments
looselyCorrelatedShortMargin	total margin for loosely correlated short instruments
closelyCorrelatedLongProfit	total profit for closely correlated long instruments
closelyCorrelatedShortProfit	total profit for closely correlated short instruments
looselyCorrelatedLongProfit	total profit for loosely correlated long instruments
looselyCorrelatedShortProfit	total profit for loosely correlated short instruments

Note:

These do not include zero sized trades.

Links:

[Correlation Functions](#)

See Also:

7.5 Group Properties

Group Index Categories:

1 = group1/industry
2 = group2/sector
3 = country
4 = currency

Dictionary Group Categories:

Group Names:	Descriptions:
group1	name of the group1. Set in the Futures or Stock Dictionary
group2	name of the group2.
industry	same as group1. Used for stocks.
sector	same as group2. Used for stocks.
country	name of the country. Set in the Stock Dictionary.

Instrument Group Properties:

Property Name:	Descriptions:
groupLongInstruments[]	the number of long instruments in the group
groupShortInstruments[]	the number of short instruments in the group
groupLongQuantity[]	the quantity long for instruments in the group
groupShortQuantity[]	the quantity short for instruments in the group
groupLongUnits[]	the number of units long for instruments in the group
groupShortUnits[]	the number of units short for instruments in the group
groupLongRisk[]	the total risk for long instruments in the group
groupShortRisk[]	the total risk for short instruments in the group
groupLongMargin[]	the total margin for long instruments in the group
groupShortMargin[]	the total margin for short instruments in the group
groupLongProfit[]	the total profit for long instruments in the group
groupShortProfit[]	the total profit for short instruments in the group

Note:

These do not include zero sized trades.



7.6 Historical Trade Properties

Trade Indexing:

Properties listed with a '[' following them may be indexed using a number which determines the trade starting with the last trade and working backward in time. If no index is supplied the property will return the information for the last trade.

```
lastTradeProfit = instrument.tradeProfit[ 1 ]
```

OR

```
lastTradeProfit = instrument.tradeProfit
```

will access the last trade's profit for this instrument.

Historical Trade Properties:	Descriptions:
tradeBarsInTrade[]	the bars in the trade
tradeCommission[]	
tradeCount	the number of prior trades including zero size trades. Used to index the following properties:
tradeCustomValue[]	the custom value as set through scripting
tradeDaysInTrade[]	the number days between entry and exit
tradeDirection[]	the direction (LONG or SHORT)
tradeDollarsPerPoint[]	the entry day dollars per point
tradeEntryBPV[]	the entry bpv of the instrument
tradeEntryDate[]	the entry date
tradeEntryFill[]	the entry fill price
tradeEntryOrder[]	the entry order price
tradeEntryRisk[]	the entry risk as a percent of entry day trading equity
tradeEntryStop[]	the protective stop on the day of entry
tradeEntryTime[]	the entry time
tradeExitDate[]	the exit date
tradeExitFill[]	the exit fill price
tradeExitOrder[]	the exit order price
tradeExitTime[]	the exit time
tradeMaxAdverseExcursion[]	the maximum adverse excursion of the trade in points
tradeMaxFavorableExcursion[]	the maximum favorable excursion of the trade in points
tradeMinFavorableExcursion[]	the minimum favorable excursion
tradePositionReferenceID[]	Each unit in a position is given a the unique unit reference ID that is passed on to this historical trade property. This ID number is assigned to the position when it is enabled, and is then available

	for each closed trade unit using the unit index with this property. See instrument.unitPositionReferenceID[] and order.referenceID .
tradeProfit[]	the closed out profit including slippage and commission
tradeProfitPercent[]	the profit as a percent of entry day trading equity
tradeQuantity[]	the quantity in shares or contracts
tradeRuleLabel[]	the rule label string as set in the unit, or the order that created the unit.
tradeUnitNumber[]	the unit number

7.7 Instrument Loading

The following functions can be used with a Block Permanent Variable of type Instrument.

Instrument Loading Functions:	Descriptions:
LoadSymbol	Loads into an instrument variable the data from an instrument. The instrument can be in the current portfolio, or not, and can be accessed by either symbol, type:symbol, or index. To be used only with a BPV variable of type Instrument; not to be used with the built in "instrument" object.
LoadByLongRank	Loads into an instrument variable the instrument located at the specified long rank position.
LoadByShortRank	Loads into an instrument variable the instrument located at the specified short rank position.
LoadExternalData	Loads into an instrument variable the data from an external file. The data loads into IPV Series variables as defined.
LoadIPVFromFile	Loads data into an IPV exactly as in the file. Does not fill holes in the data.

LoadSymbol

Sets an instrument variable to a particular instrument. The instrument can be in the portfolio or not, and the instrument can be set using the symbol, the type:symbol, or the index.

Recommended that all instruments to be loaded are loaded once in the Before Test script, so that these instruments are set to the correct date when they are used. Subsequent calls to this function in the test will not reload the instrument, but just set the variable accordingly.

Syntax

```
LoadSymbol( symbolSpecifier [ or symbol ] [ or index ], [system index] )
```

Parameters

symbolSpecifier	the symbol for the instrument with an optional market type prefix like "F:GC" or "S:IBM"
	Valid Prefixes: 'F:' - Futures 'S:' - Stocks 'FX:' - Forex
symbol	the symbol of the instrument like "S" or "AUDCAD"
index	the index in the current portfolio
systemIndex	the optional system index for the instrument to load.
returns	TRUE if the load was successful

Examples

Create a Block Permanent Instrument Object Variable called tempInstrument.

```
VARIABLES: instrumentCount TYPE: Integer

' Get the instrument count.
instrumentCount = system.totalInstruments

' Loop printing the symbol for each instrument.
FOR index = 1 TO instrumentCount STEP 1

    ' Set the portfolio instrument.
    tempInstrument.LoadSymbol( index )

    ' Print out the file name.
    PRINT "Portfolio contains: ", tempInstrument.symbol

NEXT
```

Create a [Block Permanent Variable](#) called crudeOil.

This example assumes that the "CL" symbol is of the same type as the portfolio being tested:

```
' Load the data for crude oil into the instrument
IF NOT crudeOil.LoadSymbol( "CL" ) THEN PRINT "Could not load CL"
```

While this example makes it explicit, that "CL" is of type Futures. It is defined in the Futures Dictionary, and the data is in the Futures Data Folder.

```
' Load the data for crude oil into the instrument
IF NOT crudeOil.LoadSymbol( "F:CL" ) THEN PRINT "Could not load CL"
```

Here we are loading a market index which could be used to validate market trends before putting on a position. The symbol of this instrument is "DJIA" and it is a stock.

```
IF NOT dowJoneIndustrials.LoadSymbol( "S:DJIA" ) THEN PRINT "Could not
load DJIA"
```

To check if a loaded instrument is part of the system's portfolio, check if the `instrument.priorityIndex > 0`. If it is greater than zero, then it is part of the portfolio.

for the implicit series to be created on the fly.

The header column names in the file itself are ignored.

Note that the variable "portfolioInstrument" is a BPV instrument variable, and is loaded with LoadSymbol prior to the use of this function. The default location for these files is the location of the data for the instrument. To use a full path, include the "C:\\" and any location can be used.

Syntax

```
loaded = LoadExternalData( fileName, dateColumn,
    [columnOne, columnTwo, ...] )
```

Parameters

fileName	the name of the file to open. If no path name is given, it defaults to the location of the instrument data file.
dateColumn	the name for the column specifying the date -- "date"
columnOne	the name for the first column of data after the date
columnTwo	the name for the second column of data after the date
returns	TRUE if the load was successful

Example

```
VARIABLES: instrumentCount TYPE: Integer
VARIABLES: externalFileName TYPE: String

' Get the instrument count.
instrumentCount = system.totalInstruments

' Loop initializing each instrument.
FOR index = 1 TO instrumentCount STEP 1

    ' Set the portfolio instrument. "portfolioInstrument" is defined
as a BPV Instrument variable.
    portfolioInstrument.LoadSymbol( index )

    ' Get the symbol for the instrument.
    externalFileName = portfolioInstrument.symbol +
        "_ExternalData.csv"

    ' Print out the file name.
    PRINT "Loading External File: ", externalFileName

    ' Load the external data.
    IF NOT portfolioInstrument.LoadExternalData( externalFileName,
        "date", "beta", "eps" ) THEN
        PRINT "Could not Load External Data for ", externalFileName
```

```
ENDIF
```

```
NEXT
```

This code loads external data files which use the symbol in the name and adds two new instrument properties: beta and eps. These new properties can be accessed in other scripts like:

```
IF instrument.beta > 1.2 THEN
```

or

```
IF instrument.eps > instrument.eps[90] THEN
```

File Format

The `LoadExternalData` call requires comma delimited text files with the first column being a date in the format `YYYYMMDD`.

A header is required, but ignored.

A data file, "CL_ExternalData.csv", which corresponds to the above `LoadExternalData` call might use quarterly data:

Date,	beta,	eps
20050115,	1.201,	5.8
20050415,	1.345,	6.2
20050715,	1.112,	5.3
20051015,	1.535,	6.9
20060115,	1.231,	8.4

When using Option 2, Trading Blox keeps only the required data in memory but lets you access the above properties as if there was data for each day in the instrument's data file. For example, on 20050413, the value for the beta property will be 1.201 and on 20050415 after the close the value will be 1.345. Note that even when using option 1, creating an IPV, the sparsely populated data will fill in any holes as needed so there is a value at every index. To retain the holes as a default value, use the [LoadIPVFromFile](#) function.

Property indexing uses the instrument's bar indexing so for daily bar data you will have access to the data on a daily basis updated as per the timeframes in the external data file. For example, using the above data on 20050415 after the close:

```
value = instrument.beta      ' returns 1.345
value = instrument.beta[1]  ' returns 1.201
value = instrument.beta[2]  ' returns 1.201
```

LoadIPVFromFile

Loads data from external text files and attaches it to particular instruments. Note that if using the instrument object it must have default context such as in the After Instrument Day script. Normally this function is used in the Before Simulation or Before Test script, so you need to create a BPV Instrument variable to use. It must be loaded with an instrument by using `LoadSymbol` before using this function call.

See also [LoadExternalData](#) function.

This function requires that all IPV's are created and defined. If this function is used in the Before Simulation script the data is loaded just once for the entire simulation. If placed in the Before Test

script, it will be loaded (refreshed) before each parameter test run. Note that if used in the Before Simulation script, to avoid overwriting the data with the default value, the IPV should be set as Simulation Scope.

The header column names in the file itself are ignored.

Note that the variable "portfolioInstrument" is a BPV instrument variable, and is loaded with LoadSymbol prior to the use of this function. The default location for these files is the location of the data for the instrument. To use a full path, include the "C:\\" and any location can be used.

No "Date" parameter is used in this function.

Syntax

```
loaded = LoadIPVFromFile( fileName, [columnOne, columnTwo, ...] )
```

Parameters

fileName	the name of the file to open. If no path name is given, it defaults to the location of the instrument data file.
columnOne	the name for the first column of data after the date and optional time
columnTwo	the name for the second column of data after the date and optional time
returns	TRUE if the load was successful

Example

```
VARIABLES: instrumentCount TYPE: Integer
VARIABLES: externalFileName TYPE: String

' Get the instrument count.
instrumentCount = system.totalInstruments

' Loop initializing each instrument.
FOR index = 1 TO instrumentCount STEP 1

    ' Set the portfolio instrument. "portfolioInstrument" is defined
as a BPV Instrument variable.
    portfolioInstrument.LoadSymbol( index )

    ' Get the symbol for the instrument.
    externalFileName = portfolioInstrument.symbol +
        "_ExternalData.csv"

    ' Print out the file name.
    PRINT "Loading External File: ", externalFileName

    ' Load the external data.
    IF NOT portfolioInstrument.LoadIPVFromFile( externalFileName,
        "beta", "eps" ) THEN
```

```

        PRINT "Could not Load External Data for ", externalFileName
    ENDIF
NEXT

```

This code loads external data files which use the symbol in the name and adds two new instrument properties: beta and eps. These new properties can be accessed in other scripts like:

```
IF instrument.beta > 1.2 THEN
```

or

```
IF instrument.eps > instrument.eps[90] THEN
```

File Format

The LoadExternalData call requires comma delimited text files with the first column being a date in the format YYYYMMDD.

A header is required, but ignored.

A data file, "CL_ExternalData.csv", which corresponds to the above LoadExternalData call might use quarterly data:

Date,	beta,	eps
20050115,	1.201,	5.8
20050415,	1.345,	6.2
20050715,	1.112,	5.3
20051015,	1.535,	6.9
20060115,	1.231,	8.4

This will load data into the dates supplied, and the remainder of the IPV will be at the default value.

Indexing works like a normal IPV depending on whether this IPV was setup for auto indexing or not. Auto indexing is recommended for this function. For example, using the above data on 20050715:

```

value = instrument.beta      ' returns 1.112
value = instrument.beta[1]  ' returns default value
value = instrument.beta[2]  ' returns default value

```

Note that this function will load date and time data into an IPV as well. The file format would then be Date, Time, column1, column2. This format will only work correctly if the instrument data is also intraday data. Note that all date time combos in the file must also be present in the instrument data file.

7.8 Position Functions

These functions are used to assign or change the value of position property values.

Positions Functions:	Descriptions:
SetUnitCustomValue	sets the custom value of the current position for the specified unit to the specified value
SetExitStop	sets the exit stop price for the specified unit
SetExitLimit	sets the exit limit price for the specified unit
order.SetRuleLabel	Sets the text description of the rule that created the order during the order process.

Links:

[Position Properties](#)

See Also:

SetUnitCustomValue

Sets the custom value field of the unit.

There must be a position on, so this cannot be used in the [Unit Size](#), [Can Add Unit](#), or [Can Fill Order](#) scripts unless there is already a unit on and the intention is to reference the current open position unit and not the current order. In these scripts the order is being processed so the order object should be used if that is the intention.

This function is best used in the [Entry Order Filled](#), [Exit](#), [Entry](#), or After Instrument Day scripts.

Syntax

```
SetUnitCustomValue( [unitNumber,] value )
```

Parameters

unitNumber	the unit number (optional). If not supplied this will default to the first unit
value	the custom value to be set. This value can be retrieved using the <code>unitCustomValue</code> property.

Example

```
' Set the custom value.
instrument.SetUnitCustomValue( 1.5 )

PRINT "The custom value is ", instrument.unitCustomValue
' Prints 1.5
```

SetExitStop

Sets the exit stop for the specified unit.

There must be a position on, so this cannot be used in the [Unit Size](#), [Can Add Unit](#), or [Can Fill Order](#) scripts. In these scripts the order is being processed so the order object should be used.

This function can be used in the [Entry Order Filled](#), [Exit](#), [Entry](#), Adjust Stops, or After Instrument Day scripts.

Syntax

```
SetExitStop( [unitNumber,] stopPrice )
```

Parameters

unitNumber	the unit number (optional). If not supplied this will default to the first unit
stopPrice	the value of the stop to be set.

Example for the Entry Order Filled script:

```
' Move the stop by the amount of the slippage.
instrument.SetExitStop( order.fillPrice - order.entryRisk )
```

```
' Set the stop price for the specified unit.
instrument.SetExitStop( unitNumber, newStopPrice )
```

NOTE: If you set the stop with this function, the daily risk will be calculated using this value, but **no order is placed**. To place an actual stop in the market use a broker order like this:
 broker.ExitAllUnitsOnStop(instrument.unitExitStop).

SetExitLimit

Sets the exit limit for the specified unit.

There must be a position on, so this cannot be used in the [Unit Size](#), [Can Add Unit](#), or [Can Fill Order](#) scripts. In these scripts the order is being processed so the order object should be used.

This function can be used in the [Entry Order Filled](#), [Exit](#), [Entry](#), Adjust Stops, or After Instrument Day scripts.

Syntax

```
SetExitLimit( [unitNumber,] limitPrice )
```

Parameters

unitNumber	the unit number (optional). If not supplied this will default to the first unit
limitPrice	the value of the limit to be set.

Example

```
' Set the limit price.
instrument.SetExitLimit( limitPrice )
```

```
' Set the limit price for the specified unit.
instrument.SetExitLimit( unitNumber, limitPrice )
```


NOTE: If you set the limit with this function **no order is placed**. To place an actual limit order in the market use a broker order like this: `broker.ExitAllUnitsAtLimit(instrument.unitExitLimit)`.

7.9 Position Properties

Unit Indexing

Properties listed with a '[' following them may be indexed using a number which determines the unit. If no index is supplied the property will return the information for the first unit. For example:

```
firstUnitQuantity = instrument.unitQuantity[ 1 ]
```

will access the quantity for the first unit for a position of this instrument.

Active Position Properties:	Descriptions:
position	the current position, represented by the numerical constants SHORT, OUT, or LONG.
positionDescription	the current position as a string (equal to "Long", "Short", or "Out"). Useful for printing.
currentPositionUnits	the total number of units on for the current position
currentPositionQuantity	the total number of contracts or shares on for the current position
currentPositionProfit	the total profit, in dollars, using the current close, of all units in the current position. This includes roll profit, slippage, and commission that has already moved to closed equity, as well as forex carry and stock dividends. Does not include future expected commission for the trade once it has closed.
currentPositionRisk	the total risk for this position, in dollars, based on the close and the stop prices
barsSinceExit	the number of bars since the last exit, regardless of unit

Unit Specific Properties:	Descriptions:
unitEntryDate[]	the entry date of the unit
unitEntryTime[]	the entry time of the unit
unitEntryDayIndex[]	the day index of the entry
unitEntryOrder[]	the order price for the unit
unitEntryFill[]	the fill price for the unit
unitQuantity[]	the quantity for the unit
unitExitStop[]	the exit stop for the unit as set by the original broker function call, or set by SetExitStop
unitNoExitStop[]	returns true if there is no exit stop set for the unit. Useful for cleaner reporting.
unitExitLimit[]	the exit limit for the unit as set by SetExitLimit
unitEntryRisk[]	the entry risk for the unit adjusted by the fill price

unitProfit[]	the open profit of the trade, net of commissions etc, in instrument currency.
unitRollProfit[]	the accrued closed profit of the futures trade for all rolls, in base currency.
unitRollCommission []	the accrued commission of the futures trade from all rolls, in base currency.
unitRollSlippage[]	the accrued slippage of the futures trade from all rolls, in base currency.
unitCommission[]	the computed commission that will be applied to the trade when the position exits.
unitCustomCommission []	the accrued custom commission added to the instrument through the use of the AddCommission function.
unitCarry[]	the total carry cost of the unit for forex trades.
unitBarsSinceEntry []	the number of bars since the entry of this unit
unitCustomValue[]	the custom value as set through scripting into the unit, or the order that created the unit. Float value.
unitSavedWFProfit[]	
unitPositionReferenceID[]	New orders are given a the unique reference ID. This ID number is assigned to the position when it is enabled. It is then available from this property. See Order Properties
unitRuleLabel[]	the rule label as set through scripting into the unit, or into the order that created the unit. String value.
unitMaxFavorableExcursion[]	the maximum favorable excursion of the unit
unitMaxAdverseExcursion[]	the maximum unfavorable excursion of the unit
unitMinFavorableExcursion[]	the minimum favorable excursion of the unit
unitDeliveryMonth[]	the delivery month (YYYYMM) of the unit

Links:

[Position Functions](#)

See Also:

7.10 Ranking Functions

The ranking functions are usually used in the [Portfolio Manager](#) to set ranking values and to filter the instrument from the portfolio.

Note that only instruments that are primed and ready to trade will be ranked. Other markets will be excluded from the ranking process.

Set the long ranking value and the short ranking value in the Rank Instruments script of the Portfolio Manager block. Then in the Filter Portfolio script the long rank and short rank will be available using the properties [instrument.longRank](#) and [instrument.shortRank](#). The long ranking value is sorted highest to lowest to determine the long rank. The short ranking value is sorted lowest to highest to determine the short rank.

In the case of equal ranking values, the instruments will be sorted alphabetically.

In the time between the execution of the Rank Instruments script and the Filter Portfolio script, the system sorts all the instruments based on the short and long ranking value.

Function Name:	Description:
SetLongRankingValue	sets the long ranking value. Sorted highest to lowest.
SetShortRankingValue	sets the short ranking value. Sorted lowest to highest.

The ranking values can also be set in others scripts, such as in a manual instrument loop in the Before Test Script.

- Set the Long and Short Ranking Values of each instrument
- Call the system.RankInstruments function
- Retrieve the Long Rank and Short Rank from each instrument

In addition, the custom sort value is available for use by the system. [SortInstrumentList](#)(method) function.

Property:

```
instrument.customSortValue
```

Function:

```
instrument.SetCustomSortValue( value )
```

The Custom Sort Value can also be set in other scripts, just like the ranking Values.

- Loop over each instrument setting the custom sort value
- call system.sortInstrumentList(4)
- Loop over each instrument and note how the instrument list has been sorted

Once the instrument list is sorted using the SortInstrumentList function, the Entry Orders script and other instrument scripts will execute in a new order.

Links:[Ranking Properties](#)**See Also:****SetLongRankingValue**

Sets the long ranking value. This function is generally used by a [Portfolio Manager](#) block as part of the instrument ranking process which is one way to select instruments for trading.

Syntax

```
instrument.SetLongRankingValue( rankingValue )
```

Parameters

rankingValue	the value to be used for ranking this instrument for long trades
--------------	--

Example code for the Rank Instruments script:

```
' Set the long ranking value for this instrument
instrument.SetLongRankingValue( rsi )
```

Example:

If you have three instruments in your portfolio A, B, and C.

In the Rank Instruments script you use the SetLongRankingValue function as follows:

- For instrument A you set the long ranking value to 34.
- For instrument B you set the long ranking value to 53
- For instrument C you set the long ranking value to -12.

These will be sorted from highest to lowest.

Now in the Filter Portfolio script you can access the longRank property.

- Instrument A will have a long rank of 2.
- Instrument B will have a long rank of 1.
- Instrument C will have a long rank of 3.

SetShortRankingValue

Sets the short ranking value. This function is generally used by a [Portfolio Manager](#) block as part of the instrument ranking process which is one way to select instruments for trading.

Syntax

```
instrument.SetShortRankingValue( rankingValue )
```

Parameters

<code>rankingValue</code>	the value to be used for ranking this instrument for short trades
---------------------------	---

Example code for the Rank Instruments script:

```
' Set the short ranking value.  
instrument.SetShortRankingValue( rsi )
```

Example:

If you have three instruments in your portfolio A, B, and C.

In the Rank Instruments script you use the `SetShortRankingValue` function as follows:

For instrument A you set the short ranking value to 34.

For instrument B you set the short ranking value to 53

For instrument C you set the short ranking value to -12.

These will be sorted from highest to lowest.

Now in the Filter Portfolio script you can access the `shortRank` property.

Instrument A will have a short rank of 2.

Instrument B will have a short rank of 3.

Instrument C will have a short rank of 1.

7.11 Ranking Properties

The ranking properties can be accessed from anywhere in the system. So if you set the rank in the portfolio manager, you can access that rank in the entry script.

The longRank, shortRank, longGroupRank, and shortGroupRank properties are based off the longRankingValue and the ShortRanking Value. The way these get calculated is when the RankingValue is set in the Rank Instruments script, the instruments are then sorted, and then the Rank properties are available in the Filter Porfolio script. If the RankingValue is set anywhere else in the system, the new Rank will not be available until the Portfolio Manager runs again for the next day.

Function Name:	Description:
longRankingValue	the value used to rank the instrument for long trades
shortRankingValue	the value used to rank the instrument for short trades
longRank	the rank when sorted by long ranking value
shortRank	the rank when sorted by short ranking value
longGroupRank	the rank of the instrument within its group when sorted by long ranking value
shortGroupRank	the rank of the instrument within its group when sorted by short ranking value

Links:

[Ranking Functions](#)

See Also:

7.12 Trade Control Properties

Function Name:	Description:
canTradeLong	true if the instrument is allowed to trade long today
canTradeShort	true if the instrument is allowed to trade short today

Note:
These can be used throughout the system. They are set by the Trade Control Functions in the Portfolio Manager. Once set for the instrument, these properties are available for access anywhere in the system.

Links:
[Trade Control Functions](#)

See Also:

7.13 Trade Control Functions

Usually used in a portfolio manager to allow or deny trades for the day based on some criteria.

Instruments allow all trades by default at the beginning of the test.

This value is not reset by the system day to day, so whatever is set here sticks until the next time it is updated.

Trade Control Functions:	Descriptions:
AllowLongTrades	marks instrument to allow long trades
AllowShortTrades	marks instrument to allow short trades
AllowAllTrades	marks instrument to allow all trades
DenyLongTrades	marks instrument to deny long trades
DenyShortTrades	marks instrument to deny short trades
DenyAllTrades	marks instrument to deny all trades

If the [AllowAllTrades](#) function is used for an instrument, all trades both long and short will be processed.

If the [DenyAllTrades](#) function is used for an instrument, all trades both long and short will be rejected.

The following functions can be used independently. They affect only one direction, not both.

If the [AllowLongTrades](#) function is used for an instrument, long trades will be processed.

If the [AllowShortTrades](#) function is used for an instrument, short trades will be processed.

If the [DenyLongTrades](#) function is used for an instrument, long trades will be rejected.

If the [DenyShortTrades](#) function is used for an instrument, short trades will be rejected.

Note:

Use `instrument.DenyAllTrades` at the top of the Portfolio Manager block Filter Portfolio script to deny all trades by default. In this way, the code can allow trades for instruments that meet certain criteria, and the rest will be denied.

Links:

[Trade Control Properties](#)

See Also:

AllowLongTrades

Marks the instrument to allow long trades.

Syntax:

```
instrument.AllowLongTrades
```

Parameter:

none

Description:

Function works without any user assigned values.

Results:**Example:**

```
' If this instrument is in the top rankings...  
IF instrument.longRank <= rankThreshold THEN  
  
    instrument.AllowLongTrades  
ENDIF
```

Links:**See Also:**

AllowShortTrades

Marks the instrument to allow short trades.

Syntax:

```
instrument.AllowShortTrades
```

Parameter:

none

Description:

Function works without any user assigned values.

Results:**Example:**

```
' If this instrument is in the top rankings...  
IF instrument.shortRank <= rankThreshold THEN  
  
    instrument.AllowShortTrades  
ENDIF
```

Links:**See Also:**

AllowAllTrades

Marks the instrument to allow both long and short trades.

Syntax:

```
instrument.AllowAllTrades
```

Parameter:

none

Description:

Function works without any user assigned values.

Results:**Example:**

```
' Allow all trades unless we filter below based on  
' further criteria.  
instrument.AllowAllTrades
```

Links:**See Also:**

DenyLongTrades

Marks the instrument to deny long trades. This function is the opposite of [AllowLongTrades](#).

Syntax:

```
instrument.DenyLongTrades
```

Parameter:**Description:**

Function works without any user assigned values.

Results:**Example:**

```
' If this instrument is NOT in the top rankings...  
IF instrument.longRank > rankThreshold THEN  
  
    instrument.DenyLongTrades  
ENDIF
```

Links:**See Also:**

DenyShortTrades

Marks the instrument to deny short trades. This function is the opposite of [AllowShortTrades](#).

Syntax:

```
instrument.DenyShortTrades
```

Parameter:

none

Description:

Function works without any user assigned values.

Results:**Example:**

```
' If this instrument is NOT in the top rankings...  
IF instrument.shortRank > rankThreshold THEN  
  
    instrument.DenyShortTrades  
ENDIF
```

Links:**See Also:**

DenyAllTrades

Marks the instrument to deny both long and short trades. This function is the opposite of [AllowAllTrades](#).

Syntax:

```
instrument.DenyAllTrades
```

Parameter:

none

Description:

Function works without any user assigned values.

Results:**Example:**

```
' Deny all trades unless we allow below based on further criteria.  
instrument.DenyAllTrades
```

Links:**See Also:**

Section 8 – Order

Order object properties and function are available in the [Unit Size](#), [Can Add Unit](#), and [Can Fill Order](#) script section where they have object context by default. They are also in context in the [Entry Orders](#), and [Exit Orders](#) script section after a [Broker](#) function statement when the [Broker](#) initiated order has not been rejected.

An order is available when the [system.orderExists\(\)](#) property contains a [TRUE](#) value after a [Broker](#) function statement returns execution to the [Entry Orders](#) or [Exit Orders](#) script section.

Orders are also in context in the [Entry Orders Filled](#) and [Exit Orders Filled](#) script section when those orders have been filled.

In all other script sections where an order does not have context, access to an order property or order function is possible when the [AlternateOrder](#) object process is used to bring the order object into context so that the order's property information can be accessed or changed. Do not use the Order object in any of the script sections where it does not have a default context.

Creating Orders:

All signals require an order to generate an instrument position. Each order contains information about how the user intended the order to be executed, and in some cases protected. All orders are generated by a [Broker](#) Object function. These functions decide if the order is for entry or exit. Entry orders create positions when an instrument's prices satisfies the order's conditions to enable a trade. Exit orders will terminate, or exit a position when the conditions given to the order to exit are enabled in the market, or when the adjusted quantity remaining in a position is reduced to zero.

Once an Entry order is initiated by a Broker function order details are sent to the [Unit Size](#) script where the logic in that script can assign a quantity to the order. All Trading Blox supplied money manager modules will, by design, reject orders that are sized with a quantity less than the [instrument.roundLot](#) property value. Orders with zero quantity can be used to create positions, however without a quantity of at least at or above the [instrument.roundLot](#) value the math that is applied to the price change and trade expenses will be rounded to zero. In order to generate zero quantity orders it will be necessary to modify the logic changes in the Unit Sizing script so a zero quantity order is not rejected.

Entry orders not rejected are then sent to the [Can Add Unit](#) script section where other rules and conditions can be applied to allow or reject the order. Rejected orders will no longer exist after the [Can Add Unit](#) script terminates execution. Orders that are not rejected will be accessible in the script where the Broker function created the order. Orders rejected because an instrument's [canTradeLong](#) or [canTradeShort](#) properties are set to False will create a rejection record that will appear in the Filtered Log report. Rejected orders by either of these two properties will not be accessible in the [Unit Size](#) or [Can Add Unit](#) script sections.

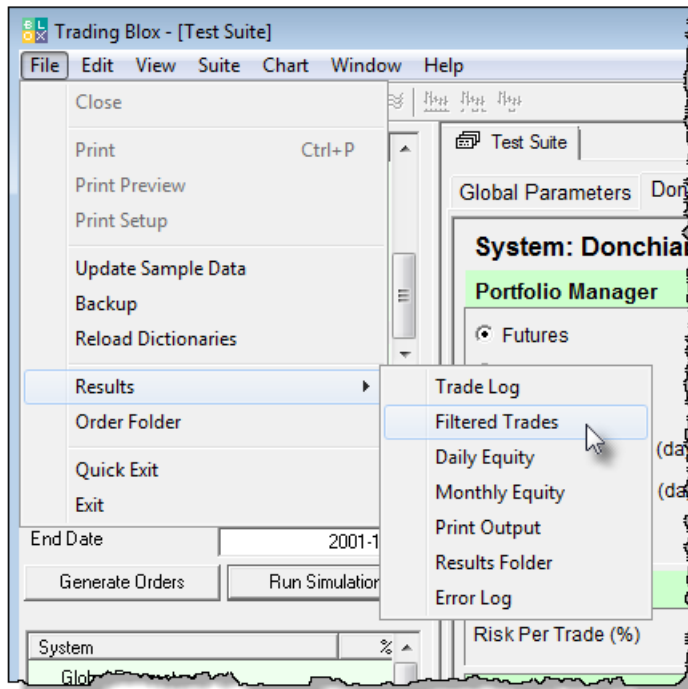
Exit orders can only be created when an instrument has an active position. Exit orders do not get processed by the [Unit Size](#) or the [Can Add Unit](#) scripts.

All Entry and Exit orders that are not rejected before being tested on the next instrument's date are processed through the [Can Fill Order](#) script section. In addition, all Entry orders are processed by the [Entry Order Filled](#) script section, and all Exit orders are processed by the [Exit Order Filled](#) script section.

Each order generated and enabled by the market is applied to either create a position, or is to be added to an existing existing position as an additional unit with be assigned its own unit identifying number. If only one signal is applied, then only one unit is used as that instrument's position

information. Order for creating a new unit will show a [unitNumber](#) as zero. Orders don't create units, but instead executed orders create units. When an exit order is created, the [unitNumber](#) property will show a value of at least one for the first order.

All rejected orders are listed in the Trading Blox Filter Log file available under the Main menu's File -> Results -> Filtered Trades selection:



All the information required to generate an order is contained in the Order Object's properties. Information that can be changed before the order is executed, or rejected, can be handled by using one of the Order Object's functions.

Order Object:	Description:
Order Properties	When information is needed from an order use the properties listed in the Order Object properties table.
Order Functions	When information in an order needs to be changed, and that order has not been executed in the market, and it has not been rejected prior to being executed, use one of the functions listed in the Order Object function table. Executed and rejected order disappear from the system and are not accessible.

All of the Order Object's properties using the Order prefix with a property or a function are only accessible in the following script sections.

Script Section:	Description:
Entry Orders	Signal orders only exists in this script section after a Broker Object function has been executed, and only when the order processing through the Unit Size and Can Add Unit script sections have not rejected the order. To know if the order still exists, use the <code>system.orderExists</code> property.

Entry Order Filled	When an Entry Order has been successful filled, it arrives in this script section.
Exit Orders	Signal orders to affect an existing position for the symbol in context of this script section are created only after a Broker Object function has been executed.
Exit Order Filled	Exit order execution to close a position, or change a position's size, this script is called with the results of order's fill information.
Can Add Unit	Only entry orders pass through this script so they can be adjusted, or rejected, for reasons other than risk or equity values.
Can Fill Order	After all the available orders are executed on new data, this script is called for each instrument so custom requirements can be applied to the order's results.
Unit Size	This is where all new Entry orders are sent once a Broker Object function has been executed. This script handles the chores associate with specifying the order's unit quantity. Orders leaving this script with its <code>order.continueProcessing</code> flag still set to True are passed along to the Can Add Unit script section, if that script section exist in the system with scripted code.

Notes:

Order object information that is available can be accessed in any of the above order scripts using the Order Object "`order`" prefix. An example is an Entry Orders script after a successful [Broker](#) function call finishes. An example on how to know if the order was created successfully is shown in this next code example that is placed after the completing [Broker](#) function:

Example:

```
' Check if the Broker Object created an Order,...
If system.orderExists() THEN
  ' Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel )

  ' Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel )
ENDIF ' s.orderExists
```

This code snippet example uses the [System Object](#) property `orderExists()` to discover if the order was successful or rejected before using the Order Object's functions to update the Positions & Orders Report and Trade Log record with the signal rule information that created the order. Testing to be sure the order exist is critical for preventing errors. To attempt to access an order that doesn't exist will cause Trading Blox to generate an error because the script is trying to assign information to an object that doesn't exist.

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the `system.orderExists` function prior to accessing it.

Using the `alternateOrder` object when orders are not in context will provide the same access as the Order object. However, the `alternateOrder` object must be brought into context using

the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

8.1 OrderProperties

The **order** object (and the **alternateOrder** object) have the following properties. These properties can only be access in scripts that have a default order object context, or if you have set the order object using the **SetOrder** system function.

Properties:	Description:
blockName	Name of the order's originating block.
clearingIntent	Value used by IB for clearing intent. Defaults to "IB"
continueProcessing	This is always TRUE so that it will continue processing. It is False when the order has been rejected.
customValue	By default it is always blank unless scripting assigned a numeric value with the order function the SetCustomValue . Once a value is assigned, this property will returns the custom value. It will also pass the value along to the instrument.unitCustomValue property so that it can be used in the system, and discovered in the system's TradeLog .
entryRisk	the entry risk of the order, this is the difference between the order price and the stop price. This is not adjusted by the fill price.
executionType	the execution type as a string: "at Market", "on Stop", "on Open", "on Close", "on Stop Close", "at Limit Close", "on Stop Open", "Limit", "on Limit Open"
fillPrice	the calculated fill price of the order. This value is not used automatically by Trading Blox.
isBuy	returns TRUE if the order is a buy order
isEntry	returns TRUE if the order is an entry
limitPrice	the profit taking limit price of the order
noStopPrice	returns true if the order has no stop price set
orderPrice	the price of the order for stop or limit orders, the close price for OnOpen orders and OnClose orders
orderReportMessage	returns the order report message as set by SetOrderReportMessage
orderType	the order type as a string: "Long Entry", "Short Entry", "Long Exit", "Short Exit"
position	the position as an integer: LONG or SHORT. Returns an integer, with 1 being long and -1 being short. The constants LONG and SHORT can be used for comparison purposes. This is not a string property.
processingMessage	returns the reject message if rejected.
quantity	the quantity of the order
referenceID	the unique reference ID for the order
ruleLabel	returns the rule label as set by SetRuleLabel
sortValue	returns the sort value as set by SetSortValue
stopPrice	the protect stop price of the order. This value is only used automatically on the order entry day, and only if the Entry Day Retracement is greater than zero.

symbol	the symbol of the order
systemBlockName	the system and block name of the originating block and system
timeInForce	the value used by IB for time in force. Defaults to "GTC"
unitNumber	the unit number of the order

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and function are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#)

See Also:

blockName

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p> <p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p> <p>See Also:</p>	

clearingIntent

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

continueProcessing

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p> <p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p> <p>See Also:</p>	

customValue

TYPE:	Description:
customValue	Any numeric value assigned using the order.SetCustomValue function.
<p>Example:</p> <pre> ' ~~~~~ ' Create a custom numeric value value = 3 x 4 ' Assign that custom numeric value to the ' order's customValue property. order.SetCustomValue(value) ' Display user's custom value PRINT "order.customValue = ", 12 ' ~~~~~ </pre> <p>Returns:</p> <pre>order.customValue = 12</pre>	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes:</p> <p>Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links:</p> <p>AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties, s</p>	
<p>See Also:</p>	

entryRisk

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

executionType

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p> <p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p> <p>See Also:</p>	

fillPrice

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p> <p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p> <p>See Also:</p>	

isBuy

TYPE:	Description:
Example:	
Returns:	
Alternate Order Object: Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.	
Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it. Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions. Once orders are brought into context their properties and function are available to reference and changes.	
Links: AlternateOrder Object , AlternateSystem Object , Order Functions , Order Object , Order Properties	
See Also:	

isEntry

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

limitPrice

TYPE:	Description:
Example:	
Returns:	
Alternate Order Object: Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.	
Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it. Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions. Once orders are brought into context their properties and function are available to reference and changes.	
Links: AlternateOrder Object , AlternateSystem Object , Order Functions , Order Object , Order Properties	
See Also:	

noStopPrice

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

orderPrice

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

orderReportMessage

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

orderType

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

position

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p> <p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p> <p>See Also:</p>	

quantity

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

referenceID

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p> <p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p> <p>See Also:</p>	

ruleLabel

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

sortValue

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

stopPrice

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

symbol

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p> <p>See Also:</p>	

systemBlockName

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p>	
<p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p>	
<p>See Also:</p>	

timeInForce

TYPE:	Description:
Example:	
Returns:	
<p><u>Alternate Order Object:</u> Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.</p> <p>Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it.</p> <p>Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>	
<p>Links: AlternateOrder Object, AlternateSystem Object, Order Functions, Order Object, Order Properties</p> <p>See Also:</p>	

unitNumber

TYPE:	Description:
Example:	
Returns:	
Alternate Order Object: Access to Order Object properties and functions in other scripts is made possible by using the AlternateOrder Object as the prefix ahead of the "." property or function. AlternateOrder Object is discussed below.	
Notes: Always check to be sure the order is available after a Broker function call using the system.orderExists function prior to accessing it. Using the alternateOrder object when orders are not in context will provide the same access as the Order object. However, the alternateOrder object must be brought into context using the system.SetAlternateOrder object function prior to any attempt to use its properties and functions. Once orders are brought into context their properties and function are available to reference and changes.	
Links: AlternateOrder Object , AlternateSystem Object , Order Functions , Order Object , Order Properties	
See Also:	

8.2 OrderFunctions

The Order functions are used to modify an existing order. You can set the fill price, quantity, stop price, or reject the order all together. It is common in the Unit Size script to use the SetQuantity function to set the quantity of the order.

Note that these can only be used in certain scripts that have default order object context. The scripts in which they can be used are listed for each function.

Order Function:	Description:
SetFillPrice	sets a new fill price for the order
SetQuantity	sets a new quantity for the order
SetStopPrice	sets a new protect stop for the order. This value is only used automatically on the order entry day, and only if the Entry Day Retracement is greater than zero.
SetLimitPrice	sets a new profit taking limit for the order. This value is not used automatically by Trading Blox.
Reject	rejects the order to stop further processing, and sets the reject message that is printed in the Filtered Trade Log
SetRuleLabel	sets the rule label string
SetCustomValue	sets the custom value number
SetOrderReportMessage	sets the order report message string
SetSortValue	sets the sort value number of the order, for use in sorting the orders
SetTimeInForce	sets the time in force property for IB orders. The default is "GTC"
SetClearingIntent	sets the clearing intent property for IB orders. The default is set by the system.SetClearingIntent, and is "IB".

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Object](#), [Order Properties](#)

See Also:

Reject

Rejects the order and prevents further processing.

This function **can only** be used in the [Unit Size](#), [Can Add Unit](#), and [Can Fill Order](#) scripts.

Syntax:

```
order.Reject( message )
```

Parameter:

message

Description:

Script reason why the order was rejection, or filtered from available orders.

Returns:

Generates a message in the Filtered Log report when software preferences have the log enabled.

Example:

```
' ~~~~~
' Common Fixed Fractional Order sizing calculation.
' ~~~~~
' Calculate amount of equity available for order sizing.
riskEquity = system.tradingEquity * riskPerTrade

' Convert instrument point risk into dollars.
dollarRisk = order.entryRisk * instrument.bigPointValue

' Order quantity will be the integer portion division.
tradeQuantity = riskEquity / dollarRisk

' If tradeQuantity is less than 1,...
If tradeQuantity < 1 THEN
  ' Order quantities less than 1 are rejected
  order.Reject( "Order Quantity less than 1." )
ELSE
  ' Order greater than 1 become order size amount.
  order.SetQuantity( tradeQuantity )
ENDIF ' tradeQuantity < 1
' ~~~~~
```

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and functions are available to reference and changes.

Links:

[bigPointValue](#), [entryRisk](#), [SetQuantity](#), [tradingEquity](#)

See Also:

[Can Add Unit](#), [Can Fill Order](#), [Unit Size](#), [AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetClearingIntent

Syntax:

Parameter:	Description:

Returns:

Example:

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and function are available to reference and changes.

Links:

See Also:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetCustomValue

Sets a user assigned custom numeric value to an order so that the value will be available when the position is active and when it is reported in the Trade Log.

Syntax:

```
order.SetCustomValue( value )
```

Parameter:

value

Description:

Property will accept decimal, or Integer values.

Returns:

Assigned a numeric value that will appear in the `order.customValue`, and `instrument.unitCustomValue` properties, after the trade has ended in the Trade Log report.

Example:

```

' ~~~~~
' Create a custom numeric value
value = 3 x 4
' Assign that custom numeric value to the
' order's customValue property.
order.SetCustomValue( value )

' Display user's custom value
PRINT "order.customValue = ", 12
' ~~~~~

```

Results:

```
order.customValue = 12
```

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and function are available to reference and changes.

Links:

[customValue](#), [unitCustomValue](#)

See Also:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetFillPrice

Sets the fill price for an order to the specified price.

This function is only available in a [Can Fill Order](#) script section and it is used to set a fill price to something other than the software's built-in fill algorithm's price.

Syntax:

```
order.SetFillPrice( fillPrice )
```

Parameter:

fillPrice

Description:

Price at which the order is filled

Returns:

[order.fillPrice](#) assigned in the [Can Fill Order](#) script section, or the price assigned by the sof

Example:

```
' Set the fill to the high of the day since this order
' was more than 10% of the market volume.
Order.SetFillPrice( Instrument.High )
```

[Alternate Order Object:](#)

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and function are available to reference and changes.

Links:

[fillPrice](#)

See Also:

[Can Fill Order](#), [AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetLimitPrice

Syntax:

Parameter:	Description:

Returns:

Example:

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and function are available to reference and changes.

Links:

See Also:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetOrderReportMessage

Sets the Order Report Message for the order. This message will show up on the order report, for this order.

Syntax:

```
order.SetOrderReportMessage( message )
```

Parameter:

message

Description:

Text description of the rule that created the order.

Returns:

Rule Labels assigned will appear in the new order section of the Position and Order Report.

Example:

```
' ~~~~~
' LONG EXIT ORDERS
If instrument.position = LONG THEN
  ' Protective Exit Price
  LongEx = instrument.RoundTick(Max(UpTrend, SellLine))

  ' Update Risk Basis Property
  instrument.SetExitStop(LongEx)

  ' Update Protective Exit Indicator
  StopPrice = LongEx

  ' Assemble Order's Rule Details
  sRuleLabel = "Lx@" + instrument.PriceFormat(StopPrice - PriceAdj) + "s,"

  ' Send Order to Market
  broker.ExitAllUnitsOnStop(LongEx)

' ~~~~~
' When Broker Order Exist,...
If system.OrderExists() THEN
  ' Apply Order Detail To Trade Information
  order.SetRuleLabel( sRuleLabel )

  ' Apply Order Details To Order Information
  order.SetOrderReportMessage( sRuleLabel )
ENDIF ' OrderExists
' ~~~~~
ENDIF ' i.Position = LONG

' ~~~~~
' Printing order.ruleLabel after assignment:
Print "order.ruleLabel = ", order.ruleLabel
```

Results:

```
order.ruleLabel = Lx@100.35s
```

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and function are available to reference and changes.

Links:

[Max](#), [OrderExists](#), [position](#), [PriceFormat](#), [RoundTick](#), [ruleLabel](#),
[unitRuleLabel](#)
[SetExitStop](#), [SetOrderReportMessage](#)

See Also:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetQuantity

Sets the quantity for an order to the specified amount. This function is only available to the [Unit Size](#), [Can Add Unit](#), and [Can Fill Order Script](#) script and is used to set the order quantity.

NOTE:

This function is only valid for Entry Orders and is ignored for Exit Orders.

Syntax:

```
order.SetQuantity( quantity )
```

Parameter:

quantity

Description:

Quantity to assign to the order.

Returns:

Integer quantity assigned.

Example:

```
' ~~~~~
' Common Fixed Fractional Order sizing calculation.
' ~~~~~
' Calculate amount of equity available for order sizing.
riskEquity = system.tradingEquity * riskPerTrade

' Convert instrument point risk into dollars.
dollarRisk = order.entryRisk * instrument.bigPointValue

' Order quantity will be the integer portion division.
tradeQuantity = riskEquity / dollarRisk

' If tradeQuantity is less than 1,...
If tradeQuantity < 1 THEN
  ' Order quantities less than 1 are rejected
  order.Reject( "Order Quantity less than 1." )
ELSE
  ' Order greater than 1 become order size amount.
  order.SetQuantity( tradeQuantity )
ENDIF ' tradeQuantity < 1
' ~~~~~
```

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and functions are available to reference and changes.

Links:

[bigPointValue](#), [entryRisk](#), [Reject](#), [tradingEquity](#)

See Also:

[Can Add Unit](#), [Can Fill Order](#), [Data Group and Types](#), [Unit Size](#), [AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetRuleLabel

Function assigns a text value that communicates the why this order was created.

Syntax:

```
order.SetRuleLabel( ruleLabel )
```

Parameter:

ruleLabel

Description:

Text description of the rule that created the order.

Returns:

Rule Labels assigned will appear in the Trade Log report.

Example:

```
' ~~~~~
' LONG EXIT ORDERS
If instrument.position = LONG THEN
'   Protective Exit Price
LongEx = instrument.RoundTick(Max(UpTrend, SellLine))

'   Update Risk Basis Property
instrument.SetExitStop(LongEx)

'   Update Protective Exit Indicator
StopPrice = LongEx

'   Assemble Order's Rule Details
sRuleLabel = "Lx@" + instrument.PriceFormat(StopPrice - PriceAdj) + "s

'   Send Order to Market
broker.ExitAllUnitsOnStop(LongEx)

' ~~~~~
' When Broker Order Exist,...
If system.OrderExists() THEN
'   Apply Order Detail To Trade Information
order.SetRuleLabel( sRuleLabel )

'   Apply Order Details To Order Information
order.SetOrderReportMessage( sRuleLabel )
ENDIF ' OrderExists
' ~~~~~
ENDIF ' i.Position = LONG

' ~~~~~
' Printing order.ruleLabel after assignment:
Print "order.ruleLabel = ", order.ruleLabel
```

Results:

```
order.ruleLabel = Lx@100.35s
```

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and function are available to reference and changes.

Links:

[Max](#), [OrderExists](#), [position](#), [PriceFormat](#), [RoundTick](#), [SetExitStop](#),
[SetOrderReportMessage](#)

See Also:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetSortValue

Sets the Sort Value for the order that will assigned to the `order.sortValue` property.

Loop over the orders setting this value, and then use the [system.SortOrdersBySortValue](#) to sort the orders.

Syntax:

```
order.SetSortValue( sortValue )
```

TYPE:

sortValue

Description:

Assign a numeric sort value. Number is any user value based upon how the user wants the orders sorted. Orders are sorted in an ascending lowest to highest value sequence.

Returns:

Places the assigned sortValue in the order's `sortValue` property.

Example:

```

' ~~~~~
' Before order Execution
' ~~~~~
' Create Column Header Titles
PRINT "#", "ao.referenceID", "ao.symbol", "ao.SortValue"
' ~~~~~
' Get Total of Open Orders
totalOpenOrders = system.totalOpenOrders

' Show Order Sequence Before Orders are Sorted
PRINT "Before Orders are Sorted"
' Loop through the open orders & Assign Ranking
FOR x = 1 TO totalOpenOrders STEP 1
' Access each open order
system.SetAlternateOrder( x )

' Generate a Random value for each order
PRINT x, alternateOrder.referenceID, alternateOrder.symbol, alternateOrder.SortValue
Next ' x
' ~~~~~

' Loop through the open orders & Assign a Random Ranking value
FOR x = 1 TO totalOpenOrders STEP 1
' Access each open order
system.SetAlternateOrder( x )

' Generate a Random value for each order
OrderRanking = Random( totalOpenOrders )

' Set the value as the order sort value.
alternateOrder.SetSortValue( OrderRanking )
Next ' x
' ~~~~~

' Sort All Open Orders in Ascending Order
system.SortOrdersBySortValue()
' ~~~~~

' Show Order Sequence After Orders are Sorted
PRINT
PRINT "After Orders are Sorted"
' Loop through the open orders & Assign Ranking
FOR x = 1 TO totalOpenOrders STEP 1
' Access each open order
system.SetAlternateOrder( x )

' Generate a Random value for each order
PRINT x, alternateOrder.referenceID, alternateOrder.symbol, alternateOrder.SortValue
Next ' x
' ~~~~~

```

Returns:

When script shown above is created it will show the order prior to being sorted and getting a random value for its order ranking property. After the orders have been assigned a random sort value, all the orders are sorted in ascending numerical order. After the sorting the orders are shown again. All reporting output will be found in the Print Output.csv file in the Trading Blox/Results folder.

#	ao.referenceID	ao..symbol	ao..SortValue
Before Orders are Sorted			
1	1000006429	AD	0
2	1000006431	EC	0
3	1000006433	EM	0
4	1000006436	GC2	0
5	1000006437	JY	0
6	1000006439	MP	0
7	1000006441	TY	0
8	1000006444	SI2	0
9	1000006448	HO2	0
10	1000006449	C2	0
11	1000006451	S2	0
After Orders are Sorted			
1	1000006449	C2	1
2	1000006429	AD	4
3	1000006433	EM	4
4	1000006441	TY	4
5	1000006444	SI2	4
6	1000006431	EC	5
7	1000006436	GC2	7
8	1000006439	MP	7
9	1000006451	S2	8
10	1000006437	JY	9
11	1000006448	HO2	11

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and function are available to reference and changes.

Links:

[PRINT](#), [referenceID](#), [SetAlternateOrder](#), [SetSortValue](#), [SortOrdersBySortValue](#), [sortValue](#), [symbol](#)

See Also:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetStopPrice

Sets the stop price for an order to the specified price.

This function **can only** be used in the [Unit Size](#), [Can Add Unit](#), and [Can Fill Order](#) scripts.

It **cannot** be used in the [Entry Order Filled](#) script because the order is already filled. However, it can be used in the instrument. [SetExitStop](#) instead.

NOTE:

Function is only valid for Entry Orders and is ignored for Exit Orders since there is only the protective stop price for exit orders when one is assigned with a Broker Exit Order function.

Syntax:

```
order.SetStopPrice( stopPrice )
```

Parameter:

stopPrice

Description:

Stop price assigned to the order.

Returns:

`order.stopPrice` contains the instrument's protective exit price when one is assigned by a Broker Entry function, or by the `order.SetStopPrice`. When the `order.stopPrice` does not contain a price value, the `order.noStopPrice` will return **True**.

Example:

Unit Size script:

```
' Increment the stop by one tick.
order.SetStopPrice( order.stopPrice + instrument.minimumTick )
```

Can Fill Order script:

```
' Move the stop by the amount of the slippage.
order.SetStopPrice( order.fillPrice - order.entryRisk )
```

Alternate Order Object:

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and function are available to reference and changes.

Links:

[fillPrice](#), [minimumTick](#), [noStopPrice](#), [SetExitStop](#), [stopPrice](#)

See Also:

[Can Add Unit](#), [Can Fill Order](#), [Entry Order Filled](#), [Unit Size](#), [AlternateOrder Object](#),
[AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetTimeInForce

Syntax:**Parameter:****Description:****Returns:****Example:****Alternate Order Object:**

Access to [Order Object](#) properties and functions in other scripts is made possible by using the [AlternateOrder](#) Object as the prefix ahead of the "." property or function. [AlternateOrder](#) Object is discussed below.

Notes:

Always check to be sure the order is available after a [Broker](#) function call using the [system.orderExists](#) function prior to accessing it.

Using the [alternateOrder](#) object when orders are not in context will provide the same access as the Order object. However, the [alternateOrder](#) object must be brought into context using the [system.SetAlternateOrder](#) object function prior to any attempt to use its properties and functions.

Once orders are brought into context their properties and function are available to reference and changes.

Links:**See Also:**

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

Section 9 – Script

Trading Blox provides a **Script Object** to allow any user to create custom methods not available in Trading Blox software. This ability is possible by using any of the functions and properties made available in the **Script Object**.

Functions and Properties listed here provide everything that is needed to create specialize calculations, file handling processes, text formatting, or any other scripting need your module requires its system design.

Script Object Functions:

Functions Name:	Descriptions:
Execute ()	Use this function to call for the execution of the custom scripts needed at this location in the code.
GetSeriesValue ()	Used to access a value from a passed in series. Access is as defined for the series so that auto indexed series are offset based, and non auto indexed series are manual direct index based.
SetReturnValue ()	Used to set the return value to a number or string.
SetReturnValueList ()	Used to set a list of number return values.

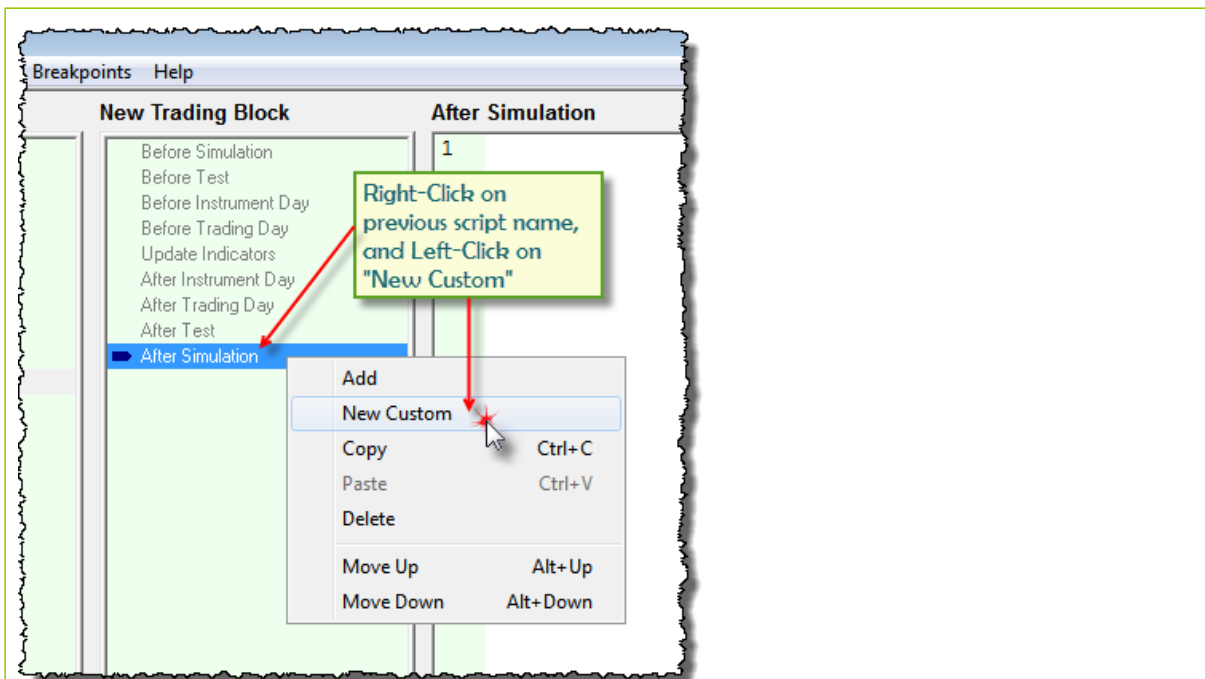
Script Object Properties:

Properties Name:	Description:
parameterCount	The count of the number of parameters passed into the custom function.
parameterList []	Used to access a specific parameter, or a number of parameters.
returnValue	Used to access a custom function's return value number.
returnValueList []	Used to access the list of returned number values.
seriesParameterCount	The count of the Series type parameters passed into the custom function
stringParameterCount	The count of the String type parameters passed into the custom function
stringParameterList []	Used to access one specific String parameter, or a number of String parameters.
stringReturnValue	Used to access a custom function's return value string.

Creating Custom Scripts:

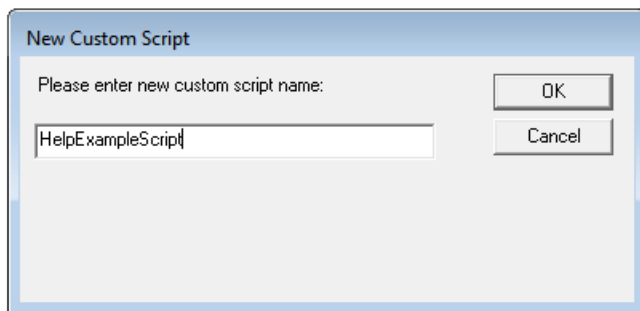
Custom scripts are created by the user adding a script name to the list of scripts in a blox. Name used when creating the script must not be one of the standard script names Trading Blox provides. Instead it should be a name that best describes the purpose of the custom script. Whatever name is used, the name chosen will be the name used when the custom script is called.

To create a custom script, begin by selecting a blox where you want the custom script to be placed. Pick the location in the script listing where you want it to appear, and then Right-Click on the script name just before intended custom script location and a menu will appear.



Custom Script Section Creation Menu Steps

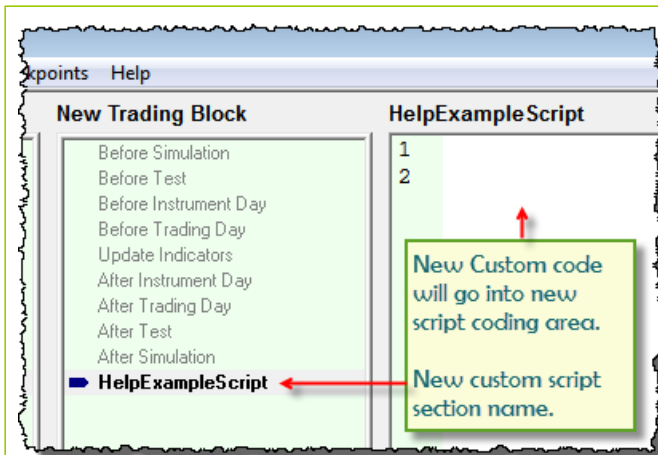
When the New Custom menu item is clicked in the pop-up menu the next dialog will appear.



Name Entered will be Custom Script Section Name

Enter a name that is representative of the custom script section's process so that it helps in identifying what will happen with this script section is executed.

When the name is entered and the New Custom Script dialog OK button is clicked, the new custom script name will appear right after the After Simulation script section where we Right-Clicked to create the custom script.



New Custom Script Section

Using Custom Scripts:

Custom script section can provide new functionality, but they share the same dependency and limitations of the the script section from which they are called to execute. This means the script section that executes the custom script section is a script section that is or can be executed for each instrument in the portfolio without the need for the [LoadSymbol](#) function, then the custom script section will share that access. If the calling script section only executes once each for each [test.currentDay](#), that script section will be required to use the [LoadSymbol](#) function to have access to an instrument.

To get a broader understanding of how the script section calling the custom section will condition the custom script section, please read the details on the [Instrument](#) topic page where each of the standard script sections are listed as having native instrument access, or are script section that only execute once for each [test.currentDate](#) and need special access to instruments.

Custom script section variables share access to the calling script section. This means that variables with the same name in the script section that calls the custom script section will have an impact on the values in the custom script section, and also in the calling script section after the custom script section has completed its execution of its scripts.

This means that it is possible to contaminate or share data between the two script sections. This is also true when a script section calls another custom script section. There can either be contamination or sharing of data that may or many not be intended.

To prevent unintended contamination of variables, use variable names in a script section that are not going to be used within any other script. One simple way is to use a prefix or suffix in the custom script that is added to any of the variable names so as the variable name will be unique to the custom that specific custom script name.

In a custom script section named "Field_Count", all the variable names have a suffix attached to their name using the two primary characters in the custom function name:

Example:

```
' Determine the Size of the String
String_Length_fc = Len(String_Fields_fc)
```

It is unlikely that a `string_length` variable would be created in a standard script section to make it unique because variables can be limited in their data scope reach by how the variables are declared. Custom script sections by default are limited in data scope reach by how they are declared. However, that scoping reach includes the script section that calls the custom script because the process of how

it is executed once called. In simple terms, when a custom script is called it is not any different from how the same code would work had the scripting in the custom script section been typed into the script section that executed the custom script.

Why then do we need custom script sections?

Custom script sections allow us to write code that can do a task that we don't have as a normal function. By creating it as a custom script section we can write the code once and then use it many times.

Once a custom script has been created it can then be used many times because the custom script section can be placed in an Auxiliary module that is easily attached to a system list. A custom script section can also be Right-Clicked, Copied, and then Pasted into another blox for use in that blox.

How many custom script sections are allowed?

There is no known limit, but there is a restriction when it comes to a custom script section name. Each custom script section in a system must be different from any other custom script section in that system. This is necessary because the Script Object's [Execute](#) function will search the entire system list looking for the name used by the [Execute](#) function. If you have more than one a custom script with the same name and it isn't exactly the same code, then the results you will get might not be what you wanted because the search process looks and then uses the first custom script section it finds with the name given to the [Execute](#) function.

Custom Function Example:

Source code shown below was created in 2008 and it is still being used today. Custom function **Field_Count** returns the number of populated fields in a string where each field in the string is separated by a comma. If the string is empty and has a length equal to zero, or only contains the comma, the result returned will be zero. If there is one field and one comma, the return is one. Two fields and two commas returns two, etc.

Example:

```

' =====
' FUNCTION NAME: Field_Count
' =====
' DESCRIPTION: This function returns the number of comma
'              delimited fields in the passed string variable.
' USE:
'   When a count of the number of comma separated values is
'   needed before the GetField functions is used.
' CODE FUNCTION CALL:
'   Function_Result = Script.Execute( "Field_Count", _
'                                   sAnyStringFieldGroup )
'
'   OR
'   PRINT Script.Execute( "Field_Count", sAnyStringFieldGroup )
' -----
' FUNCTION START - Field_Count
' -----
' Function Parameter Variables
VARIABLES: String_Fields_fc           Type: String
' Function Working Variables
VARIABLES: BeginPtr_fc, Field_Count_fc Type: Integer
VARIABLES: EndPtr_fc, String_Length_fc Type: Integer
' ~~~~~
' Assign ParameterList Items to Parameter Variables
' so that Code is Self-Documenting, and so users can
' easily see parameter requirements.
String_Fields_fc = script.stringParameterList[1] ' STRING
' ~~~~~
' Determine the Size of the String
String_Length_fc = Len(String_Fields_fc)

' When there is enough Characters,...
If String_Length_fc > 0 THEN

    BeginPtr_fc = 1 ' Start at the first character
    Field_Count_fc = 1 ' No Delimiters Found Indicate only 1 Field

    ' Look at each character in the string
    For EndPtr_fc = 1 TO String_Length_fc

        ' If the Character is a Delimiter,...
        If FindString( mid(String_Fields_fc, EndPtr_fc, 1), "," ) > -1 THI
            ' Count the field
            Field_Count_fc = Field_Count_fc + 1

            ' Adjust the Pointer's Character Location
            BeginPtr_fc = EndPtr_fc + 1
        ENDIF
    Next
ELSE
    ' Empty String Has No Fields
    Field_Count_fc = 0
ENDIF

```

```
' Field_Count_fc Shows the number of Fields found
script.SetReturnValue( Field_Count_fc ) ' Return Function Value
' -----
' Field_Count - FUNCTION END
' =====
```

Construction Temporarily Paused Here!

Example:

```
' Create a custom script named "Multiply Two Numbers" with the following code
script.SetReturnValue( script.parameterList[1] * script.parameterList[2] )

' Call this new script from another script or block:
PRINT script.Execute( "Multiply Two Numbers", 7, 8 )
```

USER FUNCTION NOTES:

NOTE:

```
' Each Parameter List TYPE is parsed into each OF the following variable
' containers based upon the LEFT TO RIGHT sequence IN which the parameter
' value IS listed when it IS called, AND also the TYPE OF variable being
' passed:
Script.ParameterList[]          Use FOR INTEGER OR FLOAT parameters
OR
Script.StringParameterList[]    Use FOR STRING parameters
```


CREATING & USING A USER FUNCTION:

```

'
' Enter your code (see example Blox listed above). With the code entered
' you can then call this script from anywhere in the system by using the
' following process:

    lResult = Script.Execute( "User_Function_Name", [parameterlist...] )

[parameterlist...] is where you pass values to your new function (see
example Blox listed above).
PRINT lResult      ' Sends the User Function result to the Log Window
                   ' or the Print Output.csv file.

Any_Var = lResult  ' Assigns the User Function Result to Any_Var

' When you call a User Function like this, the calculation results are
' returned to variable you place on the left side of the calling statement
' In this case, the variable "lResult" will hold the User Function
' calculation value you created in your function script.
'
' You can call a User Function to print directly to a PRINT statement:

    PRINT Script.Execute( "User_Function_Name", [parameterlist...] )

' In this method the function's result will print directly to TBB's Print
' Output.csv file or Log Window.

' When you create a User Function script, you'll need to assign the
' calculation results to a script's return property. This is done by
' using one of the following methods:
    ( )
    ' No Longer Supported - Replaced by script.SetReturnValue()
    Script.SetStringReturnValue( ) ' Removed in version 4.x

    Script.SetReturnValue( Any_Num ) ' Assigns a numeric or text/string

' You can call a User Function without assigning a value to capture its re
' sult. To do it that way, the calling statement would look like this:

    Script.Execute( "User_Function_Name", [parameterlist...] )

' In this case you would need to use one of the following properties to
' access the User Function results:

    lResult = Script.ReturnValue      ' Use for INTEGER OR FLOAT Return
OR
    Any_Text = Script.StringReturnValue ' Use for STRING Return

```

9.1 Script Functions

Script Object Functions:

Functions Name:	Descriptions:
Execute ()	Use this function to call for the execution of the custom scripts needed at this location in the code.
GetSeriesValue ()	Used to access a value from a passed in series. Access is as defined for the series so that auto indexed series are offset based, and non auto indexed series are manual direct index based.
SetReturnValue ()	Used to set the return value to a number or string.
SetReturnValueList ()	Used to set a list of number return values.

Functions used in the custom function:

```
' Used to set the return value to a number or string
script.SetReturnValue( value or string value, or value and string value

' Used to set a list of number return values.
script.SetReturnValueList( value1, value2, value3... )

' Used to access a value from a passed in series. Access is as defined :
' the series so that auto indexed series are offset based, and non auto
' indexed series are manual direct index based.
script.GetSeriesValue( seriesParameterIndex, offsetIndex )

' These are Subroutine Processes for Setting the User Function's
RETURN value
Script.SetStringReturnValue( STRING value ) ' Use FOR a STRING RETURN
```

Execute

Custom scripts are called for execution from within any of the regular Trading Blox scripts, or from within another custom script.

Syntax:

```
script.Execute( "CustomScriptName", [parameterlist...] )
```

Parameter:**Description:**

"CustomScriptName"

Custom script's given name when it was created.

parameterlist

Parameters values custom script will require to execute correctly.

Returns:

Custom scripts can return a value, but it isn't required.

Custom scripts that create a new function for processing information usually will return a Floating number or a string.

Example:

```
' Custom script adds two numbers together
Print script.Execute( "AddTwoNumbers", 2, 2)
```

Return:

4

Example:

```
' Custom script adds two numbers together
value = script.Execute( "AddTwoNumbers", 2, 2)
Print value
```

Return:

4

Links:

[Script](#)

See Also:

GetSeriesValue

Used to access a value from a passed in series. Access is as defined for the series so that auto indexed series are offset based, and non auto indexed series are manual direct index based.

Syntax:

```
script.GetSeriesValue( aSeriesName, )
```

Parameter:**Description:****Example:****Results:****Links:****See Also:**

SetReturnValue

Used this function to return a value from a Number or a String.

Syntax:

```
script.SetReturnValue( anyValue )
```

Parameter:

anyValue

Description:

Value can be any numeric or text value that needs to be returned.

Example:

```
' Estimate Size to Remove
SizeToRemove_GRQ = (iMinLotCount_GRQ * reductionRate)

' Return quantity Size to remove
script.SetReturnValue( SizeToRemove_GRQ )
```

Results:

Returns value contained withing the variable SizeToRemove_GRQ

Links:

[Script](#), [Script Functions](#), [Script Properties](#)

See Also:

SetReturnValueList

Syntax:

Parameter:	Description:

Example:

Results:

Links:

See Also:

9.2 Script Properties

Script Object Properties:

Properties Name:	Description:
<code>parameterCount</code>	The count of the number of parameters passed into the custom function.
<code>parameterList []</code>	Used to access a specific parameter, or a number of parameters.
<code>returnValue</code>	Used to access a custom function's return value number.
<code>returnValueList []</code>	Used to access the list of returned number values.
<code>seriesParameterCount</code>	The count of the Series type parameters passed into the custom function
<code>stringParameterCount</code>	The count of the String type parameters passed into the custom function
<code>stringParameterList []</code>	Used to access one specific String parameter, or a number of String parameters.
<code>stringReturnValue</code>	Used to access a custom function's return value string.

Parameters used in the custom function:

`script.parameterList[]` -- Used to access a number parameter
`script.stringParameterList[]` -- Used to access a string parameter
`script.parameterCount` -- The count of number parameters passed into the custom function
`script.stringParameterCount` -- The count of string parameters passed into the custom function
`script.seriesParameterCount` -- The count of series parameters passed into the custom function

Parameters used after a call to a custom function from the calling script:

`script.returnValue` -- Used to access the number return value
`script.stringReturnValue` -- Used to access the string return value
`script.returnValueList[]` -- Used to access the list of returned number values

NEW Parameters:

```

' Variable Containers of Passed Values to User Created Functions
Script.ParameterList[]      ' Use for INTEGER & FLOAT values
Script.StringParameterList[] ' Use for STRING values

' Quantity Count of Passed Parameter Variables in User Function
Script.ParameterCount      ' Count of INTEGER & FLOAT variables
Script.StringParameterCount ' Count of STRING variables

' Return Variable Container Of Last User Function Result
Script.ReturnValue         ' Use for INTEGER OR FLOAT Returns
Script.StringReturnValue    ' Use for STRING Return
  
```

ParameterCount

Syntax:

Parameter:	Description:

Example:

Results:

Links:

See Also:

ParameterList**Syntax:****Parameter:****Description:****Example:****Results:****Links:****See Also:**

ReturnValue

Used to return a numeric value from a custom function.

Syntax:

```
lResult = script.returnValue
```

Parameter:

<None>

Description:

Values assigned using the **script.SetReturnValue()** function.

Example:

```
' Set the return value to 4
script.SetReturnValue( 4 )
' Assign the return value to the long integer lResult
lResult = script.returnValue
' Print the value of lResult
Print lResult
```

Results:

Print statement will display: **4**

Links:

[Script](#), [Script Functions](#)

See Also:

ReturnValueList**Syntax:****Parameter:****Description:****Example:****Results:****Links:****See Also:**

SeriesParameterCount

Syntax:

Parameter:	Description:

Example:

Results:

Links:

See Also:

StringParameterCount

Syntax:

Parameter:

Description:

Example:

Results:

Links:

See Also:

StringParameterList

Syntax:

Parameter:	Description:

Example:

Results:

Links:

See Also:

StringReturnValue**Syntax:****Parameter:****Description:****Example:****Results:****Links:****See Also:**

Section 10 – System

The system object contains properties which describe system level attributes.

Since systems control the portfolio, the most common use for system properties is to determine the number of instruments in the current portfolio.

System objects can also be used to determine dynamic correlations between instruments in the portfolio.

System Function & Control Areas:	Description:
Accessing System Portfolio Instruments	
SetAlternateSystem	This is a test object function that provides access to any of the multiple systems that are executing in the current simulation test.
System Functions	
System Properties	

10.1 Global Suite System

The global suite system is a special type of system. If a system has the same name as the suite, it will be by definition a Global Suite System.

The scripts attached to this global suite system will be processed in a particular order: *before* the others for the before scripts, and *after* the others for the after scripts.

The following "day/bar" scripts can be used in a GSS:

- o Before Simulation,
- o Before Test,
- o Before Trading Day,
- o Before Bar,
- o Before Order Execution
- o After Bar,
- o After Trading Day,
- o After Test,
- o After Simulation

The following "Order" script:

- o Entry Order Filled,
- o Exit Order Filled,
- o Can Add Unit,
- o Can Fill Order

For the above "Order" scripts, the default system, default instrument and default order context is taken from the system that originated the order. This is different from the non orders scripts, for which the default system is the GSS, and the default instrument is null. The order object is null by default, but can be valid after a call to the `alternateBroker` function. Use the `alternateSystem.OrderExists` property to check if the order is available for access.

The Global Suite System has a system index of 0, and a system name of "Global Parameters".

Use `system.IsGlobalSuiteSystem` property to check if the system is a Global Suite System, if this is important. Note that for order scripts, the system will be from the system originating the order, so this property will return false. In the non order scripts, the system is the actual GSS, so this property will return true. The GSS does not trade, so there is no instrument list, no positions, and no equity curve. But BPV's and IPV's can be created, and accessed using the `SetAlternateSystem` and `SetAlternateInstrument` functions. In this way the GSS can be used as a global storage location for overall control and computation of test level variables.

For cases where the physical location of the script, such as system, is important, use the [Block](#) object properties. An example might be when a block is used in both a GSS and regular system, and uses an order script to process orders. The order script will be called twice, once in originating system and once in the GSS. Checking if the `block.systemIndex` will indicate if the script is really in a GSS.

The Net Risk block in the Blox Marketplace is a good example of how to loop over all instruments, in all systems, and compute a suite level value for each market.

```
PRINT
PRINT test.currentDate
```

```

totalRisk = 0

' Loop over all the instruments that are used in this test. Includes
all systems and all support forex conversion markets.
FOR testInstrumentIndex = 1 to test.instrumentCount STEP 1

' Get the symbol name.
PRINT "Getting instrument number", testInstrumentIndex
suiteInstrumentSymbol = test.instrumentList[ testInstrumentIndex ]
PRINT "Processing", suiteInstrumentSymbol

' Get the suite level instrument.
if suiteInstrument.LoadSymbol( suiteInstrumentSymbol, 0 ) THEN

' Reset our net position to zero for this instrument.
suiteInstrument.netRisk = 0

' Loop over all the systems in the test.
FOR systemIndex = 1 to test.systemCount STEP 1

' Set the alternate system so that we can use the name,
or other system properties.
PRINT "Setting to system index", systemIndex
test.SetAlternateSystem( systemIndex )
PRINT "Processing for system", alternateSystem.name

' Load the instrument symbol combo.
IF systemInstrument.LoadSymbol( suiteInstrumentSymbol,
systemIndex ) THEN
PRINT "Loaded", systemInstrument.symbol

' If this instrument is in the portfolio for the
system, then check the position.
IF systemInstrument.InPortfolio THEN
PRINT "In portfolio for system",
alternateSystem.name, "with risk of",
systemInstrument.currentPositionRisk

' To get the net risk, we use positive for
long risk and negative for short risk.
IF systemInstrument.position = LONG THEN
PRINT "Long"
suiteInstrument.netRisk =
suiteInstrument.netRisk + systemInstrument.currentPositionRisk
ENDIF

IF systemInstrument.position = SHORT THEN
PRINT "Short"
suiteInstrument.netRisk =
suiteInstrument.netRisk - systemInstrument.currentPositionRisk
ENDIF

```

```

ELSE
    PRINT "Not in portfolio for system",
alternateSystem.name
ENDIF

ELSE
    PRINT "Unable to load", suiteInstrumentSymbol
ENDIF

NEXT

' Whether positive or negative, it's still risk.
suiteInstrument.netRisk = ABS( suiteInstrument.netRisk )

' Print out the net risk for this instrument.
PRINT "Net Risk", suiteInstrument.netRisk
totalRisk = totalRisk + suiteInstrument.netRisk

ELSE
    PRINT "Unable to load", suiteInstrumentSymbol
ENDIF

NEXT

totalRisk = totalRisk / test.totalEquity * 100

```

10.2 System Functions

Referencing Sytem Object function will always used the "system." object referencing name in the prefix area when calling any of the functions listed here:

Example:

```

' Sort Instruments by Dictionary Order Value:
system.SortInstrumentList( 3 )

```

System Function:	Description:
RankInstruments	Virtually ranks the instruments using the defined long and short ranking. Sets the corresponding Long Rank and Short Rank ordinal values based on the Long Ranking Value and Short Ranking Value respectively. Long Rank is highest to lowest, whereas Short Rank is lowest to highest. Only primed markets ready to trade are ranked.
SetAccountNumber ()	Sets the IB account number for all orders in the system.
SetAlternateOrder (index)	Sets the alternateOrder object by index. Used when looping over all the open orders. To access the order, use the alternateOrder object. This object acts just like the default

	Order Object and has the same properties and functions.																
SetClearingIntent ()	Set to "IB" to clear the system orders in IB, and set to "AWAY" when sending to IB but clearing elsewhere																
SetVirtual (true/false)	Sets whether the system will be treated as virtual or not. Defaults to false at the beginning of each simulation. If set to virtual, then the results of the system will not be included in the test results.																
SortInstrumentList (method)	Sorts the physical instrument list that is used for the simulation loop using the method indicated: <table border="1"> <thead> <tr> <th>Method Value:</th> <th>Sorting Method Description:</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Long Ranked number.</td> </tr> <tr> <td>2</td> <td>Long Ranking script assigned value.</td> </tr> <tr> <td>3</td> <td>Dictionary assigned Original Order Sort value.</td> </tr> <tr> <td>4</td> <td>Custom Sort Value order.</td> </tr> <tr> <td>5</td> <td>Alphabetical by Symbol</td> </tr> <tr> <td>6</td> <td>Alphabetical by Group.</td> </tr> <tr> <td></td> <td>Note: All markets are sorted regardless of whether they are primed.</td> </tr> </tbody> </table>	Method Value:	Sorting Method Description:	1	Long Ranked number.	2	Long Ranking script assigned value.	3	Dictionary assigned Original Order Sort value.	4	Custom Sort Value order.	5	Alphabetical by Symbol	6	Alphabetical by Group.		Note: All markets are sorted regardless of whether they are primed.
Method Value:	Sorting Method Description:																
1	Long Ranked number.																
2	Long Ranking script assigned value.																
3	Dictionary assigned Original Order Sort value.																
4	Custom Sort Value order.																
5	Alphabetical by Symbol																
6	Alphabetical by Group.																
	Note: All markets are sorted regardless of whether they are primed.																
SortOrdersBySortValue	Sorts the open order list by the order sort value, as set into each order by order.SetSortValue. This function should only be used in the Before Order Execution script.																
<p>Links:</p> <p>System Properties</p> <p>See Also:</p>																	

Accessing System Portfolio Instruments

There are times when you might want to access one or more of the portfolio's instruments, an instrument properties, or indicators. This is easily accomplished with the instrument's [LoadSymbol](#) function. To make the following code work, to set an instrument variable to a particular instrument in the portfolio by numeric index.

Example:

Accessing instruments out of their normal context scripts requires the use of a Instrument container class variable, which is shown in this BPV dialog:

Any name can be used for the container variable. When accessing any IPV out of context, the variable name must be used in the prefix or object name location. In the code script shown below, the symbol is accessed by using the variable name and the instrument property for the symbol, which is symbol -- `portfolioInstrument.symbol`

```
' Local declared variables
VARIABLES: instrumentCount, x Type: Integer

' Get the instrument count.
instrumentCount = system.totalInstruments

' Loop printing the symbol for each instrument.
For x = 1 TO instrumentCount STEP 1

  ' Set the portfolio instrument.
  portfolioInstrument.LoadSymbol( x )

  ' Print out the file name.
  If portfolioInstrument.inPortfolio THEN
    PRINT x, ". Portfolio contains: ", portfolioInstrument.symbol
  ENDIF
Next ' x
```

In this code section the script is using the System's `totalInstruments` property. This property contains the total number of symbols listed in the portfolio so the `For` loop structure would know how many times it should loop to get access to all the portfolio's instruments.

When the above is executed with the Canadian Dollar, Euro, Feeder Cattle and Corn in the portfolio, the main screen's Log Window will show this information:

Returns:

- 1 - Portfolio contains: CD
- 2 - Portfolio contains: EC

3 - Portfolio contains: FC
4 - Portfolio contains: C2

Links:

[Instrument Loading](#), [System Properties](#),

See Also:

RankInstruments

Syntax:

Parameter:

Description:

Example:

Results:

Links:

See Also:

SetAccountNumber

Syntax:

Parameter:	Description:

Example:

Results:

Links:

See Also:

SetAlternateOrder**Syntax:****Parameter:****Description:****Example:****Results:****Links:****See Also:****10.3 System Properties**

The following properties refer to the system which contains the block in which a script runs. If you have multiple systems in your test, these values will be different for Blox that run in those different systems.

The system equity numbers are updated prior to the After Trading Day script, so in this script they will represent the most current equity.

Property Name:	Description:
name	the name of the system. Useful for printing.
tradingEquity	<p>the amount of money available for trading by the current system.</p> <p>The trading equity is determined by the Global Setting parameter "Trading Equity Base," the system allocation slider, the leverage amount, and the drawdown reduction amount.</p> <p>For example, if Trading Equity Base is set to "Total Equity", then the value of <code>system.tradingEquity</code> is equal to the <code>test.totalEquity</code> multiplied by the leverage multiplied by the system's allocation percentage as set by the slider and reduced by the drawdown reduction threshold/amount.</p> <p>For order generation when a value has been entered for Order</p>

	Generation Equity this number replaces the test total equity number. So the trading equity is calculated from there.
<code>totalEquity[]</code>	<p>this starts out as <code>system.tradingEquity</code>, and is then affected by the total profits, losses of trades by the system.</p> <p>This property is Indexable, so you can access the system's total equity for bars in the past. This value will be the same as <code>test.totalEquity</code>, minus interest, if you only have one system in your test with a 100% allocation. For order generation this is the Order Generation Equity times the allocation. Note that this value is from the close of the prior day.</p>
<code>closedEquity[]</code>	The closed equity of the system as of the close of the prior day.
<code>currentOpenEquity</code>	The current open equity of the system. Dynamically takes into consideration positions as they are exited.
<code>currentClosedEquity</code>	The current closed equity of the system. Dynamically takes into consideration positions as they are exited.
<code>coreEquity</code>	The core equity of the system. Dynamically computed based on current stop prices.
<code>cash</code>	the current cash of the system. System closed equity minus open position margin (for futures) or cost of purchase (for stocks).
<code>allocationPercent</code>	the percentage of equity available to the current system. Set using the slider in the global parameters
<code>maximumDrawdown</code>	the maximum drawdown for the system, in percent from the peak (a positive number between 0% and 100%).
<code>currentDrawdown</code>	the current drawdown for the system, in percent from the peak (a positive number between 0% and 100%).
<code>currentRisk</code>	the risk in dollars of all open positions in the system. The risk for a given position is determined by looking at the difference between the close and the protect stop for each unit
<code>totalMargin</code>	the total margin across all instruments. For futures this is the sum of the futures margin, for stocks it is the sum of the cash required to buy the stocks.
<code>totalInstruments</code>	the total number of instruments in the portfolio being tested.
<code>tradingInstruments</code>	the total number of instruments in the portfolio being tested that have price information for the current test date, and are primed.
<code>canTradeInstruments</code>	the total number of instruments in the portfolio being tested that have price information for the current test date, are primed, and are allowed to trade by the portfolio manager.
<code>totalPositions</code>	the number of instruments in the portfolio with a position of LONG or SHORT with a non zero position size
<code>totalLongPositions</code>	the number of instruments in the portfolio with a long position with a non zero position size
<code>totalShortPositions</code>	the number of instruments in the portfolio with a short position with a non zero position size

<code>totalUnits</code>	the total number of units for long and short positions with a non zero position size
<code>totalLongUnits</code>	the total units for long positions only with a non zero position size
<code>totalShortUnits</code>	the total units for short positions only with a non zero position size
<code>index</code>	the index number of the system, from 1 to the number of systems in the test.
<code>totalOpenOrders</code>	the total number of open orders for the system
<code>portfolioName</code>	the name of the portfolio being used for the system
<code>orderExists</code>	returns true if the default Order object has context, and false if the Order object is null and cannot be accessed
<code>IsGlobalSuiteSystem</code>	returns true if the system is a Global Suite System

Links:
[System Functions](#)

See Also:

orderExists

This property is used to determine if an order exists prior to the order object properties and functions being accessed. Orders that are not in context, or do not exist when scripts attempt to access them will cause a run-time error during execution.

TYPE:	Description:
<code>orderExists()</code>	Returns TRUE when an order exists, and if the default Order , or <code>AlternateOrder</code> object has context. Most orders that exist can be accessed after the <code>Broker</code> function that creates them, and only when the order has not been rejected. When an order is not available because it wasn't created, was rejected or it was executed previously, this property will return a Null or FALSE condition.

Example:

```
'  Generate Long Entry on Next open, without
'  a protective exit price
broker.EnterLongOnOpen

'  Create details why order is created.
sRuleLabel = "Order Details go here."

'  When Broker Order Exist,...
If system.OrderExists() THEN
    '  Apply Exit Stop Update Information
    order.SetOrderReportMessage( sRuleLabel)

    '  Apply Exit Stop Update Information
    order.SetRuleLabel( sRuleLabel)
ENDIF '  system.orderExists
```

Returns:

A **True** condition will be returned when an order exists and is in context.

Links:**See Also:**

[Data Group and Types](#)

Section 11 – Test

The Test Trading Object contains test-level properties. They contain information about the overall test.

Test Object Types:	Description:
Equity Properties	Equity properties for every system in the test.
General Properties	Test object properties for every system in the test.
String Arrays	Test level global string array functions and properties.
Miscellaneous Functions	Test level functions that provide access to paths, and control how test are executed or terminated and which reporting features are changed or added.
Test Statistics	After test level statistic & summary reporting functions.
Trade Properties	Test level closed trade details for all the instruments in the test.

11.1 Equity Properties

These properties access equity values for every system in the test (a test can run multiple systems at once). They are also indexable - if you put a number in brackets, you will get the historical value from that day. For instance, `Test.ClosedEquity[5]` will be the closed equity 5 days ago. Several of these properties are also graphed when the results for a test are displayed.

Bar Indexing:

Properties listed with a '[' following them may be indexed using a number which determines which day's data to access. There are also built in constants for 'today' and 'yesterday' which can be used. For example:

```
' Test Total Equity from 5-bars ago is assigned to the variable equity
equity = Test.TotalEquity[ 5 ]
```

OR

```
' Yesterday's Instrument's Date is assigned to the variable yesterday.
' A value of 1 references date record just before this date.
YesterdaysDate = Instrument.Date[ 1 ]
```

When `YesterdaysDate` is referenced it will return the date value in the data record just before the current instrument date.

NOTE:

The most current equity numbers are from the prior day, since the final equity figures are not determined until the end of the day when all scripting has finished for the `Test.CurrentDate` value.

Update End-of-Day Equity numbers are available when scripting reaches the **After Trading Day** script section.

Equity Properties	Property Descriptions
cash	Current cash of the test. Test closed equity minus open position margin (for futures) or cost of purchase (for stocks)..
closedEquity[]	Total Closed Equity for all systems. For order generation this is the Order Generation Equity minus Open Equity.
closedEquityHigh	the current high water mark for closed equity. Used to compute the currentClosedDrawdown.
coreEquity	Core Equity at each Test.CurrentDay location of a test (not indexed).
currentClosedDrawdown	the current bar's closed equity drawdown.
currentDrawdown[]	the percent of total equity drawdown. Graphed as the "Drawdown" graph under test results.
currentRisk[]	the total percent of total equity at risk, based on the close for markets with open positions minus the stop price for those positions. Graphed as the "Total Risk Profile".
otherExpense	the total other expenses as set by UpdateOtherExpenses

Equity Properties	Property Descriptions
startingEquity	the equity as of the start of the simulation. Equal to the Test Starting Equity as specified in the Global Parameters
totalEquity[]	<p>Starts as the Test Starting Equity as specified in Global Parameters. It is then affected by the total profits, losses, and interest of trades by all systems. Graphed as the "Equity Curve" graph. For order generation this is the Order Generation Equity.</p> <p>Note: This property is not affected by any of the following Global Parameter settings: System Allocation Slider Drawdown Reduction Threshold, or its Amount, Choice of Base Equity selection: Total Equity or Closed Equity.</p>
totalEquityHigh	the current high water mark for total equity. Used to compute the currentDrawdown.
totalMargin	the total current margin of all open positions in the test
vadi[]	<p>VADI is the acronym for "Value Added Daily Index". VADI calculations start at the start of trading equity, and it increases and decreases as a ratio of profit/loss as a percent of trading equity. VADI is net of capital adds and draws, and it includes accrued fees, whereas equity only includes booked fees.</p> <p>Note: Fees are accrued on a daily basis, and booked as defined in global parameters (daily/monthly/quarterly/yearly).</p>

11.2 General Properties

The following properties refer to the test object and they will be the same regardless from what system they are referenced.

All Test Object properties are used with the "test." object prefix.

Example:

```
PRINT "Test Name: ", test.name
PRINT "Order Report Path: ", test.orderReportPath
```

Properties:	Description:
abortTestPending	Returns true when the Abort Test or Abort Simulation functions have been used.
baseCurrency	Returns the ISO code of the system wide base currency
baseCurrencyBorrowRate	Borrow rate of the system wide base currency
baseCurrencyLendRate	Lend rate of the system wide base currency
CapitalAddsDraws	Total capital adds and draws to date, from the capital adds draws file.
currentDate	Current simulation date. In YYYYMMDD format.
currentDay	Number count of the current day. Count starts at one on the day of the Test Start. Also used as the Test Bar when using Intraday Data, so this can be the "Test Bar Number" when using date and time.
currentParameterTest	Number of the current active parameter step test
currentTime	Current simulation time. In HHMM format.
feesIncentiveAccrued	Incentive fees accrued but not yet booked. Note: Fees are accrued on a daily basis, and booked as defined in global parameters (daily/monthly/quarterly/yearly).
feesIncentiveTotal	Total incentive fees booked to date.
feesManagementAccrued	Management fees accrued but not yet booked. Note: Fees are accrued on a daily basis, and booked as defined in global parameters (daily/monthly/quarterly/yearly).
feesManagementTotal	Total management fees booked to date.
forexDataPath	Provides the full path to the Forex Files. Default Forex path: C:\Trading Blox\Data\Forex\
futuresDataPath	Provides the full path to the Futures Files. Default Forex path: C:\Trading Blox\Data\Futures\
instrumentCount	Total number of instruments being tested across all systems. This includes forex conversion files and if systems are using different portfolio types,

	could include stocks, futures, and forex markets. Does not include markets loaded using the LoadSymbol function.
instrumentList[]	List of all instruments in all the system portfolios, including the Forex conversion files. Indexed from 1 to InstrumentCount. Returns the full symbol, such as F:CL or S:IBM.
leverage	Leverage as set in Global Parameters
name	Name of the Current Test Suite.
orderGenerationBar	Returns true if the current bar is after the test end record, and therefore the order generation bar.
orderGenerationTest	Returns true if the test is generating orders, rather than just running a performance test.
orderReportPath	Full path of the current order report
primeStart	Earliest date for all loaded data for any instrument
resultsReportPath	Path of the results folder for this test. Used to access charts and graphs in the test results folder for display.
stockDataPath	Provides the full path to the Stock Files. Default Forex path: C:\Trading Blox\Data\Stocks\
summaryResultsPath	Full path of the current test results saved file location.
systemCount	Number of systems to be tested.
testEnd	Test End date. The user entered end date, or the end of data, which ever comes first.
testStart	Test Start date. The first trading day equal to, or after the user entered start date, or the start of data, which ever is later.
threadCount	Number of active threads available in this Simulation test.
threadIndex	Active thread index of this simulation test. Each thread index up to ThreadCount will run concurrently, so if variables, or information needs to be passed from one thread to another, make sure the ThreadIndex is a match.
timeIncrement	Returns the TimeIncrement used for the test loop when using Intraday data. Output format is HHMM (Hour-Minute) order.
timeStamp	Start Time stamp of this test. Used to access charts, graphs, and other results for the test run.
totalParameterTests	Total number of distinct parameter tests
walkForwardStatus	Returns 0 for a normal test, 1 for the optimization portion of the walk forward test, and 2 for the out of sample portion of the walk forward test.

Global Parameters report the current setting used in the Global Settings.

Global Parameter Access	
accountForContract Rolls	
accountForForexCarry	

commissionPercentValue	
commissionPerContract	
commissionPerShare	
commissionPerTrade	
convertStockSplit	
earnDividends	
earnInterest	
entrydayRetracement	
forexTradeSize	
ignoreTestPositions	
incrementTestStart	
maxMarginEquityPercent	
maxVolumePerTrade	
minimumFuturesVolume	
minimumSlippage	
minimumStockVolume	
payMargin	
rollSlippageATR	
setTestDuration	
slippagePercent	
tradeLockLimit	
tradeOnTick	
useBrokerPositions	
usePipBasedSlippage	
useStartDateStepping	

At the beginning of each trading simulation day, the test's date is set to the current trading day while the instrument date is set to the previous trading day. This prevents the creation of postdictive errors or errors where trading system logic is allowed to access information that is not available in actual trading. In effect, postdictive errors are errors which rely on seeing the future.

OrderReportPath

Property returns the full path and folder name of the current test Suite.

Syntax:

CurrentOrderFolderPath = [test.orderReportPath](#)

Parameter:

<None>

Description:

Output is a [String](#) containing the current Trading Blox path for saving order files.

Notes:

Use this property to discover the current order report saving location.

Example:**Links:**

WWW.TRADING-SOFTWARE-DOWNLOAD.COM

ResultsReportPath

Property returns the full path and folder name of the current test Suite.

Syntax:

```
CurrentTestResultsFolderPath = test.resultsReportPath
```

Parameter:	Description:
<None>	<p>Output is a String containing the newly created performance results files path-name and folder-name used in the Summary Performance Test Results page.</p> <p>Used to access charts and graphs in the test results folder for display, or for accessing Trade and Equity Logs enabled in the Trading Blox Preferences Reporting sections.</p>

Notes:

Use this property to discover the current test suite and folder full path details.

Property makes it easy to store and access custom charts and other test files.

Example:

```
' Assign results reporting folder path and name to a variable.
CurrentTestResultsFolderPath = test.resultsReportPath
```

OR

```
' Display path and folder name to the Print Output.csv
' file or Main screen Log Window.
Print test.resultsReportPath
```

Returns:

```
' Printed output would look something like...
"C:\Trading Blox\Results\Test 2013-01-08_08_47_55"
```

Links:

[Print](#)

SummaryResultsPath

Property returns the full path and folder name of the current test Suite.

Syntax:

```
CurrentOrderFolderPath = test.summaryResultsPath
```

Parameter:

<None>

Description:

Output is a [String](#) containing the current Trading Blox path for saving the current test result files.

Notes:

Use this property to discover the current test results file saving location.

Example:**Links:**

11.3 Test String Arrays

Test scoped String arrays provide access to the Test-Level String Table that allows a user to place, and then retrieve text information into a multi-dimensional table.

For access to the global test level string arrays, use the following functions and properties:

```
test.CreateStringArray( arrayCount, elementCount, stringLength )
```

Creates multiple (arrayCount) string arrays each with a fixed number of elements (elementCount) and a fixed string length (stringLength) for each string in the array.

```
test.SortStringArray( arrayIndex, direction, elementCount )
```

Sorts one of the string arrays (arrayIndex) using the direction (1 for ascending and -1 for descending). Only sorts the first elementCount number of elements in the array.

```
string = test.GetStringArrayElement( arrayIndex, elementIndex )
```

Returns a string from the arrayIndex string array at elementIndex element.

```
test.SetStringArrayElement( arrayIndex, elementIndex, string )
```

Sets a string into the arrayIndex array at elementIndex element.

11.4 Miscellaneous Functions

Test level properties and functions that provide access to paths, and control how test are executed or terminated and which reporting features are changed or added.

Test Function Name:	Description:
AbortSimulation ("message")	Aborts the simulation
AbortTest	Stops the current parameter test only. The simulation will continue with the next parameter test
AddStatistic	Adds a new custom statistic which will show up on the summary list
CapitalAddsDraws	
GetSteppedParameter (paramIndex, paramValueIndex)	Returns the attributes of each stepped parameter.
SetAutoPriming (True/False)	Use in Before Simulation Script to disable auto priming.
SetAlternateSystem (sysIndex)	AlternateSystem object is a test function that provides access to any of the multiple systems that are executing in the current simulation test.
SetChartSimulationHtml	Function creates end of test tasks to automatically display a custom chart images in the area below the Stepped Parameter Summary Performance Table .
SetChartTestHtml	Function creates end of test tasks to automatically display a custom chart images in the area below the BPV Custom Graphs are displayed.
SetDisplayOrderReport(True/False)	Sets whether the order report will display at the end of the order generation run. Defaults to true at the beginning of each test run.
SetEarliestTime(HHMM)	Used in the Before Simulation script to set the earliest time each day on which the test loop will start. Defaults to the earliest time for any bar of data for all instruments in the test.
SetGeneratingOrders (True/False)	Use in Before Test script to set whether a test run is going to generate orders, or just run a normal back test.
SetGoodnessToChart(listofstats)	Creates multi parameter charts (contour and 3D) for all status in the list, such as ("Mar, Sharpe")
SetLatestTime(HHMM)	Used in the Before Simulation script to set the latest time each day on which the test loop will end. Defaults to the latest time for any bar of data for all instruments in the test.
SetListNonTradedInstruments (True/False)	Sets the preference to list all the instruments in the portfolio that did not trade. If set to true, then all non traded instruments will be listed in the trade chart with a zero entry. If set to false, then only the instruments with trades will be listed in the trade chart. This
SetSilentTestRun (True/False)	Used in the Before Test Script to suppress any output from the test run.
SetSmartFillExit	Sets the global Smart Fill Exit parameter to true or false in scripting. Useful when the system has many exits placed each bar, and requires this functionality.

SetStartEndDates(startDate, endDate)	Dynamically sets the start and end dates of the test. Use in the Before Test script. Must be between the original user set start end dates because that controls how much data was loaded.
SetStartingEquity(equityValue)	Dynamically sets the starting equity of the test. Use in the Before Test script.
SetTimeIncrement(HHMM)	Used in the Before Simulation script to set the time increment for the test. Defaults to the smallest time increment for all data in all instruments in the test.
UpdateOtherExpenses	Adds or subtracts other expenses

AbortSimulation

Stops the entire simulation by aborting all parameter testing.

Syntax:

```
test.AbortSimulation( [message] )
```

Parameter:	Description:
[message]	Information about the reason why the test was aborted.

Returns:

<none>

Example:

```
' Abort the simulation because we could not load our data.  
test.AbortSimulation( "Test Completed" )
```

Links:

[Message Box](#)

See Also:

AbortTest

Stops the current parameter run but not the simulation. The simulation will continue with the next parameter test.

Test results can be filtered in the after test script using the [abortTest](#) or [SetSilentTestRun](#) functions.

This function sets the test.abortTestPending flag to true, so that additional processing in the script could be skipped if necessary. The test will actually abort after the script has finished.

Syntax:

```
test.AbortTest
```

Parameter:

<none>

Description:**Returns:**

<none>

Example:

```
' Abort the test because of invalid parameters  
test.AbortTest
```

Links:**See Also:**

AddStatistic

You can add custom statistics to the summary page for sorting. Best if used in After Test script after the statistic has been calculated.

Adds a new statistic which will show up in the test summary list and can be sorted there.

Syntax:

```
test.AddStatistic( statisticName, value, [decimal places], [type] )
```

Parameter:	Description:
statisticName	String name of the statistic to be added
value	Value of that statistic
[decimal places]	Optional number of decimal places for display of float numbers. Default is 2 if left out
[type]	Places is still optional. Float is default, although if value is a string the String type is assumed.

Notes:**Optional Types:**

"Integer"	Prints as an integer, truncated.
"Float"	Prints as a floating point number, default.
"Decimal"	Prints as a decimal, multiplied by 100 and a % added.
"Currency"	Prints as a comma delimited number with currency symbol prefixed.
"Date"	Prints as a date string YYYY-MM-DD.
"String"	Prints as a string. Must be a string, cannot be a number.

Example:

```
test.AddStatistic( "Adjusted MAR", 2.12345 )           ' Output is 2.12345
test.AddStatistic( "My Custom Stat", 2.12345, 3 )     ' Output is 2.12345
test.AddStatistic( "My Custom Stat", 2.12345, "Integer" ) ' Output is 2
test.AddStatistic( "The Best Market", "Gold", "String" ) ' Output is Gold
test.AddStatistic( "The Best Market", "Gold" )       ' Output is Gold
test.AddStatistic( "Return", 2.12345, "Percent" )    ' Output is 2.12345%
test.AddStatistic( "Return", 2.12345, 0, "Percent" ) ' Output is 2.12345%
test.AddStatistic( "Worst Date", 20010521, "Date" )  ' Output is 2001-05-21
test.AddStatistic( "Highest Value", 500000, "Currency" ) ' Output is 500,000.00
```

Links:**See Also:**

CapitalAddsDraws**Syntax:****Parameter:****Description:****Returns:****Example:****Links:****See Also:**

GetSteppedParameter

When a test is being simulated this function will returns all the attributes of each stepped parameter, or when the values of the parameters are zero, it will return the number of stepped parameters to be sequenced.

Syntax:

```
test.GetSteppedParameter( paramIndex, paramValueIndex )
```

Parameter:

paramIndex

Description:

ParamIndex is based on the stepping priority.

paramValueIndex

paramValueIndex is shown below.

Returns:

When values greater than 0 are used in each of the parameter locations, the return value will be the attributes of each parameter.

When 0 is used in both of the parameter fields, the number of stepped parameters in the test is the returned information.

Example:

```
steppedParameterCount = test.GetSteppedParameter( 0, 0 )
```

```
PRINT "Using the following stepped parameters."
```

```
PRINT "Name", _
      "Step Start", _
      "Step End", _
      "Step Step", _
      "Step Count", _
      "Step Index", _
      "Step Value", _
      "Step Priority"
```

```
FOR i = 1 to steppedParameterCount
  ' Stepped parameter values
  StepName = test.GetSteppedParameter( i, 1 )
  stepStart = test.GetSteppedParameter( i, 2 )
  stepEnd = test.GetSteppedParameter( i, 3 )
  stepStep = test.GetSteppedParameter( i, 4 )
  stepCount = test.GetSteppedParameter( i, 5 )
  stepIndex = test.GetSteppedParameter( i, 6 )
  stepValue = test.GetSteppedParameter( i, 7 )
  stepPriority = test.GetSteppedParameter( i, 8 )
```

```
PRINT stepName, _
      stepStart, _
      stepEnd, _
      stepStep, _
      stepCount, _
```



```
stepIndex, _  
stepValue, _  
stepPriority
```

[NEXT](#)

Links:

See Also:

SetAlternateSystem

Companion function to the software's System Object. Its purpose is to allow access to scripts and values in systems other than the system in which this function is being executed. Most often this is used in a Suite with a GSS system of the same name as the Suite name, but it can be used from a different system. In use it is designed to provide access to other systems in the same Suite when the context of that system where the information is located is not the system where the information is needed.

When this function is used it brings the system identified by the system-index value into context so the system executing this function can access information. WhenOnce in context any of the functions or properties that are available from within a system are also made available. Names used are identical to those listed in the System Object.

Global Suite System (GSS) are where this function is handy so as to allow information to pass between the system running in the GSS container.

Syntax:

```
test.SetAlternateSystem( systemIndex )
```

Parameter:

systemIndex

Description:

System index number. When this function is executed it sets the special built-in object "alternateSystem" with a new system by index.

System index number is assigned by the order in which systems are added to a Suite. Index numbers for system begins at zero, which is reserved for the Suite's GSS system, and the index values assigned end at the number value that represents the number of non-GSS systems in the Suite.

Trading Blox allows a specific number of scripts for use in a GSS module. To see which scripts are available, and to understand the order of when those scripts will execute review the [Global Script Timing Table](#).

Notes:

Sets the special built-in object "alternateBroker" with a new system by index. The [alternateBroker](#) object can then be used to place orders for other systems.

Example:

```
' Loop over the systems in the test.
FOR systemindex = 1 TO test.systemCount

  ' Set the alternate system by index.
  test.SetAlternateSystem( systemIndex )

  ' Print each system name and available equity
  PRINT systemIndex, alternateSystem.name, alternateSystem.totalEquity
NEXT
```

Example of setting the alternate system, and using the alternateBroker object:

```
IF inst.LoadSymbol( "F:GC", 1 ) THEN

    test.SetAlternateSystem( 1 )

    IF inst.isPrimed AND inst.position = OUT THEN
        alternateBroker.EnterLongOnopen( inst.symbol )

        IF alternateSystem.OrderExists() THEN
            order.SetQuantity( 10 )
        ENDIF
    ENDIF
ELSE
    PRINT "Unable to load symbol"
ENDIF
```

Links:

[System Object](#)

See Also:

[Global Script Timing Table](#)

SetAutoPriming

This function is used to enable or disable a test simulation and it must be called in the **Before Simulation** script.

By default Trading Blox sets this property to **TRUE** so that test being run will use the software's ability to automatically reserve enough data with each of the instrument files to allow period bar lengths to perform calculations without causing an error.

When set to **FALSE**, **Auto-Priming** will not reserve any priming records. This means that blox scripting must determine when there is enough data available to process calculations without the look back reach of some calculation reaching back past the first available record in a series.

Priming of calculated indicators in the **Indicator** section of the **Trading Blox Editor** need to reference look back values to prevent testing errors:

The screenshot shows the 'New Parameter' dialog box with the following details:

- Name for Code:** AnyBarLookback
- Name for Humans:** Series Calculation Length Example:
- Parameter Type:** Integer - whole number values e.g. 1, 400, 5, etc. (selected)
- Default Value:** 126
- Scope:** Block
- Used for Lookback:** (highlighted with a red box)
- Stepping Enabled:**
- Stepping Priority:** 0

When a Parameter's "**Used for Lookback**" option is enabled, Trading Blox will reserve records for priming so the calculated indicator in the Indicator section doesn't cause an error. When more than one parameter enables its "**Use for Lookback**" option, the sum of the look backs is the amount of instrument records that are reserved. This delay can be seen in the charting display by looking at the bar number where the indicators first appear on the chart.

Syntax:

```
test.SetAutoPriming( TrueFalse )
```

Parameter:

TrueFalse

Description:

By default Auto Priming for parameters is set to TRUE. It can be disabled for special handling when there is no chance that an indicator or calculation will be used when there is insufficient bars available to all the range of the

calculation to perform without error.
<p>Returns:</p> <p>Function doesn't return a value. It just enables or prevents the automatic reservation of look back data bars.</p>
<p>Example:</p> <pre>' BEFORE SIMULATION SCRIPT SECTION ' Disable Auto Priming test.SetAutoPriming(FALSE)</pre>
<p>Links:</p> <p>See Also:</p> <p>Miscellaneous Functions</p>

SetChartSimulationHtml

Function creates end of test tasks to automatically display a custom chart images in the area below the **Stepped Parameter Summary Performance Table**.

`Test.SetChartSimulationHtml` allows the user to insert HTML into the simulation summary chart

Syntax:

```
' Create a task item for the summary report to load
' a custom chart below the plotted stepped parameter chart
' at the top of the Summary Results Report.
test.SetChartSimulationHtml( sHTMLImageReference )
```

Parameter:	Description:
sHTMLImageReference	HTML Image loading reference that includes image path and full file name. See code example below for exact details on how to create a HTML image source reference.

Notes:

This function is placed in the **BEFORE TEST** script section.

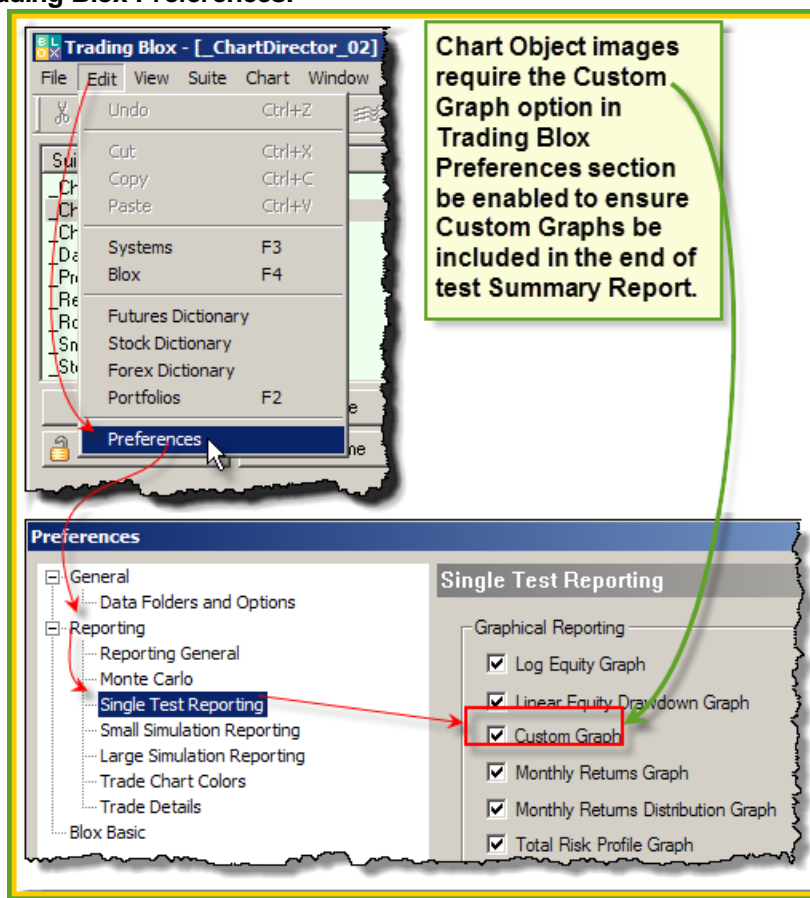
Image width and height assigned to this function should match that the size used to create the chart. If the the space allocated by the HTML statement to too small, some of the displayed image will be blocked. If they are too large, more space around the image will be added creating wasted space.

When used with a multiple stepped test, only one image should be created, so only one image is placed in the report. With thread processing in Trading Blox it will be necessary to and it will be placed in the top section of the Summary Performance Report after the Contour and other simulation scoped graphs.

Where this method differs is in the placement of where this method's charts are placed in the end of test Summary Performance Report. When this method is used, the created chart will be inserted right after the multi-parameter contour and 3D charts, which are created only once for the entire simulation, not for every test step.

In multiple step simulations, the Contour Stats block is a good example of setting the place holder in the Before Simulation script, checking for thread index one so that only one insert is made. As you know, every thread runs the Before Simulation script, so we don't want multiple inserts made.

Trading Blox Preferences:



Preference settings to enable Custom Graphs and Custom Charts.

Display charts just below Multi-Parameter Table in the Stepped Parameter Summary Performance table:

Example:

BEFORE TEST SCRIPT

```

' =====
' This task will load the chart SystemEquity.jpg image into
' the simulation report:
' =====
' This statement creates a single chart displaying task.
test.SetChartSimulationHtml("<img src='" _
                            + test.resultsReportPath _
                            + "\SystemEquity" _
                            + AsString(test.currentParameterTest) _
                            + ".gif" _
                            + "' width=830 height=500>")
' =====
OR
' =====
' This task will load two chart images in the
' simulation report:
' =====
' Next two lines assign the full path and file name to two BPV
' variables:
chartHtml1 = "<img src='" _
            + test.resultsReportPath _
            + "\Winning Trades" _
            + AsString( test.currentParameterTest ) _
            + ".gif" _
            + "' width=415 height=400>"

chartHtml2 = "<img src='" _
            + test.resultsReportPath _
            + "\Losing Trades" _
            + AsString( test.currentParameterTest ) _
            + ".gif" _
            + "' width=415 height=400>"

' This statement creates a task to display two charts
' side by side.
test.SetChartSimulationHtml( chartHtml1 + chartHtml2 )
' =====
OR
' =====
' This task will load the same two chart images defined above
' simulation report, but it will place the first image above
' the second image:
' =====

' This statement creates a task to display two charts
' one above the other.
test.SetChartSimulationHtml( chartHtml1 + "<br>" + chartHtml2 )
' =====

```

Links:

[currentParameterTest](#), [resultsReportPath](#)

SetChartTestHtml

Function creates end of test tasks to automatically display a custom chart images in the area below the **BPV Custom Graphs** are displayed.

Syntax:

```
test.SetChartTestHtml( sHTMLImageReference )
```

Parameter:

sHTMLImageReference

Data Information:

HTML Image loading reference that includes image path and full file name.

Multiple task can be included in as a single parameter by placing a plus-sign + between each image to be loaded.

See code example below for exact details on how to create a HTML image source reference.

Note:

The width and height specified should match that used to create the chart. If the the space allocated by the HTML statement to too small, some of the displayed image will be blocked. If they are too large, more space around the image will be added creating wasted space.

When this method is used with stepped test, the parameter test index is added to the file name created. File names created will each have an index that matches equal the number of steps in test when this method is used with multiple stepped test. When only 1-step is in a test, there will only be one image. When there are more steps in a test, each test-step will have an image and that image will have that step's index value as part of its file name.

Inserts HTML references that include a Custom Chart created graph image into the bottom area where Trading Blox BPV series Custom Charts. When Chart Object images are created they are saved into the Results folder with used to population of end of test Summary Performance Report images.

Trading Blox reporting preferences for the level of reporting intended must show the reporting option selected has the Custom Graph option enabled with a checkmark.

Trading Blox Preferences:

Chart Object images require the Custom Graph option in Trading Blox Preferences section be enabled to ensure Custom Graphs be included in the end of test Summary Report.

Preference settings to enable Custom Graphs and Custom Charts.

Display charts just below Multi-Parameter Table in the Stepped Parameter Summary Performance table.

Example:

BEFORE TEST SCRIPT

```

' =====
' This task will load the chart SystemEquity.jpg image into
' the simulation report:
' ~~~~~
' This statement creates a single chart displaying task.
test.SetChartSimulationtHtml("<img src=' " _
+ test.resultsReportPath _
+ "\SystemEquity" _
+ AsString(test.currentParameterTest) _
+ ".gif" _
+ "' width=830 height=500>")
' =====

```

OR

```

' =====
' This task will load two chart images in the
' simulaition report:
' ~~~~~
' Next two lines assign the full path and file name to two BPV
' variables:
chartHtml1 = "<img src='" _
+ test.resultsReportPath _
+ "\Winning Trades" _
+ AsString( test.currentParameterTest ) _
+ ".gif" _
+ "' width=415 height=400>"

chartHtml2 = "<img src='" _
+ test.resultsReportPath _
+ "\Losing Trades" _
+ AsString( test.currentParameterTest ) _
+ ".gif" _
+ "' width=415 height=400>"

' This statement creates a task to display two charts
' side by side.
test.SetChartSimulationtHtml( chartHtml1 + chartHtml2 )
' =====

```

OR

```

' =====
' This task will load the same two chart images defined above
' simulation report, but it will place the first image above
' the second image:
' ~~~~~

' This statement creates a task to display two charts
' one above the other.
test.SetChartSimulationtHtml( chartHtml1 + "<br>" + chartHtml2 )
' =====

```

Links:

[AsString](#), [currentParameterTest](#), [resultsReportPath](#)

SetGeneratingOrders

Regardless of whether the test was run using Run Simulation or Generate Orders, you can set whether the test will generate orders or not. Set to true to generate orders for this test, set to false to run a normal backtest. By default this is set by which button was pressed to invoke the simulation.

Generally this function is used in the Before Test script.

Useful if an order generation run requires a couple of stepped test runs prior to the order generation run.

Syntax:**Parameter:****Description:****Returns:****Example:****Links:****See Also:**

SetSilentTestRun

Continues the test, but sets the test to silent, so that no test results are displayed in the summary report.

Test results can be filtered in the after test script using the [abortTest](#) or [SetSilentTestRun](#) functions.

Syntax:

```
test.SetSilentTestRun( true/false )
```

Parameter:

```
true/false
```

Description:

Use True or False keywords, or the values, 1 = True, 0=False

Returns:

No Return

Example:

```
IF test.totalTrades < tradesThreshold THEN

    test.SetSilentTestRun( true )

    PRINT "Filtered test ", test.currentParameterTest, _
        " because total trades of ", test.totalTrades, _
        " were less than threshold of ", tradesThreshold

ENDIF
```

Links:**See Also:**

UpdateOtherExpenses

Adjusts the Other Expenses category by the specified amount. This can be used to account for fees or taxes. This amount can be accessed using `test.otherExpenses` property and will print on the Summary Report as "Other Expenses"

This function immediately moves the indicated equity from closed equity to Other Expenses.

Syntax:

```
test.UpdateOtherExpenses( expenseAdjustment )
```

Parameter:

expenseAdjustment

Description:

Amount to add, or subtract from the other expenses category

Returns:

See example comments.

Example:

```
' Moves one percent of the total equity from closed equity
' to other expenses. This amount is no longer available equity
' to the test.
otherExpenseAdjustment = test.totalEquity * .01
test.UpdateOtherExpenses( otherExpenseAdjustment )

' Adds $100,000 to the test closed equity. Subtracts from
' the other expenses.
test.UpdateOtherExpenses( -100000 )
```

Links:

See Also:

11.5 Test Statistics

These are the test level statistics. They are designed to be used in the After Test script. They match the summary statistics printed by Trading Blox in the summary report.

They can also be used for export, other calculations, or with the [AddStatistic](#) function to have them in the sortable results list.

Note:

Some of these statistics calculate the value when used, so use with care as this could be a performance issue.

Statistic Name:	Description:
goodness	the statistic as set in the goodness preferences
expectationRatio	the expectation ratio
annualGeometricReturn	the annual geometric return
dailyGeometricReturn	the daily geometric return using calendar days
marRatio	the mar ratio
averageDailyReturn	the average daily return
averageMonthlyReturn	the average monthly return
averageAnnualReturn	the average annual return
dailyReturnStandardDeviation	the standard deviation of the daily returns
monthlyReturnStandardDeviation	the standard deviation of the monthly returns
annualReturnStandardDeviation	the standard deviation of the annual returns
modifiedSharpeRatio	the modified sharpe ratio
dailySharpeRatio	the daily sharpe ratio
dailyGeoSharpeRatio	the daily geometric sharpe ratio
monthlySharpeRatio	the monthly sharpe ratio
annualSharpeRatio	the annual sharpe ratio
annualizedMonthlySharpeRatio	the annualized monthly sharpe ratio
dailyReturnDownsideDeviation	the downside standard deviation of the daily return
monthlyReturnDownsideDeviation	the downside standard deviation of the monthly return
annualReturnDownsideDeviation	the downside standard deviation of the annual return
dailySortinoRatio	the daily sortino ratio
monthlySortinoRatio	the monthly sortino ratio
annualSortinoRatio	the annual sortino ratio
calmarRatio	the calmar ratio
rSquared	the R Squared
rar	the RAR
rCubed	the R Cubed
averageMaxDrawdown	the average maximum drawdown

averageLongestDrawdown	the average longest drawdown
robustSharpe	the robust sharpe
maxClosedDrawdown	the max closed equity drawdown percent
maxOpenDrawdown	the max total equity drawdown percent
maxOpenMonthlyDrawdown	the max total equity monthly drawdown percent
maxClosedMonthlyDrawdown	the max closed equity monthly drawdown percent
longestOpenDrawdownMonths	the longest total equity drawdown in months
averageClosedDrawdown	the average closed equity drawdown percent
averageOpenDrawdown	the average total equity drawdown percent
closedDrawdownStandardDeviation	the standard deviation of the closed equity drawdown percent
openDrawdownStandardDeviation	the standard deviation of the total equity drawdown percent
netProfit	the net profit
totalWinDollars	the total dollars from winning trades
totalLossDollars	the total dollars from losing trades
profitFactor	the profit factor (win/loss)
averageRiskPercent	the average percent risk per trade
averageWinPercent	the average percent win of the winning trades
averageLossPercent	the average percent loss of the losing trade
averageTradePercent	the average percent profit per trade
percentProfitFactor	the profit factor percent (win percent / loss percent)
totalTrades	the total trades not including zero size trades
winCount	the total winning trades, including break even
lossCount	the total losing trades
winningMonths	the total number of winning months
losingMonths	the total number of losing months
monthCount	the total number of months
roundTurnCount	the number of round turns
earnedInterest	the total earned interest
marginInterest	the total margin interest
totalSlippage	the total slippage
totalCommissions	the total commission
totalCarry	the total cost of carry

monteCarloConfidenceReturn	
monteCarloConfidenceMAR	
monteCarloConfidenceSharpe	
monteCarloConfidenceRSquared	
monteCarloConfidenceDrawdown	
monteCarloConfidenceDrawdown2	
monteCarloConfidenceDrawdown3	
monteCarloConfidenceDrawdownLength	
monteCarloConfidenceDrawdownLength2	
monteCarloConfidenceDrawdownLength3	

11.6 Trade Properties

Test level closed trade details for all the instruments in all of the systems in the test suite.

Trades from each of the systems can be reported with the test property "`tradeSystem`" trade details can be associated with each system in a suite.

Properties:	Description:
<code>averageTradeDuration</code>	Average trade duration of all trades in the test, computed using the <code>tradeDaysInTrade</code> property
<code>savedTradeCount</code>	Number of trades saved by the WF process from one OOS test to the next.
<code>test.tradeCommission[]</code>	
<code>tradeBarsInTrade[]</code>	Number of bars between entry and exit
<code>tradeCommission[]</code>	
<code>tradeCount</code>	Number of prior trades including zero size trades. Used to index the following properties:
<code>tradeCustomValue[]</code>	Custom value as set through scripting
<code>tradeDaysInTrade[]</code>	Number of days between entry and exit (includes weekends and holidays)
<code>tradeDirection[]</code>	Direction as a description of LONG or SHORT text.
<code>tradeDollarsPerPoint[]</code>	Dollars per point on the entry day
<code>tradeEntryBPV[]</code>	Entry BPV of the instrument
<code>tradeEntryDate[]</code>	Entry date
<code>tradeEntryFill[]</code>	Entry fill price
<code>tradeEntryOrder[]</code>	Entry order price
<code>tradeEntryRisk[]</code>	Entry risk as a percent of entry day trading equity
<code>tradeEntryStop[]</code>	Initial entry day stop, if used
<code>tradeEntryTime[]</code>	Entry time
<code>tradeExitDate[]</code>	Exit date
<code>tradeExitFill[]</code>	Exit fill price
<code>tradeExitOrder[]</code>	Exit order price
<code>tradeExitTime[]</code>	Exit time
<code>tradeMaxAdverseExcursion[]</code>	Maximum Adverse excursion of the trade
<code>tradeMaxFavorableExcursion[]</code>	Maximum Favorable excursion of the trade
<code>tradeMinFavorableExcursion[]</code>	Minimum Favorable excursion of the trade
<code>tradePositionReferenceID[]</code>	Position Reference value.
<code>tradeProfit[]</code>	Closed out profit including Slippage and Commission
<code>tradeProfitPercent[]</code>	Profit as a percent of entry day trading equity
<code>tradeQuantity[]</code>	Quantity in shares or contracts
<code>tradeRuleLabel[]</code>	Rule label as set through scripting

tradeSymbol[]	Symbol for the instrument for this trade
tradeSystem[]	System index for the trade
tradeUnitNumber[]	Unit number for the trade

Trade Indexing:

Properties listed with a '[]' following them require an indexed using a number which determines which trade is reported. When the value of the index is 1, the trade reported will be latest, or most recent trade will be reported first. When the index value is stepped beginning at the value of the number of the trades, then the oldest trade detail will be reported first.

Most Recent Trade Reported First:**Example:**

```

' Report Sytem & Trade count data
PRINT "-----"
PRINT "testStart", ",", "test.testStart"
PRINT "testEnd", ",", "test.testEnd"
PRINT "systemCount", ",", "test.systemCount"
PRINT "totalTrades", ",", "test.totalTrades"
PRINT
' Create Column Header Titles
PRINT "Trade#", ",", "System#", ",", "Symbol", ",", "Date"

' Loop through all the trades generated
' by the systems in the suite
For x = 1 TO test.totalTrades STEP 1
' Report Trade#, System#, and trade symbol
PRINT x, ",", "_
      test.tradeSystem[x], ",", "_
      test.tradeSymbol[x], ",", "_
      test.tradeEntryDate[x]
Next ' x

```

Returns:

```

testStart 1/3/2000
testEnd 12/31/2002
systemCount 3
totalTrades 1303

Trade#    System#    Symbol    Date
1         3         HG2      12/18/2002
2         3         MP       12/30/2002
[SNIP]
51        1         EM       5/14/2002
52        1         ED       12/23/2002
53        1         SF       12/23/2002
54        3         NG2     12/6/2002
55        1         HO2     12/30/2002
56        3         CD      12/2/2002
57        2         CD      11/29/2002
58        2         S2     12/23/2002
59        1         HG2     11/25/2002
60        2         MP      12/19/2002
61        3         SB2     12/2/2002
[SNIP]

```

Oldest or first Trade Execute shown First:**Example:**

```

' Report System & Trade count data
PRINT "-----"
PRINT "testStart", " ", "test.testStart"
PRINT "testEnd", " ", "test.testEnd"
PRINT "systemCount", " ", "test.systemCount"
PRINT "totalTrades", " ", "test.totalTrades"
PRINT
' Create Column Header Titles
PRINT "Trade#", " ", "System#", " ", "Symbol", " ", "Date"

' Loop through all the trades generated
' by the systems in the suite
For x = test.totalTrades TO 1 STEP -1
' Report Trade#, System#, and trade symbol
PRINT x, " ", "_
      test.tradeSystem[x], " ", "_
      test.tradeSymbol[x], " ", "_
      test.tradeEntryDate[x]
Next ' x

```

Returns:

```

testStart 2000-01-03
testEnd   2002-12-31
systemCount 3
totalTrades 1303

```

Trade#	System#	Symbol	Date
1303	2	LC	2000-01-05
1302	2	CT2	2000-01-03
1301	2	RB2	2000-01-06
[SNIP]			
1294	3	FC	2000-01-06
1293	2	LC	2000-01-07
1292	2	FC	2000-01-03
1291	3	EM	2000-01-03
1290	1	HO2	2000-01-24
1289	2	SI2	2000-01-12
1288	3	HG2	2000-01-04
1287	3	US	2000-01-04
1286	2	BP	2000-01-20
1285	2	AD	2000-01-03
[SNIP]			

Common Questions

Part



Part 6 – Common Questions

Visit the Documentations page on the website for current PDF, Help File, or Web Based versions of these manuals.

[Documentation Webpage](#)

Section 1 – The Life of a Test

1) Test Start

The Start date of the test is when the blox scripts will start running, and if the indicators are primed the instruments will start trading.

2) Test End Date

The End date of the test is when the system will close all open positions and calculate the summary results for the test.

If you are generating orders, the open positions will not be closed but will be displayed on the Order Generation Report. Orders will be generated for the day after the test end date.

3) Indicator and Lookback Priming

On the Start Date, if you have enough data, all your instrument indicators will be primed and ready for use. If you don't have enough data, such as when you set the start date near or even before the beginning of the data, then each instrument will start trading when it is primed.

The priming for each indicator is determined by the overall prime bars required for the system. The prime bars is determined by adding the max required bars for indicator priming to the max required bars for lookback parameters.

So if you have a simple moving average indicator that uses a parameter with value 20, the indicator priming bars will be 20. If in addition you have another lookback type parameter with value 10, then the total priming required will be 30 bars.

Here is an example, where `simpleMovingAverage` is defined as a 20 day moving average, and `lookbackParameter` is a lookback parameter with value 10:

```
simpleMovingAverage[ lookbackParameter ]
```

Section 2 – How Stops Work

There's a few different things to know about how stops work in Trading Blox.

How do I enter an order with a stop?

Use a broker order like the following:

- `broker.EnterLongOnOpen(exitStop)` - Buy on the open with an optional stop price
- `broker.EnterShortOnOpen(exitStop)` - Sell on the open with an optional stop price
- `broker.EnterLongOnStop(entryStop, exitStop)` - Buy on a stop with an optional exit stop price
- `broker.EnterShortOnStop(entryStop, exitStop)` - Sell on a stop with an optional exit stop price
- `broker.EnterLongAtLimit(entryLimit, exitStop)` - Buy at the limit with an optional stop price
- `broker.EnterShortAtLimit(entryLimit, exitStop)` - Sell at the given limit with an optional stop price

The `entryStop` or `entryLimit` (where applicable) is the basic stop or limit order. The `exitStop`, which is available to all these orders, is a protective stop. Let's say the broker enters long at \$11 with an `exitStop` of \$10. If the prices drops below \$10 on the day of entry, the position will automatically be exited. But check the Entry Day Retracement Percent for details on whether an entry day stop will be exited based on the low or the close of the day. Note that stops placed in this manner are for the day of entry only. Be sure to place the stops in the Exit Script each day to hold the stop for the duration of the trade.

Do I have to set stops?

No, `exitStop` is optional for all broker orders. Some reversal systems are always in the market and don't use stops. However, remember that risk calculations are done using these stops. If you have no stops, Trading Blox assumes undefined risk. Using Blocks like the Fixed Fractional Money Manager, which is calculated using the `entryRisk`, will result in no trades.

I set a protective stop for my order and the price went below it on the next bar. My position was not exited.

The protective stop as placed with an entry order is saved with the instrument, but only placed for the day of entry. See the following question.

Can I keep my protective stop "active" during the duration of my position?

Yes! The protective stop is stored as `instrument.unitExitStop`. You must enter a broker order every day to "hold" this stop. You could use an order like the following:

```
broker.ExitAllUnitsOnStop( instrument.unitExitStop )
```

Many of our built-in systems use this method of "holding" stops.

Can I change my stop once it is set?

The function `instrument.setExitStop()` will set a new stop.

`instrument.SetExitStop(unitNumber, stopPrice)` - Sets a new stop price for a particular unit. This value is used to calculate risk at the end of the day. If you just have one unit on, you do not need to enter a `unitNumber`.

```
instrument.SetExitStop( newStopPrice )
```

```
broker.ExitAllUnitsOnStop( instrument.unitExitStop )
```

What script should I change my stops in?

The "Adjust Stops" script is the best place. The stops are then used in the days risk and other calculations.

We usually place all entry broker orders in the entry script, and exit broker orders in the exit script. But you can 'set' the stop in the Adjust Stops script.

What if I have multiple units?

Use the same function - you can enter a different stop for each unit:

```
instrument.SetExitStop( unitNumber, stopPrice )
```


Section 3 – Shortcut Keys

Trading Blox Main Screen Shortcut Keys:

Keys	Operation
F2	Display Active Portfolio
F3	Open System Editor
F4	Open Code Editor (Builder Version Only)
F5	Execute Simulation Test
F7	Execute Positions and Orders Report

Note:

Laptop keyboards often require the user use the keyboards function key (Fn) to enable the F-key action listed in the table.

Trading Blox Builder Editor & Integrated Debugger Shortcut Keys:

Keys	Use-In	Operation:
Control + S	Editor	Save
Control + A	Editor	Select All
Control + C	Editor	Copy Selected Text
Control + X	Editor	Cut Selected Text
Control + Z	Editor	Undo (multiple levels supported)
Control + Y	Editor	Redo
Control + V	Editor	Paste copied or cut text
Control + F	Editor	Find
Control + G	Editor	Find Again (same direction)
Shift + Control + G	Editor	Find Again (reverse direction)
Control + H	Editor	Replace
Escape		Exit the editor. Prompts to save if changes were made
F5	Debug	Run to the next Breakpoint
F9	Both	Toggle a breakpoint on the current line (In Code Editor Builder Version Only)
Shift + F9	Both	Clear all breakpoints
F10	Debug	
F11	Debug	Step through each line of code

Debugger - Add-Items :

1. Ability to Watch variables in the debugger. Double click or use F8 to add to watch screen.
2. Debugger now shows the current value and the past four indexed values for series variables.

3. Debugger can now STEP Into other functions and scripts. Use F11 to enable STEP into, use F10 to disable STEP into.
 4. Use F11 to start a test in debug mode.
-

Index

- A -

AbortParameterRun 409
 AbortTest 409
 activeStatus 548
 addLine 409
 addLineSeries 409
 Auto-Priming 700
 averageVolume 548

- B -

bar 548
 barsSinceEntry 409
 Basic Money Manager 52
 bigPointValue 548
 Block Object 216
 Blox
 Working with 72
 Boolean 144
 Breakpoint 407
 Broker
 Entry Orders 429
 Exit Orders 450
 brokerSymbol 548

- C -

close 548
 Comments 165
 Commodity Channel Index 298
 Common Questions 717
 How Stops Work 719
 Shortcut Keys 721
 The Start and End Dates 718

Comparison 348
 Constants 142
 conversionRate 548
 Correlation Functions 558
 Add Closely Correlated 559
 Add Loosely Correlated 559
 Reset Closely Correlated 558
 Reset Loosely Correlated 558
 Correlation Properties 560
 currency 548
 currencyBorrowRate 548
 currencyDate 548
 currencyLendRate 548
 currencyTime 548
 currentBar 548
 CurrentDate 190
 currentDay 157, 409, 680
 currentDrawdown 409
 currentOpenEquity 409
 currentParameterRun 409
 currentParameterTest 409
 currentPositionProfit 409
 currentPositionQuantity 409
 currentPositionRisk 409
 currentPositionUnits 409
 CurrentTime 191
 currentWeek 548

- D -

Data Access Properties 548
 Data Function
 Add Commission 553
 GetDateTimeIndex 554
 GetDayIndex 555
 Price Format 556
 Real Price 556
 Round Tick 557
 Round Tick Down 557

Data Function
 Round Tick Up 557

Data Functions 553

dataLoadedBars 409, 548

dataVendorID 548

date 548

Date Functions 176

- DateToJulian 179
- DayMonthYearToDate 180
- DayOfMonth 181
- DayOfWeek 182
- DayOfWeekName 183
- DaysInMonth 184
- JulianToDate 186
- Month 188
- MonthName 189
- SystemDate 190
- SystemTime 191
- WeekNumberISO 193
- Year 195

Day 48

dayClose 548

dayHigh 548

dayIndex 548

dayLow 548

dayNumber 409

dayOpen 548

dayVolume 548

Debugger 407

defaultAverageTrueRange 548

deliveryMonth 409, 548

deliveryMonthLetter 548

Dema 298

description 548

displayDigits 548

dividend 548

Dominant Cycle 298

Dominant Cycle Highest 298

Dominant Cycle Lowest 298

Dominant Cycle Phase 298

- E -

Ehlers Lead Sinewave 298

Ehlers Nonlinear Ma 298

Ehlers Sinewave 298

Ehlers Zero Lag Ema 298

Email Manager

- EmailConnect 518
- EmailConnectSSL 519
- EmailDisconnect 524
- EmailSend 521
- EmailSendHTML 522

endBar 548

endDate 548

EnterLongOnStopOpen 409

EnterLongStopOpenOnly 409

EnterShortOnStopOpen 409

EnterShortStopOpenOnly 409

Entry Orders

- EnterLongAtLimit 441
- EnterLongAtLimitClose 448
- EnterLongAtLimitOpen 434
- EnterLongOnClose 443
- EnterLongOnOpen 431
- EnterLongOnStop 439
- EnterLongOnStopClose 446
- EnterLongOnStopOpen 433
- EnterShortAtLimit 442
- EnterShortAtLimitClose 449
- EnterShortAtLimitOpen 437
- EnterShortOnClose 444
- EnterShortOnOpen 432
- EnterShortOnStop 440
- EnterShortOnStopClose 447
- EnterShortOnStopOpen 435

Entry Scripts

- Entry Order Filled 121
- Entry Orders 114

equityDrawdown 409

exchange 548

Exit Orders

ExitAllUnitsAtLimit 460
 ExitAllUnitsAtLimitClose 466
 ExitAllUnitsAtLimitOpen 455
 ExitAllUnitsOnClose 462
 ExitAllUnitsOnOpen 452
 ExitAllUnitsOnStop 458
 ExitAllUnitsOnStopClose 464
 ExitAllUnitsOnStopOpen 454
 ExitUnitAtLimit 461
 ExitUnitAtLimitClose 467
 ExitUnitAtLimitOpen 457
 ExitUnitOnClose 463
 ExitUnitOnOpen 453
 ExitUnitOnStop 459
 ExitUnitOnStopClose 465
 ExitUnitOnStopOpen 456

Exit Scripts

Exit Order Filled 120
 Exit Orders 113

extraData1 548

extraData8 548

- F -

FALSE 144, 700

FAMA 298

File Functions

CopyFile 198
 CreateDirectory 199
 DeleteFile 199
 EditFile 200
 FileExists 200
 MoveFile 201
 OpenFile 201
 OpenFileDialog 202
 SaveFileDialog 203

FileManager

Close 528
 EndOfFile 531
 OpenAppend 532
 OpenRead 533

OpenWrite 535

ReadLine 538

WriteLine 540

WriteString 542

fileName 548

firstDataLoadedDate 548

Fixed Fractional Money Manager 52

Floating 144

folder 548

forexBaseBorrowRate 548

forexBaseLendRate 548

forexPipSize 548

forexPipSpread 548

forexQuoteBorrowRate 548

forexQuoteLendRate 548

futuresMonth 409

- G -

General Functions

BuildDividendFiles 205
 ColorBackground 223
 ColorCrossHair 223
 ColorCustom1 223
 ColorCustom2 223
 ColorCustom3 223
 ColorCustom4 223
 ColorDownBar 223
 ColorDownCandle 223
 ColorGrid 223
 ColorLongTrade 223
 ColorShortTrade 223
 ColorTradeEntry 223
 ColorTradeExit 223
 ColorTradeStop 223
 ColorUpBar 223
 ColorUpCandle 223
 FileVersion 205
 FileVersionNumerical 205
 GetRegistryKey 205
 License Name 214
 LicenseName 205

General Functions

LineNumber 205
 LoadUnadjustedClose 223
 LoadVolume 223
 Message Box 217
 MessageBox 205
 NumberOfExtraDataFields 223
 PlaySound 205, 221
 Preference Items 223
 ProcessDailyBars 223
 ProcessMonthlyBars 223
 ProcessWeekends 223
 ProcessWeeklyBars 223
 ProductVersion 205
 ProductVersionNumerical 205
 RaiseNegativeDataSeries 223
 SetRegistryKey 205
 YearsOfPrimingData 223

generatingOrders 409

Getting Started 2

Adding Money Management 52
 Creating a System 4
 Tutorial 4, 52
 What are blox? 3

Group Properties 562

GSS 698

- H -

high 548

Historic Volatility 298

Historical Trade Properties 564

- I -

Indicators 286

Accessing 295
 Basic 286
 Calculated 292
 Custom 294
 Invalid Items 292
 Valid Items 292

inPortfolio 548

Instantaneous Trendline 298

Instantaneous Trendline Alternate 298

Instrument 144

Instrument Loading 566

Load By Long Rank 569

Load By Short Rank 570

Load External Data 570

Load IPV From File 572

Load Symbol 567

instrumentCount 409

Integer 144

intradayData 548

isForex 548

isFuture 548

isPrimed 548

isStock 548

- J -

julianDate 548

- K -

Kaufman Adaptive Moving Average 298

Keywords

Abort 399, 691, 692

Assert 399

Break 399

DO 397

ELSE 402

End 691, 692

ENDIF 402

ENDWHILE 405

FOR 400

IF 402

LOOP 397

NEXT 400

Stop 399, 691, 692

THEN 402

UNTIL 397

Keywords

WHILE 397, 405

- L -

Laguerre Moving Average 298

lastBarOfDay 548

lastDataLoadedDate 548

lastDayOfMonth 548

lastDayOfWeek 548

lastDayOfYear 548

lastTradingInstrument 548

LoadPortfolioInstrument 409

LoadSymbol 409

low 548

- M -

MAMA 298

margin 548

Mathematical Comparison 348

Mathematical Functions 227

Absolute Value 228

Arc Cosine 230

Arc Sine 230

Arc Tangent 231

Arc Tangent XY 231

Average 231

CAGR 232

Correlation 234

Correlation Log 234

Cosine 235

Degrees to Radians 235

ema function 236

Exponents 237

Hypotenuse 239

IfThenElse 239

IsUndefined 240

Log 241

Max 241

Min 241, 244

Radians to Degrees 242

Random 242

RandomDouble 243

RandomSeed 243

Round 244

Sine 247

Square Root 247

Standard Deviation 247

Standard Deviation Log 248

sumValues 248

Tangent 249

minimumTick 548

minimumVolume 548

Momentum 298

Money 144

Money Manager Scripts

Unit Size 115

monthClose 548

monthHigh 548

monthIndex 548

monthLow 548

monthOpen 548

Multi-Money Manager 52

- N -

nativeBPV 548

negativeAdjustment 548

- O -

Objects 413

alternateBroker 425

AlternateOrder 592

alternateSystem 664, 698

Block 418

Broker 425

Email Manager 517

FileManager 525

Instrument 544

Name 418

Objects 413
 Order 592
 Script 643
 ScriptName 418
 System 664
 Test 677
 On Balance Volume 298
 On-Stop 48
 open 548
 openEquity 409
 openInterest 548
 Operators 347
 Order Functions
 Reject 623
 SetClearingIntent 622
 SetCustomValue 626
 SetFillPrice 628
 SetLimitPrice 622
 SetOrderReportMessage 630
 SetQuantity 632
 SetRuleLabel 634
 SetSortValue 636
 SetStopPrice 640
 SetTimelInForce 622
 Order Properties 596
 orderGenerationBar 409
 Orders
 Entry Orders 429
 Exit Orders 450
 orderSortValue 548
 OtherExpense 678

- P -

Parameters 349
 Percent 144
 Percent R 298
 Percent Rank 298
 Placing Orders 425
 Portfolio Manager Scripts

Filter Portfolio 109
 Rank Instruments 108
 Position Functions 575
 Set Exit Limit 576
 Set Exit Stop 576
 Set Unit Custom Value 575
 Position Properties 578
 positionInstruments 409
 Positon Adjustment Functions 468
 Adjust Position At Limit 471
 Adjust Position On Close 468
 Adjust Position On Open 469
 Adjust Position On Stop 470
 Price 144
 priorityIndex 409, 548

- R -

Range 298
 Ranking Functions 580
 Set Long Ranking Value 581
 Set Short Ranking Value 582
 Ranking Properties 583
 rankInstruments 409
 Rate of Change 298
 Registry Keys
 GetRegistryKey 212
 SetRegistryKey 225
 resultsReportPath 680
 Risk Manager Scripts
 Adjust Instrument Risk 128
 Can Add Unit 116
 Can Fill Order 119
 Compute Instrument Risk 126
 Compute Risk Adjustment 127
 Initialize Risk Management 125
 roundLot 548

- S -

savedWFProfit 548

- Scripts 104
 - After Instrument Day 130
 - After Simulation 133
 - After Test Script 132
 - After Trading Day 131
 - Basic Scripts 78
 - Before Instrument Day 111
 - Before Simulation 106
 - Before Test 107
 - Before Trading Day 110
 - Common to Many Blox 103
 - Entry Blox Scripts 89
 - Exit Blox Scripts 90
 - Money Manager Blox Scripts 91
 - Portfolio Manager Blox Scripts 88
 - Risk Manager Blox Scripts 92
 - Update Indicators Blox Scripts 93
 - Working with 75
 - Selector 144
 - Series 144, 157
 - SetReturnValue 409
 - SetSeriesValues 382
 - SetStringReturnValue 409
 - setxAxisLabels 409
 - Simons Historic Volatility 298
 - Simulation Loop
 - Comprehensive 81
 - sortInstruments 409
 - startBar 548
 - startDate 548
 - StartingEquity 678
 - startWeek 548
 - Statements 395
 - Assignment 396
 - DO 397
 - Error 399
 - FOR 400
 - IF 402
 - Print 404
 - VARIABLES 144
 - WHILE 397, 405
 - stockSplitRatio 548
 - STRING 144, 250, 285
 - String Functions
 - ASCII 252
 - ASCIIToCharacters 253
 - Concatenate 250
 - FindString 254
 - FormatString 255
 - GetField 261
 - GetFieldCount 262
 - GetFieldNumber 263
 - LeftCharacters 265
 - LowerCase 264
 - MiddleCharacters 266
 - RemoveCommasBetweenQuotes 267
 - RemoveNonDigits 269
 - ReplaceString 270
 - RightCharacters 271
 - StringLength 272
 - TrimLeftSpaces 273
 - TrimRightSpaces 274
 - TrimSpaces 275
 - ucase 276
 - symbol 548
 - System
 - Correlation 359
 - Correlation Log 360
 - Portfolio Instrument Access 668
 - System Functions 667
 - System Properties 673
 - systemClosedEquity 548
 - systemOpenEquity 548
 - Systems
 - Working with 69
 - systemTotalEquity 548
- T -
- Tema 298
 - Test
 - CreateStringArray 688
 - Equity Properties 678

- Test
 - General Properties 680
 - GetStringArrayElement 688
 - SetStringArrayElement 688
 - SortStringArray 688
 - String Array Functions 688
 - Test Statistics 710
 - Trade Properties 713
- Test Functions 689
 - Abort Simulation 691
 - Abort Test 692
 - Add Statistic 693
 - GetSteppedParameter 696
 - Set Alternate System 698
 - Set Auto Priming 700
 - Set Generating Orders 707
 - SetSilentTestRun 708
 - Update Other Expenses 709
- testClosedEquity 548
- testOpenEquity 548
- testTotalEquity 409, 548
- time 548
- Time Function
 - Hour 185
 - Minute 187
- Time Functions
 - TimeDiff 192
- totalEquity 409
- totalInstruments 409
- TotalMargin 678
- totalParameterRuns 409
- totalParameterTests 409
- totalPositionProfit 409
- totalPositionRisk 409
- totalPositions 409
- totalPositionSize 409
- totalUnits 409
- Trade Control Functions 585
 - Allow All Trades 588
 - Allow Long Trades 586
 - Allow Short Trades 587
 - Deny All Trades 591
 - Deny Long Trades 589
 - Deny Short Trades 590
- Trade Control Properties 584
- tradeDayOpen 548
- tradeOrder 409
- tradesOnTradeBar 548
- tradesOnTradeDate 548
- Trading Blox 86
 - Auxiliary Blox Reference 93
 - Entry Blox Reference 89
 - Exit Blox Reference 90
 - Money Manager Blox Reference 91
 - Portfolio Manager Blox Reference 88
 - Risk Manager Blox Reference 92
- Trading Blox Architecture
 - Blox 65
 - Indicators 65
 - Parameters 65
 - Process Flow 79
 - Scripts 65
 - Simulation Loop 80
 - Suites 65
 - Systems 65
 - Trading Objects 65
 - Units 65
 - Variables 65
 - Working with Systems, Blox and Scripts 69
- Trading Objects 413
- tradingBars 409
- tradingMonths 548
- Trend Vigor 298
- TRUE 144, 700
- True High 298
- True Low 298
- True Range 298
- TYPE 144, 283
- Type Conversion Functions 277
 - AsFloating 278
 - AsInteger 279

Type Conversion Functions 277

AsString 281
IsFloating 283
IsInteger 284
IsString 285

- U -

unadjustedClose 548
unAdjustedVolume 548
unitBarsSinceEntry 409
unitCustomValue 578
Update Indicators Scripts
 Update Indicators 118
Used for Lookback 700
usedMargin 548

- V -

Variables 168
 Average 358
 Block Permanent Variables 351
 Cross Over 363
 GetSeriesSize 368
 Highest 368
 HighestBar 369
 Instrument Permanent Variables 353
 Lowest 370
 LowestBar 371
 Median 372
 Naming Variables 166
 RegressionEnd 372
 RegressionSlope 373
 RegressionValue 375
 RSI 375
 Scope 163
 Series Functions 357, 358, 363, 368, 369,
 370, 371, 372, 373, 375, 376, 381, 382, 385,
 387, 388, 389, 390, 391, 392, 393
 Series Indexing 365
 SetSeriesColorStyle 376
 SetSeriesSize 381
 SetSeriesValues 382

SortSeries 385
SortSeriesDual 387
Standard Deviation 388
Standard Deviation Log 389
Sum 390
Swing High 390
Swing High Bars 391
Swing Low 392
Swing Low Bars 393
The VARIABLES Statement 144

volume 548

- W -

weekClose 548
weekHigh 548
weekIndex 548
weekLow 548
weekOpen 548
Weighted Moving Average 298

- X -

xAxis 409

- Z -

ZScore 298