

Trading Blox Builder's Guide

© 2024, Trading Blox, LLC. All rights reserved.

Build: 5.4.7



Trading Blox Builder's Guide

By Traders... For Traders

Trading Software for Mechanical Systems Traders

© 2024, Trading Blox, LLC. All rights reserved.

*This user's guide and all its contents are copyright
2003-2024, Trading Blox LLC*

*Trading Blox LLC
www.tradingblox.com
978-772-2620
customersupport@tradingblox.com*

Table of Contents

Foreword	0
Part I Getting Started Tutorial	1
1 What Are Blox?	3
2 Creating a New System	4
1. New System Blox	6
2. Adding Parameters	12
3. Adding Indicator	17
4. Entering Code	20
5. Building A System	23
6. Creating A Suite	28
3 Improving a New System	31
Protective Position Pricing	33
Copy System Items	36
Protective Exit Orders	45
Entry Order Protection	55
Active Order Protection	61
Order Sizing	66
Trading Risk	73
Money Management	77
Part II Trading Blox Architecture	79
1 Working with Systems, Blox & Scripts	84
Working with Systems	86
Working with Blox	90
Blox Name Changes	97
Working with Scripts	104
Basic Scripts	107
Equity Types	109
2 Process Flow	111
3 Simulation Loop	112
4 Comprehensive Simulation Loop	113
Part III Blox Module Reference	118
1 Blox Types	122
Portfolio Manager	124
Entry	126
Exit	127
Money Manager	127
Risk Manager	128
Auxiliary	129
2 Blox Scripts	130
New Block Creation	134
Managing Block Scripts	138
3 Blox Script Timing	139

4 Blox System Placements	143
5 Global Script Timing	145
6 Script Section Reference	148
Before Simulation	150
Set Parameters	152
Set Parameters Methods.....	156
Before Test	160
Rank Instruments	160
Filter Portfolio	162
Before Trading Day	162
Before Instrument Day	162
Before Bar	164
Before Close	165
Exit Orders	165
Entry Orders	166
Can Place Order	168
Unit Size	169
Can Add Unit	169
Filtered Order Notification	171
Before Order Execution	176
Update Indicators	176
Can Fill Order	178
Exit Order Filled	179
Entry Order Filled	180
After Instrument Open	181
After Bar	183
Adjust Stops	183
Initialize Risk Management	184
Compute Instrument Risk	184
Compute Risk Adjustment	184
Adjust Instrument Risk	185
After Instrument Day	187
After Trading Day	187
After Test	187
After Simulation	188
Post Process Utility	189
 Part IV Blox Basic Language Reference	 191
1 Basic Keywords	193
2 Data Variables	194
Colors	195
Constants Reference	200
About Data Variables	202
Data Comments	205
Data Groups	206
Block Permanent Variables.....	208
Series Data Information	216
Instrument Permanent Variables.....	221
Parameters	229
Parameter Control	238
Parameter Value Updates.....	243
Parameter Selector	246

Parameter Stepping	251
Parameter Stepping Priority.....	253
Indicators	259
Local Variables	260
Data Names	262
Data Scope	265
Defined Elsewhere	271
Data Types	273
Boolean	278
Floating	281
Instrument & BPV	282
Integer	285
Percent	286
Price	288
Selector	289
Series	292
String Series	294
Numeric Series	296
Dual Series	298
String	299
3 Function Reference	300
Custom	301
Custom User Functions.....	302
Date & Time	306
ChartTime	307
DateToJulian	309
DayMonthYearToDate.....	310
DayOfMonth	311
DayOfWeek	312
DayOfWeekName	313
DaysInMonth	315
Hour	315
JulianToDate	317
Minute	318
Month	319
MonthName	320
SystemDate	321
SystemTime	322
TimeDiff	324
WeekNumberISO	325
Year	327
File & Disk	328
BuildDividendFiles	330
ClearLogWindow	332
CloseAuxiliaryWindow.....	333
CloseLogWindow	335
CopyFile	336
CreateDirectory	337
DeleteFile	338
EditFile	339
Extract	340
FileDate	341
FileDateTime	342
FileExists	343

FileSize	344
FileTime	345
LoadBPVFromFile	345
LoadIPVFromFile	350
MoveFile	359
OpenAuxiliaryWindow	360
OpenFile	362
OpenFileDialog	363
OpenLogWindow	364
RenameFile	365
SaveFileDialog	366
General	367
ColorRGB	369
Color Selection Dialog	370
FileVersion	374
FileVersionNumerical	375
GetRegistryKey	376
LicenseName	377
LineNumber	378
Message Box	380
PlaySound	385
Preference Items	386
ProductVersion	387
ProductVersionNumerical	388
SetRegistryKey	389
Type	390
Variables	391
Math	393
AbsoluteValue	395
ArcCosine	396
ArcSine	397
ArcTangent	398
ArcTangentXY	399
Average	400
CAGR	402
Ceiling	403
Correlation	405
CorrelationLog	407
Cosine	409
DegreesToRadians	410
EMA	411
Exponent	412
Floor	413
Hypotenuse	415
IfThenElse	416
IsUndefined	417
Log	418
Max	419
Min	420
RadiansToDegrees	422
Random	423
RandomDouble	424
RandomSeed	425
Round	426

RSquared	426
Sign	428
Sine	430
Square Root	431
StandardDeviation	431
StandardDeviationLog	434
SumValues	436
Tangent	438
Series	439
AsDate	441
GCD	443
AsSeries	444
Average	445
CopySeries	447
Correlation	448
CorrelationLog	450
CorrelationLogSynch	452
CorrelationSynch	455
CrossOver	458
Data Series Indexing	460
DownloadWebFile	463
GetSeriesSize	464
Highest	465
HighestBar	467
Lowest	469
LowestBar	471
LowestBar2	473
MaxSynchBars	474
Median	475
RegressionEnd	476
RegressionSlope	478
RegressionValue	480
RSI	482
SetAllowParameterWrite	483
SetAllowPostDictive	483
SetAuxiliaryWindowText	485
SetIBTWSDDataTimeZone	495
SetSeriesAutoIndex	496
SetSeriesColorStyle	497
SetSeriesEnable	508
SetSeriesSize	509
SetSeriesValues	512
SetSeriesValueType	515
SortSeries	516
SortSeriesDual	519
StandardDeviation	522
StandardDeviationLog	524
Sum	526
SwingHigh	528
SwingHighBars	530
SwingLow	532
SwingLowBars	534
SetSeriesNameColor	535
String	537

ASCII	539
ASCII To Characters	541
AsString	543
Character to ASCII to Character	545
EncloseInQuotes	549
FindString	550
FormatString	551
GetField	560
GetFieldCount	562
GetFieldNumber	564
LeftCharacters	565
LowerCase	566
MiddleCharacters	567
RemoveCommasBetweenQuotes	568
RemoveNonDigits	570
ReplaceString	571
RightCharacters	573
StringLength	574
TrimLeftSpaces	575
TrimRightSpaces	576
TrimSpaces	577
UpperCase	578
Type Conversion	579
AsFloating	580
AsInteger	581
AsSeries	583
AsString	584
IsFloating	586
IsInteger	587
IsString	588
4 Indicator Pack 1	589
Indicator Pack 1 Indicators	590
Average Trend Channel	595
Chaiken Money Flow	597
Commodity Channel Index	601
Dominant Cycle	604
Historic Volatility	606
Kaufman Adaptive Moving Average	609
Keltner Channel	613
Klinger Oscillator Indicator	618
On Balance Volume	621
Trend Vigor	623
True Strength Indicator	626
Indicator Pack 1 Series Functions	629
EhlersZeroLagEma	631
InstantaneousTrendLine	633
MarketNoise	634
MedianAbsoluteDeviation	636
Momentum	638
MRO	640
Percentile	643
PercentRank	644
RateOfChange	646
SpearmanCorrelation	648

SpearmanCorrelationSync.....	649
SpearmanLogCorrelation.....	650
SpearmanLogCorrelationSync.....	651
ValueChart	652
WMA - Weighted M-Avg.....	654
Z-Score	656
5 Indicator Reference	658
Basic Indicators	660
Calculated Indicators	663
Creating Indicators	665
Custom Indicators	670
Indicator Access	672
Indicator Bar to Week Time Frame	674
6 Operator Reference	679
Comparison	683
7 Script Problem Information	684
Auto-Keyword Changes	685
Debugger	687
Break Point Editor	691
Key Word Changes	693
Restricted Keywords	695
8 Statement Reference	697
Assignment	698
DO	699
ERROR	702
FOR	703
IF	706
PRINT	708
WHILE	711

Part V Trading Objects Reference 712

1 Alternate Objects	716
AlternateBroker Object	718
AlternateOrder Object	720
AlternateSystem Object	721
2 Block	722
Group	725
Name	727
ScriptName	728
System	729
SystemIndex	730
3 Broker	732
Entry Order Functions	737
EnterLongOnOpen	738
EnterShortOnOpen	740
EnterLongOnStopOpen.....	741
EnterLongAtLimitOpen.....	742
EnterShortOnStopOpen.....	744
EnterShortAtLimitOpen.....	746
EnterLongOnStop	748
EnterShortOnStop	749

EnterLongAtLimit	750
Limit Order Operation	752
EnterShortAtLimit	753
EnterLongOnClose	755
EnterShortOnClose	757
EnterLongOnStopClose.....	758
EnterShortOnStopClose.....	760
EnterLongAtLimitClose.....	762
EnterShortAtLimitClose.....	764
Exit Order Functions	766
ExitAllUnitsOnOpen	768
ExitUnitOnOpen	769
ExitAllUnitsOnStopOpen.....	770
ExitAllUnitsAtLimitOpen.....	771
ExitUnitOnStopOpen.....	772
ExitUnitAtLimitOpen	773
ExitAllUnitsOnStop	774
ExitUnitOnStop	775
ExitAllUnitsAtLimit	777
ExitUnitAtLimit	778
ExitAllUnitsOnClose	780
ExitUnitOnClose	781
ExitAllUnitsOnStopClose.....	782
ExitUnitOnStopClose.....	783
ExitAllUnitsAtLimitClose.....	784
ExitUnitAtLimitClose	785
Broker Order Information	786
Entry Day Exit Order	787
Position Adjustment Functions	788
AdjustPositionOnClose.....	790
AdjustPositionOnOpen.....	791
AdjustPositionOnStop.....	792
AdjustPositionAtLimit.....	793
AdjustSystemRiskToMax.....	794
Adjusting Size Example	796
4 Chart	798
AddBarLayer	805
AddBarSeries	809
AddContourLayer	811
AddHLOCLayer	814
AddLineLayer	815
AddLineSeries	817
AddMultiChart	822
AddScatter	823
Custom Chart Definitions	825
Make	826
NewAngularMeter	828
NewMultiChart	829
NewPie	830
NewXY	833
SetAutoScale	835
SetAxisTitle	836
SetBarGapShape	839
SetChartColors	843

SetLegendPosition	844
SetPieChartLabelColor	845
SetPlotArea	846
Setting AutoScale	848
SetxAxisDates	850
SetxAxisLabels	853
SetxAxisLinearScale	856
Sety2AxisLabelStyle	857
Sety2AxisLinearScale	858
Sety2AxisTitle	859
SetyAxisAutoScale	860
SetyAxisLabelStyle	861
SetyAxisLinearScale	862
5 Email Manager	863
EmailAddImage	864
EmailConnect	865
EmailConnectSSL	866
EmailDisconnect	868
EmailSend	869
EmailSendHTML	871
6 File Manager	873
Close	875
CountLines	877
DefaultFolder	878
EndOfFile	879
OpenAppend	881
OpenRead	883
OpenWrite	885
PartialLine	887
ReadLine	888
WriteLine	890
WriteString	892
7 Instrument	894
Accessing Instruments	896
Instrument Context	901
Instrument Priming	903
Correlation Functions	905
ResetCloselyCorrelated.....	906
ResetLooselyCorrelated.....	907
AddCloselyCorrelated.....	908
AddLooselyCorrelated.....	909
Correlation Properties	910
Data Properties	912
Data Class Properties.....	923
TradesOnTradeBar	927
DataFunctions	929
AddCommission	931
DisableTrading	932
Extract	938
GetDateTimelndex	940
GetDayIndex	941
PriceFormat	942
RealPrice	943

RoundTick	944
RoundTickDown	945
RoundTickUp	946
Group Properties	947
Historical Trade Properties	949
Loading Functions	951
LoadSymbol	952
LoadByLongRank	956
LoadByShortRank	957
LoadExternalData	958
LoadBPVFromFile	960
LoadIPVFromFile	965
Position Functions	974
SetExitLimit	975
SetExitStop	976
SetUnitCustomValue	978
SetUnitPositionMessage	979
Position Properties	981
UnitRollCalculations	984
Ranking Functions	986
SetLongRankingValue	988
SetShortRankingValue	990
SetCustomSortValue	992
Ranking Properties	992
Trade Control Functions	994
AllowLongTrades	996
AllowShortTrades	997
AllowAllTrades	998
DenyLongTrades	999
DenyShortTrades	1000
DenyAllTrades	1001
Trade Control Properties	1002
8 Order	1003
Order Context	1006
Changing Orders	1006
Creating Orders	1008
Order Functions	1011
Reject	1013
SetAlgoParameter	1014
SetClearingIntent	1015
SetCustomValue	1016
SetFillPrice	1017
SetLimitPrice	1018
SetOrderRef	1019
SetOrderReportMessage	1020
SetQuantity	1022
SetRuleLabel	1024
SetSortValue	1026
SetStopPrice	1029
SetTimelnForce	1031
OrderProperties	1032
blockName	1035
brokerSymbol	1036
clearingIntent	1037

continueProcessing.....	1038
customValue	1040
entryRisk	1042
executionType	1043
fillPrice	1045
goodAfterDateTime.....	1046
goodTillDateTime	1047
hidden	1048
IBOrderNumber	1049
isBuy	1050
isEntry	1050
isWholeExit	1052
limitPrice	1053
noStopPrice	1054
orderPrice	1055
orderReportMessage.....	1056
orderType	1057
OutSideRTH	1058
position	1060
processingMessage.....	1061
quantity	1062
referenceID	1063
ruleLabel	1065
scriptName	1066
sortValue	1067
stopPrice	1068
symbol	1069
systemBlockName	1070
systemName	1071
unitNumber	1072
9 Script	1073
Creating Custom Scripts	1076
Custom Parameters	1078
Custom Script Use	1080
Creating A User Function	1082
Custom Function Example	1087
Script Functions	1088
Execute	1091
GetSeriesValue	1093
InstrumentLoop	1095
OrderLoop	1097
SetReturnValue	1099
SetReturnValueList.....	1100
Script Properties	1102
ParameterCount	1104
ParameterList	1105
ReturnValue	1107
ReturnValueList	1108
SeriesParameterCount.....	1111
StringParameterCount.....	1112
StringParameterList.....	1113
StringReturnValue	1115
10 System	1116

Global Suite System	1117
System Functions	1122
Accessing System Portfolio Instruments.....	1125
Extract	1128
OrderExists	1130
RankInstruments	1132
SetAccountNumber.....	1133
SetAlternateOrder	1134
SetRoutingExchange.....	1137
SortInstrumentList	1138
TotalOrders	1142
System Properties	1144
marketOrdersValue.....	1151
totalOpenOrders	1155
11 Test	1157
Equity Properties	1158
Capital Adds Draws Total.....	1161
Fee Accruals	1165
Scripting Add-Draws.....	1168
VADI	1169
General Properties	1171
instrumentListRef	1175
OrderReportPath	1176
ResultsReportPath	1177
SummaryResultsPath.....	1178
Global Parameter Properties	1179
Functions	1183
AbortSimulation	1186
AbortTest	1187
AddStatistic	1188
GetStatisticValue	1190
GetSteppedParameter.....	1191
SetAlternateSystem.....	1194
SetAutoPriming	1198
SetChartSimulationHtml.....	1201
SetChartTestHtml	1205
SetDisplayOrderReport.....	1208
SetGeneratingOrders.....	1209
SetGoodnessForWalkForward.....	1211
SetGoodnessToChart.....	1212
SetIBPositionSynchOrderType	1214
SetMinimumTick	1214
SetSilentTestRun	1216
SetSmartFillExit	1217
SetTotalParameterRuns.....	1219
SortInstrumentList	1220
Test String Arrays	1222
Test String Array Sorting	1226
UpdateOtherExpenses.....	1235
SetMarginEquity	1236
Test Statistics	1237
Trade Properties	1241

Part VI Common Questions	1245
1 Trading Blox Keywords	1247
2 The Life of a Test	1248
3 Test Results	1250
4 How Stops Work	1251
5 Shortcut Keys	1253
6 TradingBlox.ini	1255
7 Removed Keywords	1256
Part VII Troubleshooting	1257
Index	1259

Getting Started Tutorial

Part



Part 1 – Getting Started Tutorial

Trading Blox Builder Guide

Program Version: 5.4.7

Help Version: Thursday, March 21, 2024

Trading Blox creates trading systems. Simple systems, very sophisticated trading systems using basic building blocks we call Blox. With Blox, you can quickly assemble a complete trading system, including entries, exits, money management, risk management, and portfolio management.

This section of the Trading Blox Builder Builder's guide is information about the various language features of Trading Blox Builder Basic Language. Information about the settings and how to test trading ideas is available in the Trading Blox Builder User's Guide.

A forum for mechanical system traders is in the Traders' Roundtable. In this forum, you will find questions and answers about ideas and other forum members posted. To get access to the Traders' Roundtable, send a request to customersupport@tradingblox.com, where you can get access to Trader's Roundtable Forum.

Once you can log into the forum, you'll find [Trading Blox Support](#), where questions and answers are available. In that same group is [Blox Marketplace](#), where users have place blox and other information that can be download by all Trading Blox Builder owners.

In the other sections of the forum, there are many references to many trading ideas and solutions.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 361

Section 1 – What Are Blox?

Blox Modules:

Blox are system blocks that encapsulate trading script ideas. Most of the Blocks are self-contained parts of a trading system designed to be connected with other Blocks 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 Blocks. 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 Block module, a package is created that can easily be linked to one or more systems that need the trading idea contained within the Blox. Blocks are trading objects, and while these objects only need to be created once, they can be used 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 learning how to use Trading Blox Builder, we have created two groups of topics that show what is important for each of the stages in creating a block that will perform a system task. As each topic lesson progresses, the details in the stages needed with a system will become more clear. For example, the main components needed in a system will be covered in the two groups of lessons as we continue each lesson and create a simple Moving Average Crossover system.

All the details that follow in this part of our tutorial area, and in the following tutorial section, are intended to provide insight for how to work with some of Trading Blox features. This process where blocks are scripted and created, will develop some understanding of what are some of a system's important features. As the lessons continue, the features required for trading methods will be shown

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.

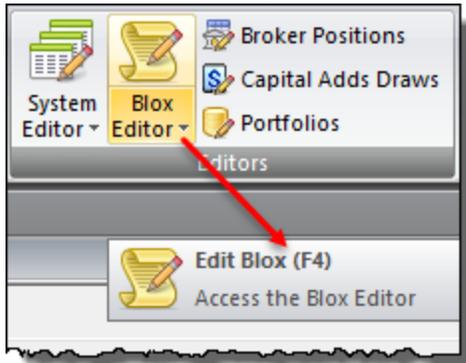
- Take a few minutes to understand how Trading Blox Basic uses its language operators by reviewing the tables on this page: [Operator Reference](#)

Links:

[Operator Reference](#)

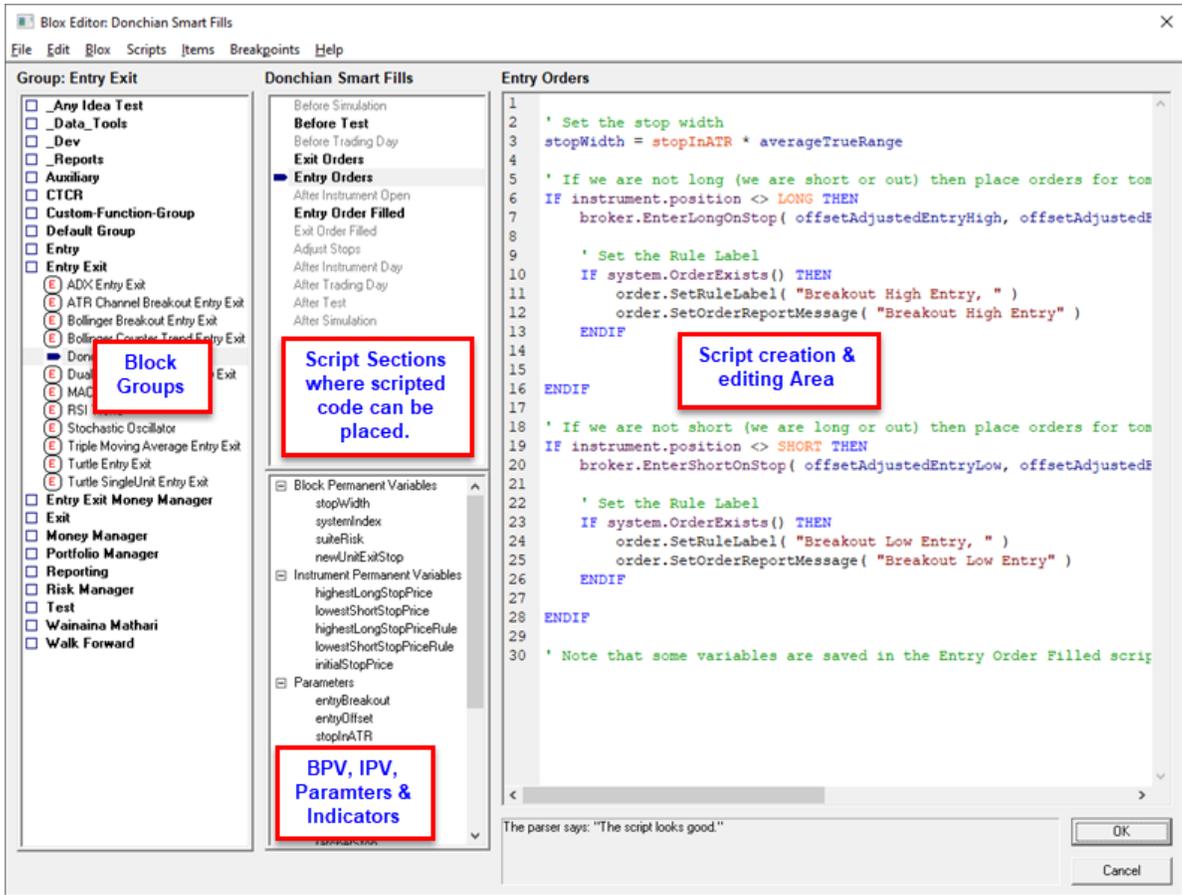
2.1 1. New System Blox

To get started, click the **Blox Editor** in the Editor Group menu button at the top of the main screen. When you click on the , 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:



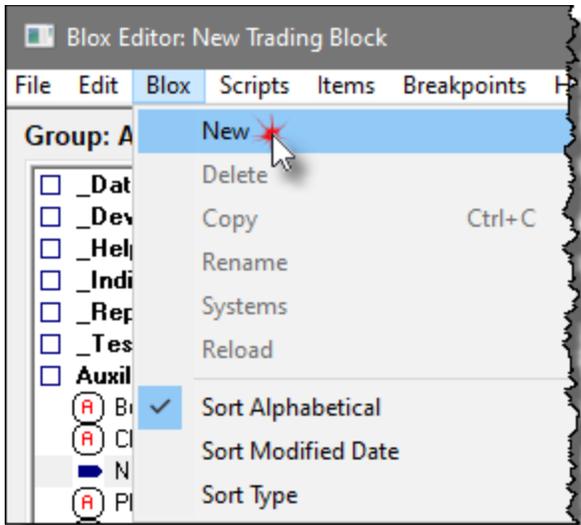
Blox Editor Access Button & F4-Key Access

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:



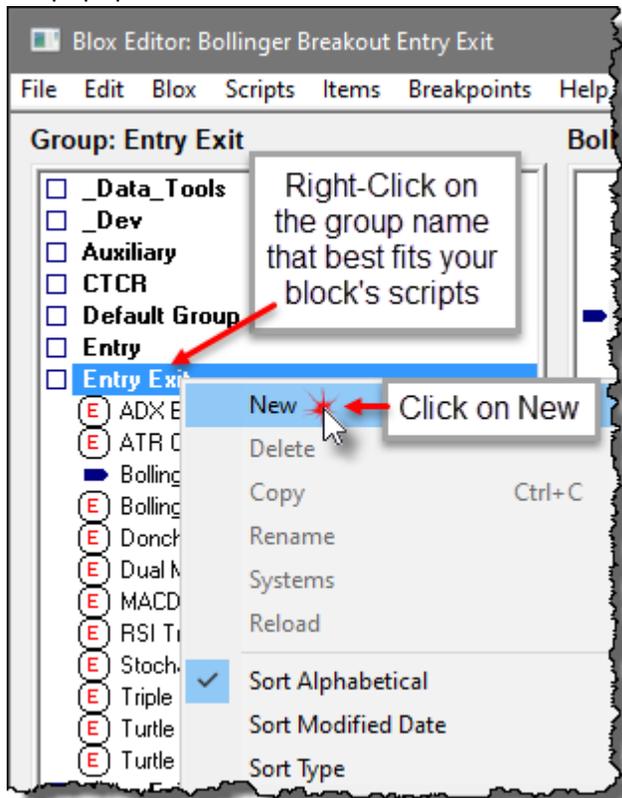
Trading Blox Editor Groups

Our next step is to create a new Blox module:



Trading Blox Editor Groups

A new block can be created by clicking on the Blox menu option in the Blox Basic Editor and selecting New. A second method is to right click on a Group name and select the New option from the popup menu.



Right-Button Click to Create New Block

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 you will be creating, and whether you want to add the common script sections for this blox type included in the Blox automatically when it appears:

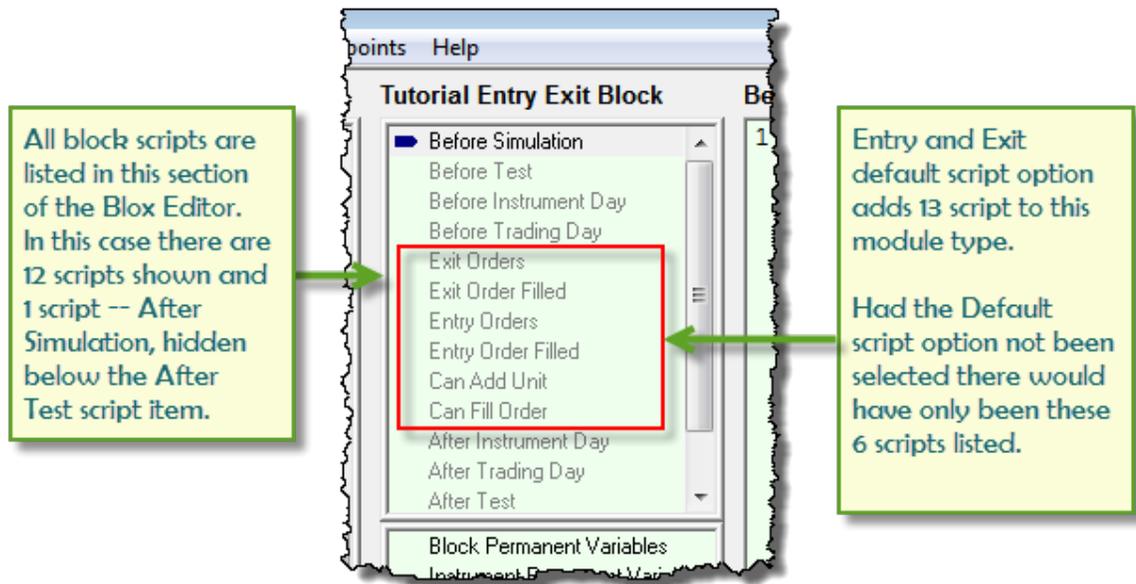
Enter the field names that identify the Block

- Name this new Block the "**Tutorial Entry Exit Lesson 1**" by typing in the blox name in the Name field.
- The second field identifies this block as an **Entry Exit** block.
- Next the section name can identify the system identified when created, or it can be anything that would be helpful later on.
- The last text field is the label that will be displayed when the "Tab" option is enabled. See the Trading Blox Builder User's Guide > **Preferences > General > Use System Component Tabs** for information in how to change how parameters are listed on the Main System Display.
- Our last step is to 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**.

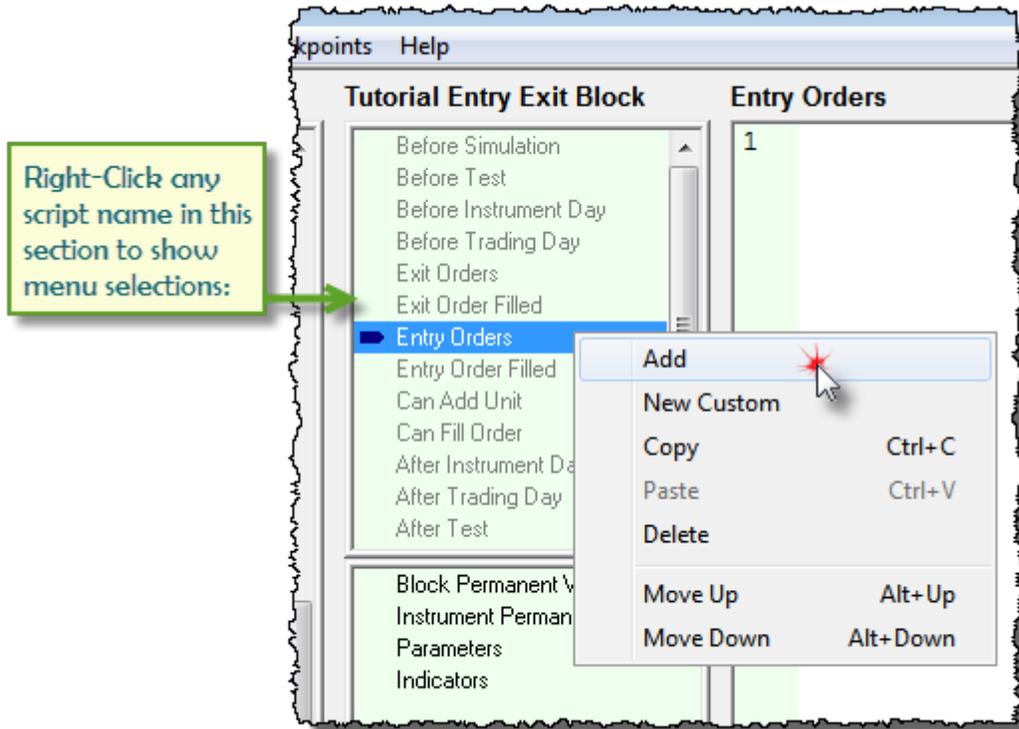
Note:

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.

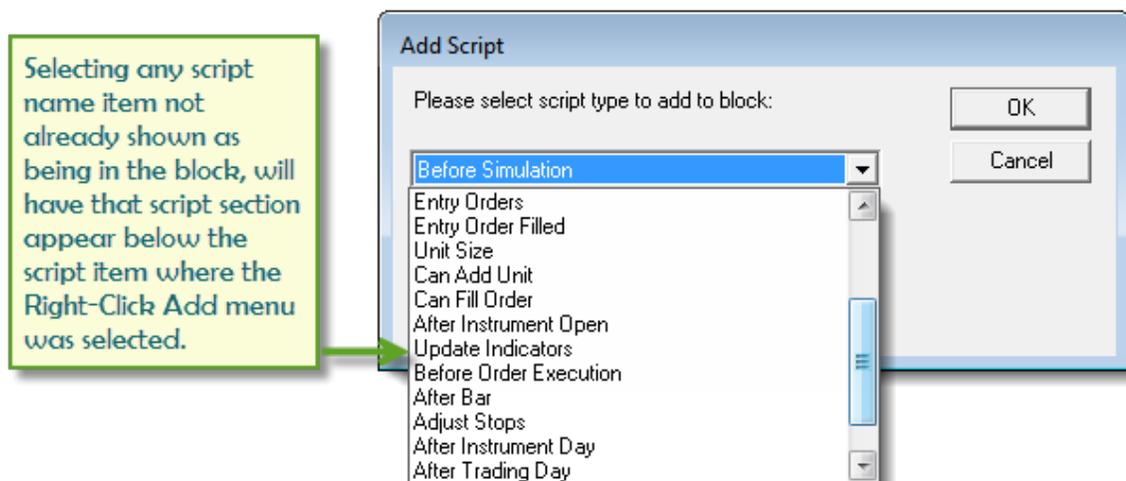
We now have a new block in the list, and ready to begin adding rules. When we created this tutorial block we elected to add all the common default scripts. When we review the list of scripts we find there are 13 names in our list:



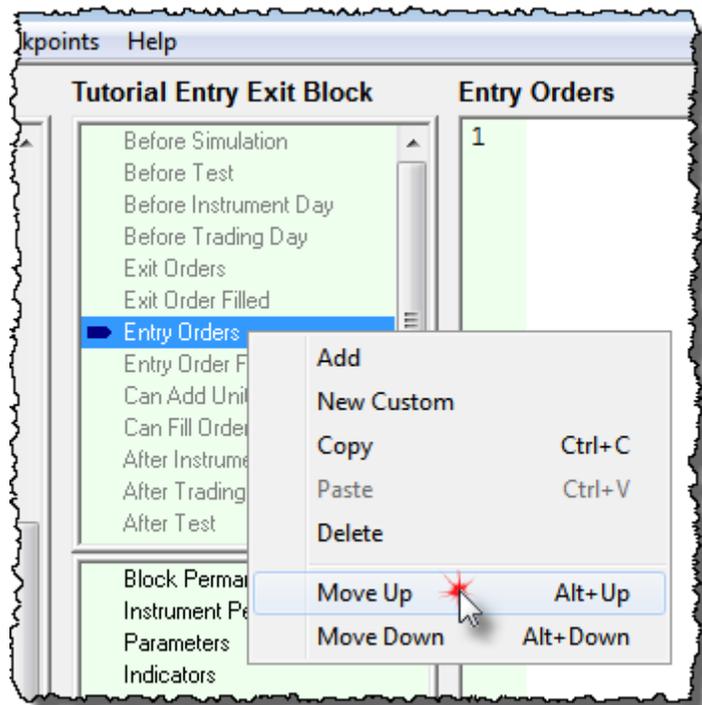
When we created our block, we enabled the Default Script option so the often used scripts in this type of module would be added and shown for this tutorial. Had we not used that option and if we needed a script section, it could have been easily added by right-clicking on any of the script names in the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



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.

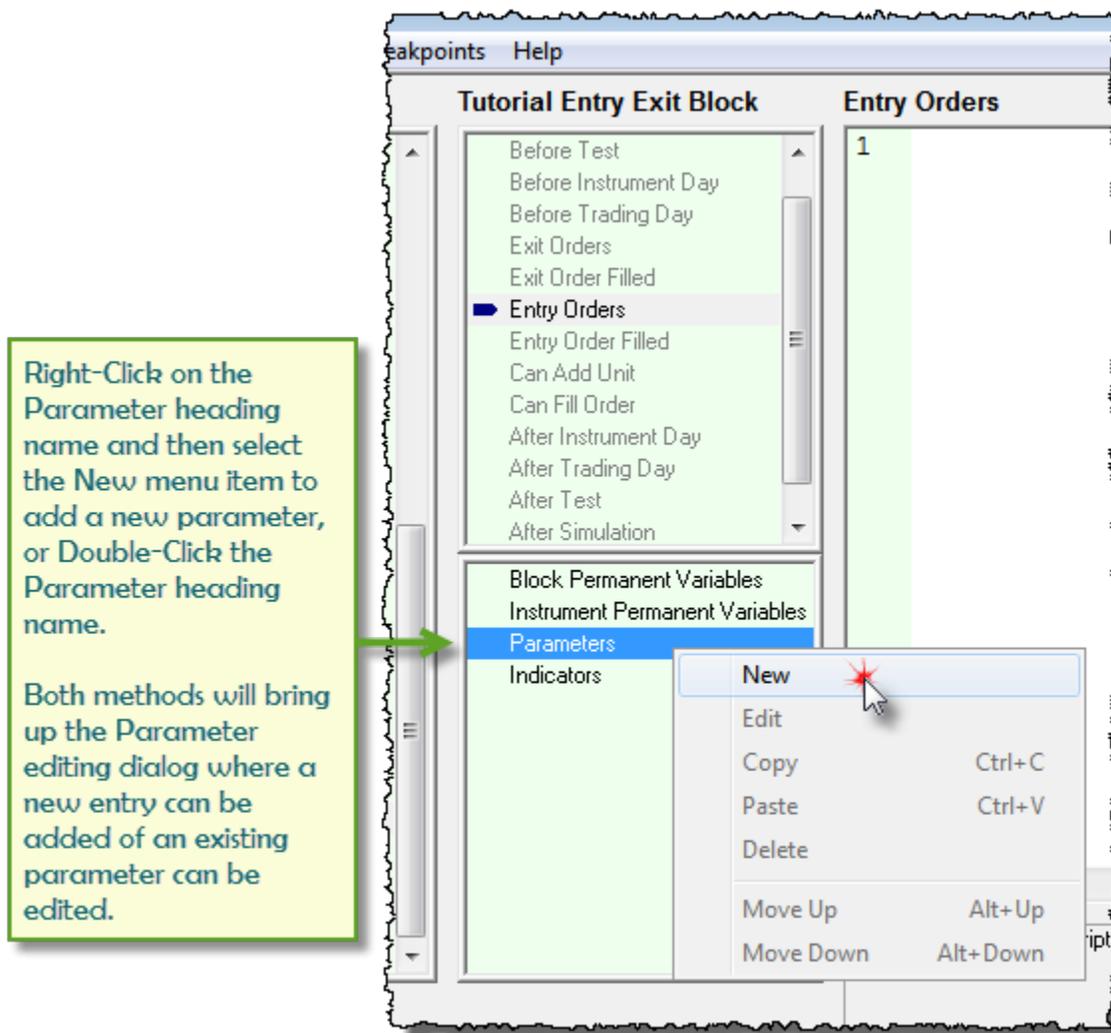
2.2 2. Adding Parameters

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.



Adding Parameters

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.

New 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

Default Value:

Scope:

Used for Lookback Stepping Enabled

Stepping Priority:

Selector Entries:

Entry	Basic Constant

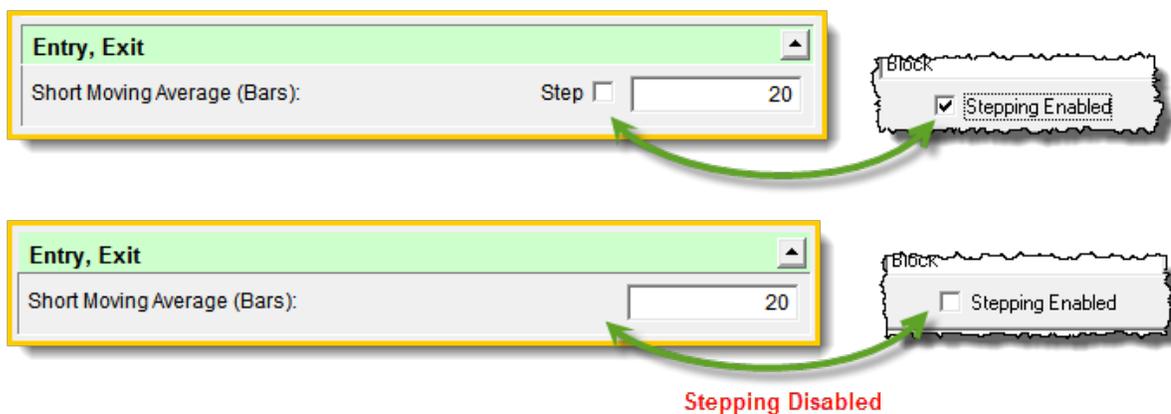
Buttons: OK, Cancel, Add Entry, Delete Entry, Move Entry Up, Move Entry Down

Adding Parameters

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.



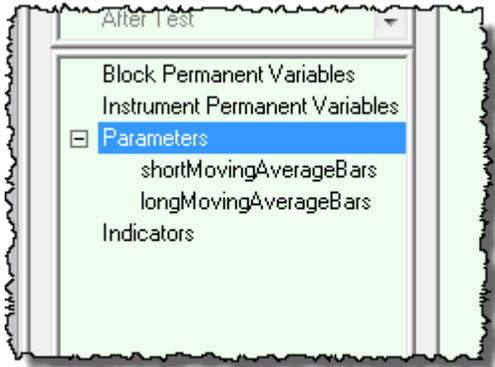
Adding Parameters

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

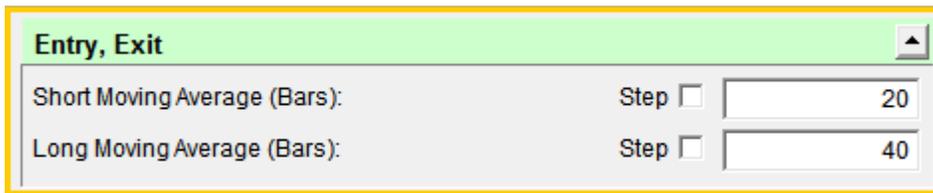
With the above understanding, create the `longMovingAverageBars` period calculation length parameter. Enter the value of 40 as its default setting. Use the same option selection we used for the first parameter. Once you are finished, press **OK**.

Use the same process as described for the `longMovingAverageBars` parameter to create the `ShortMovingAverageBars` calculation period length parameter. Enter a value of 20 as this parameters period length.

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.

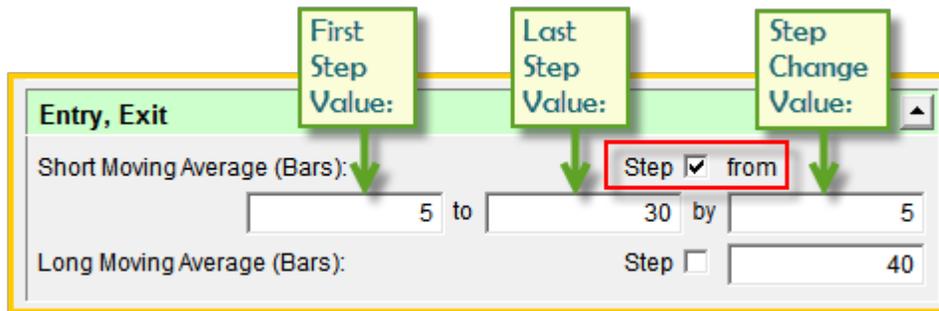


Adding Parameters



Adding Parameters

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:



Adding Parameters

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.

Adding Indicator

- **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 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 have 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: 1

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:

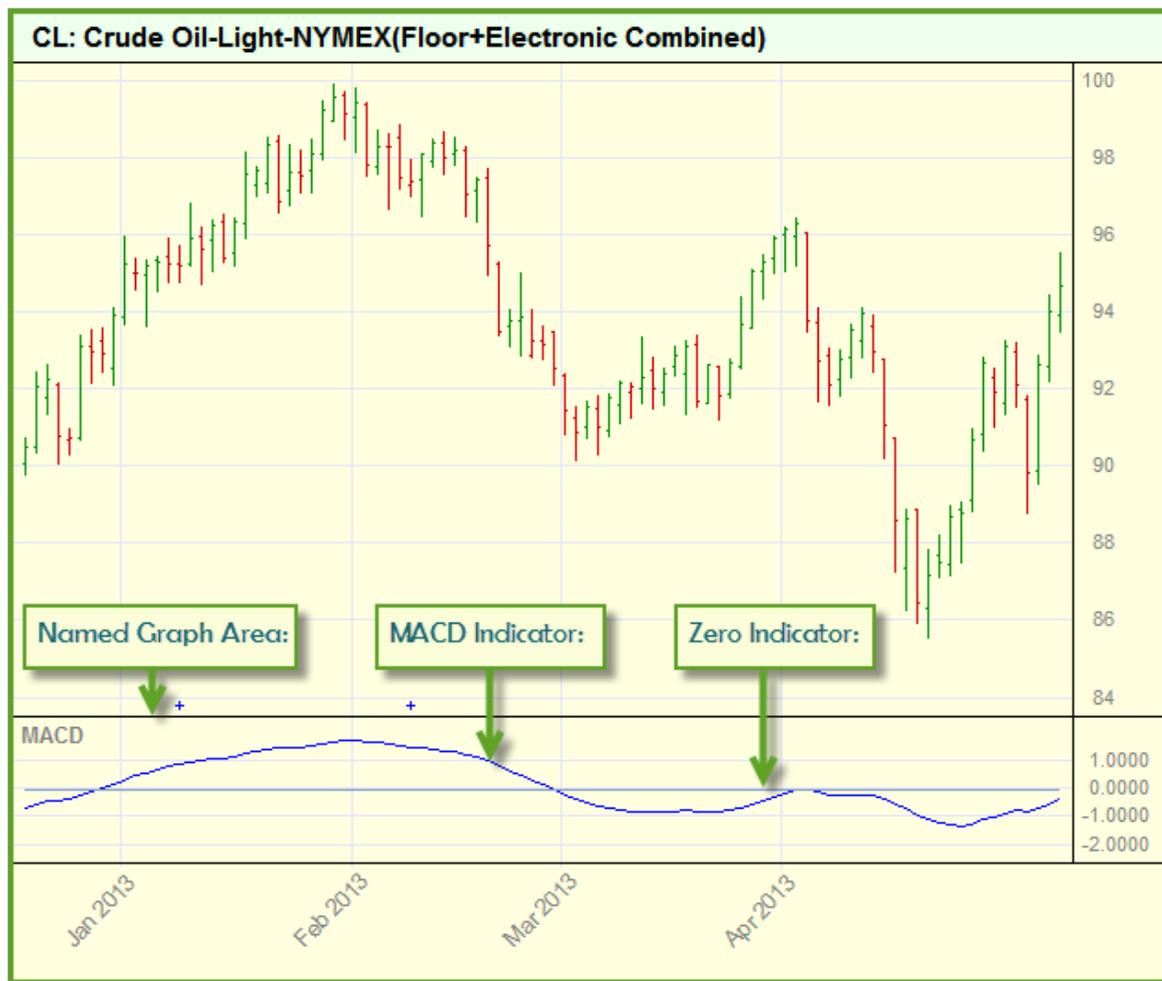
Offset Plot Ahead One B:

OK Cancel

Adding Indicator

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:



Adding Indicator

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

To create new indicators, review the information here: [Creating Indicators](#)

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 keyboard 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 positive then enter **LONG** on the open. If the **MACD** goes negative then enter short on the open.
- This next example shows 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:

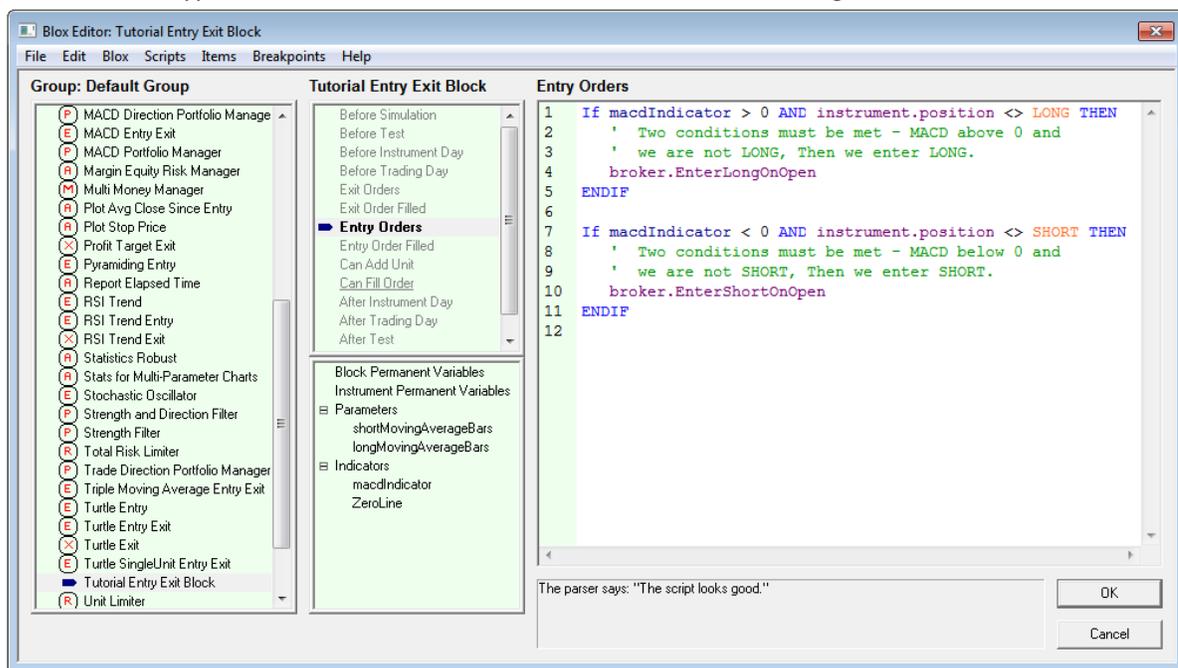
Example (don't type this line):

```
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:



Entering Code

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.

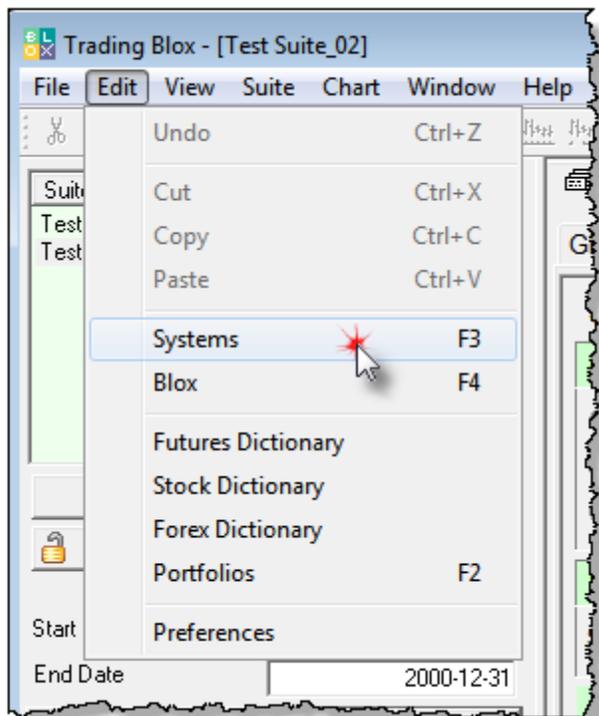
Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 280

2.5 5. Building A System

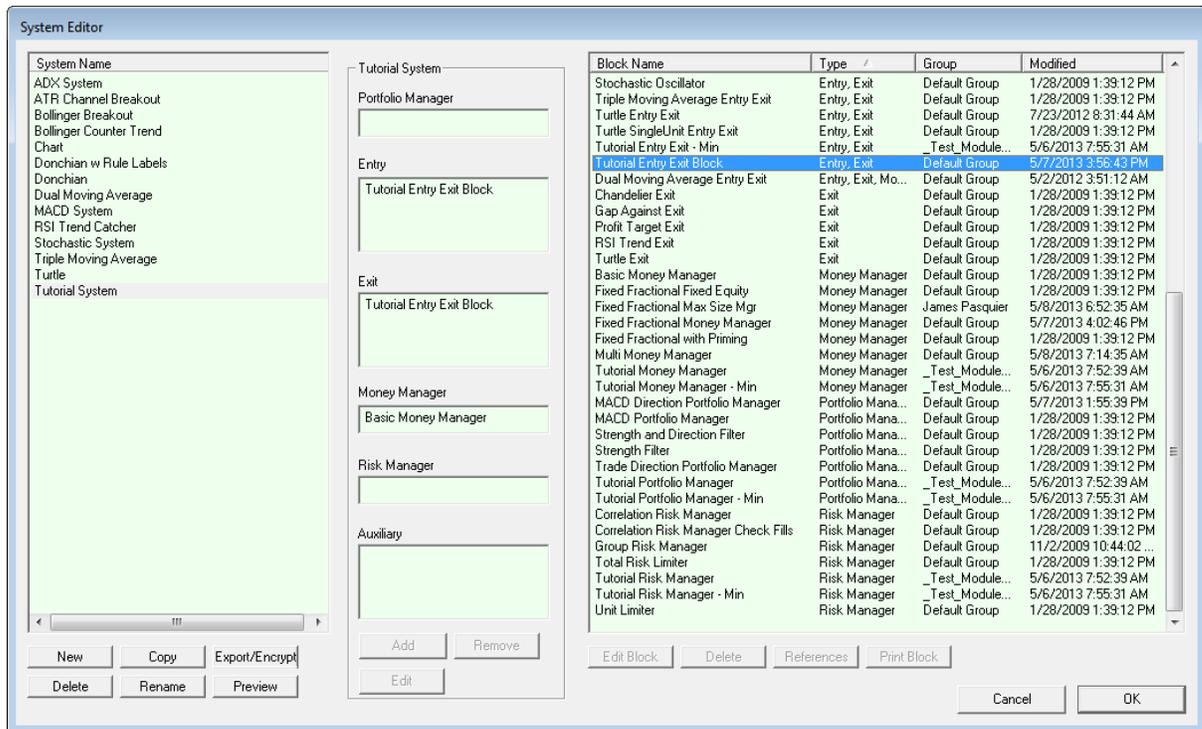
A Trading Blox system is a group of selected Blox modules that are available in the System Editor's module list. Each trader who builds their own system structure selects the specific modules they need to achieve the system's intended goal.

- 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**.



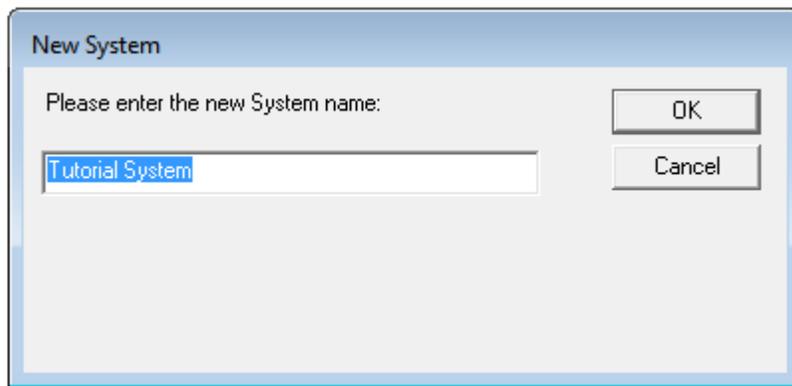
Building a System

- 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:



Building a System

- 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:



Building a System

- 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

Building a System

- Next locate the "Basic Money Manager" provided with Trading Blox. Click on it and then right-click the it so that it is 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

Building a System

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



Building a System

- 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 lose the previous module's edited parameter settings, and the new module will be loaded and display its default value until you change them.

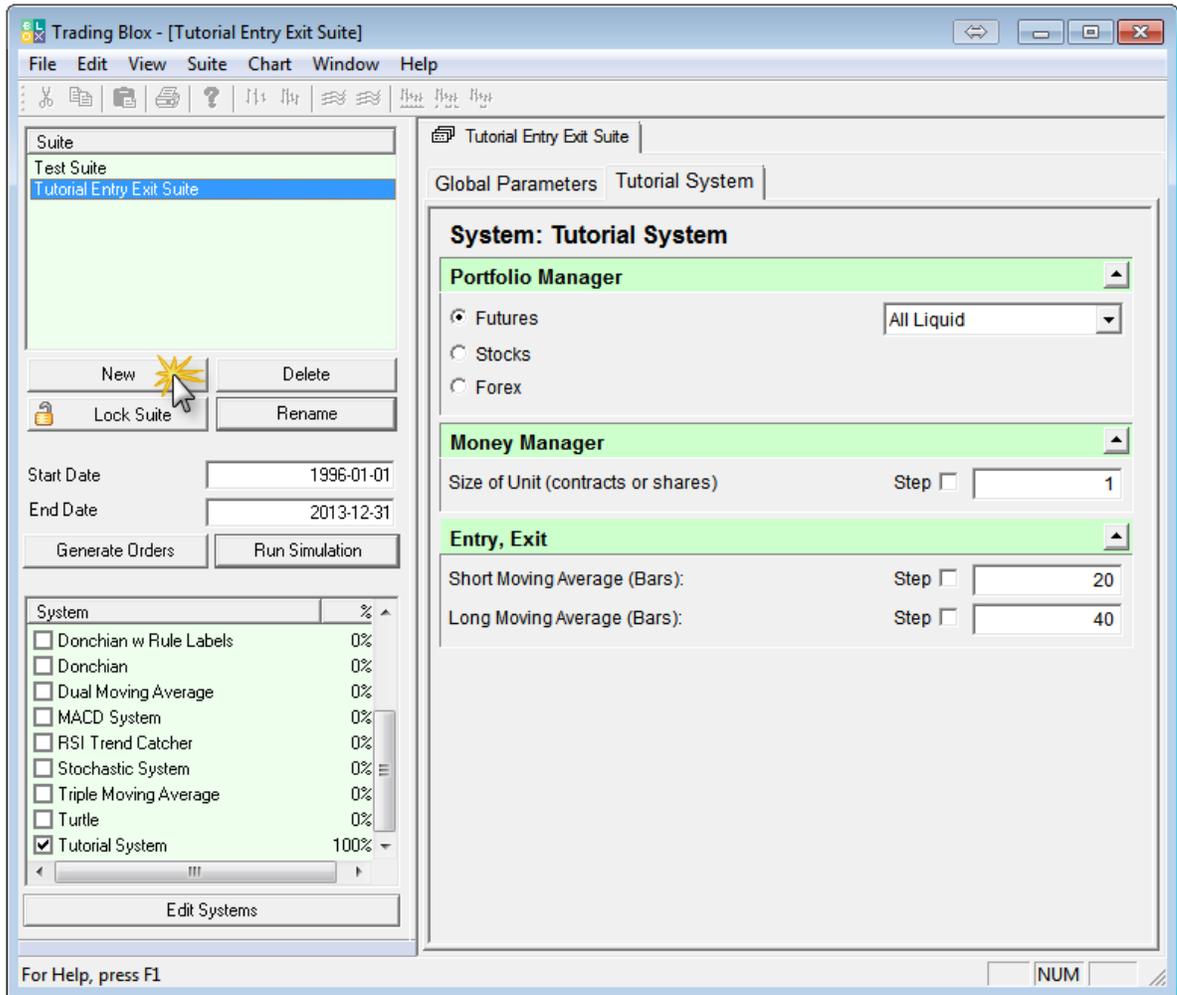
Entry and Exit, and Auxiliary module can have multiple modules of that type listed.

This completes the first series in this tutorial topic.

2.6 6. Creating A Suite

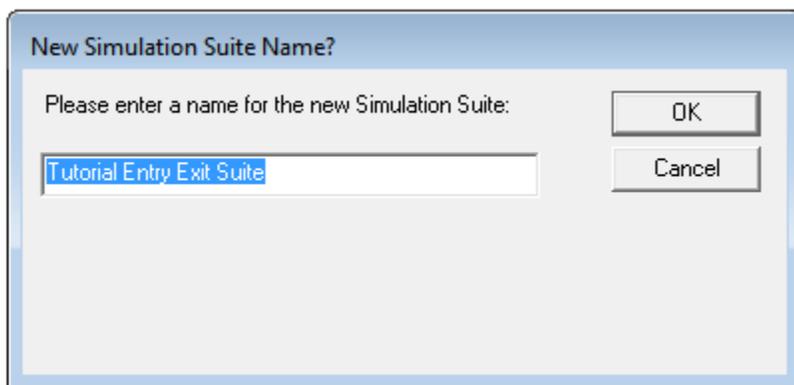
All simulation suites are created on this main menu screen.

- 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:



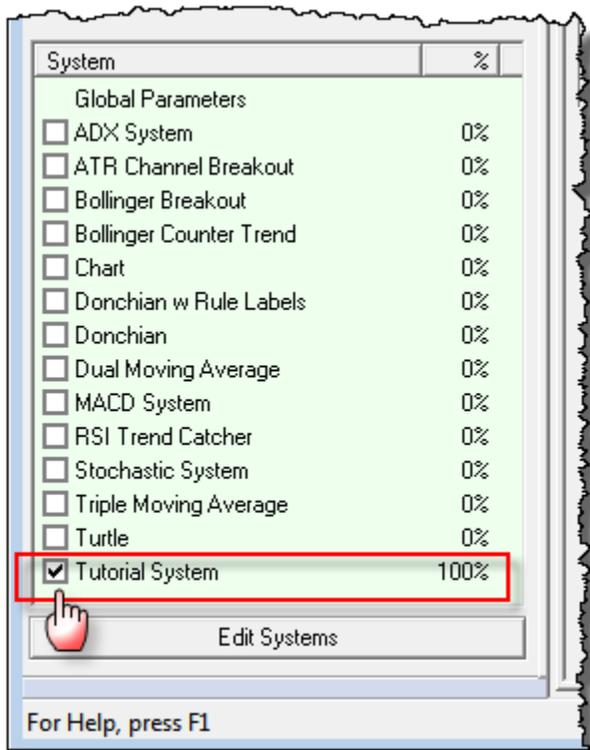
Creating A Suite

- 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:



Creating A Suite

- 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:



Creating A 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.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 236

Section 3 – Improving a New System

[Active Order Protection](#) 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	Trading Risk
8	Money Management

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 interfere with the position's 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.

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 479

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.

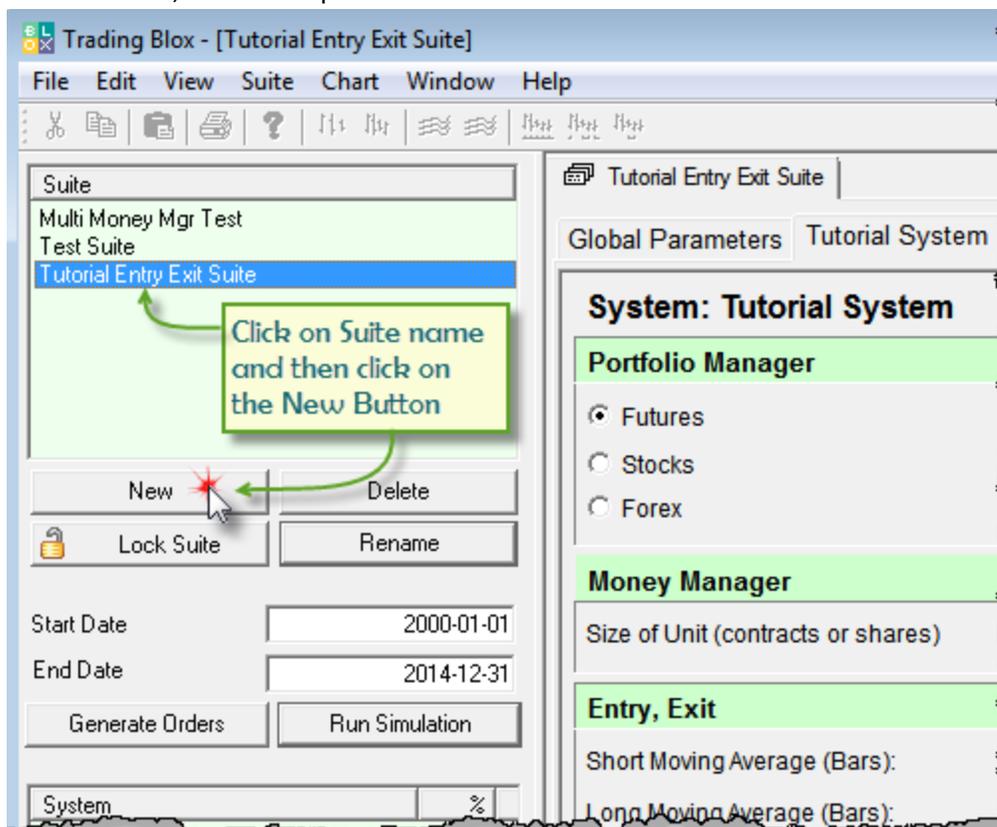
To make a copy of the module we will first start with making a copy of the first tutorial's Suite. This process allows us to create an exact copy of the original tutorial suite Global Parameter Settings along with the a link to the first tutorial's system list we used. We will also make a copy of the 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:

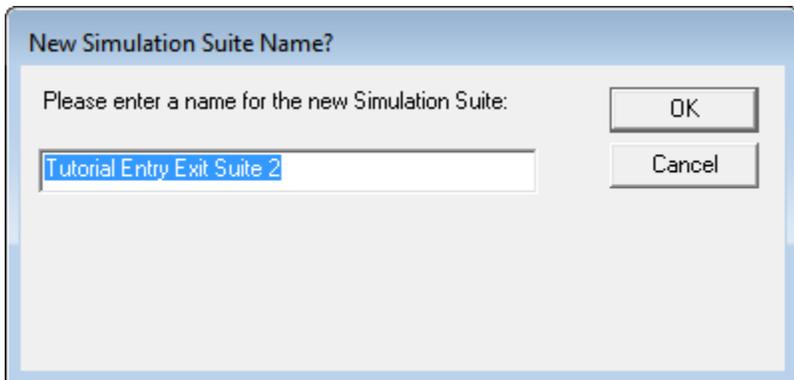
Creating a copy of an existing suite is the easiest way to retain the Global Parameter Settings you have established for a previous suite's simulation controls. In Trading Blox the "New" button acts will copy the selected suite highlighted. It will also retain the system names selected.

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.



Copy System Items

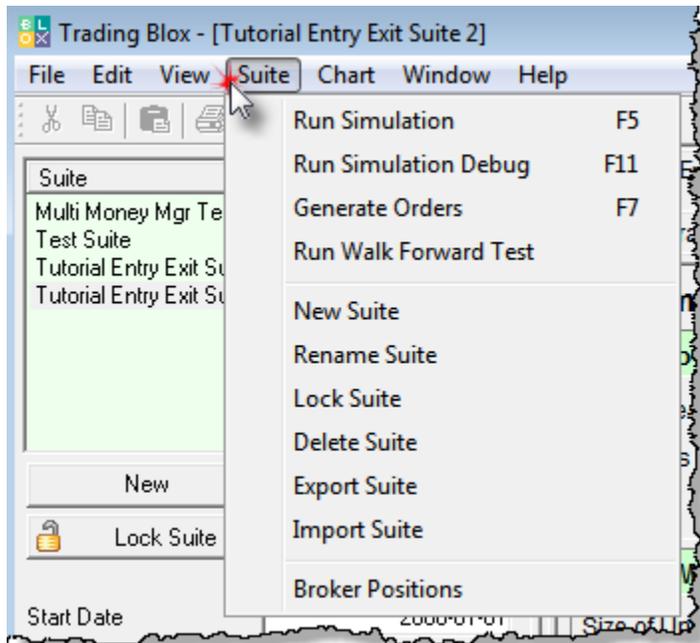
When the "New" button is clicked the suite simulation dialog window will open and it will automatically add the number 2 at the rightmost side of the name displayed. We are going to use the name with the #2 suffix and will click the OK button now.



Copy System Items

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.

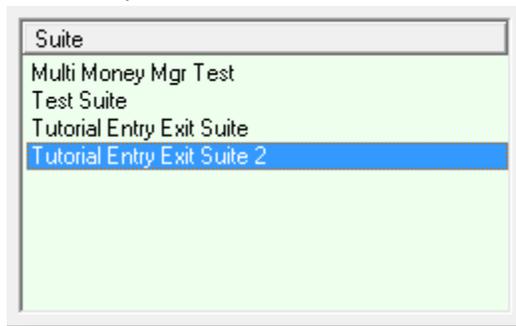
In Trading Blox Main Suite menu provides a complete range of methods for working with new or existing suites:



Copy System Items

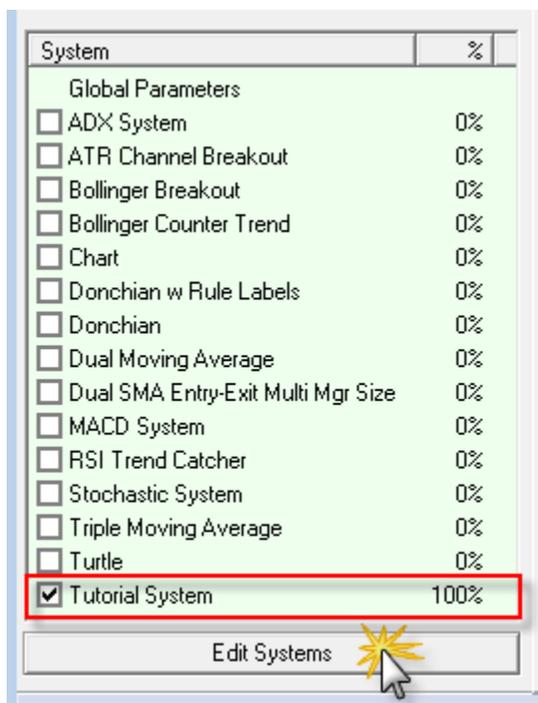
Making A Suite Copy:

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



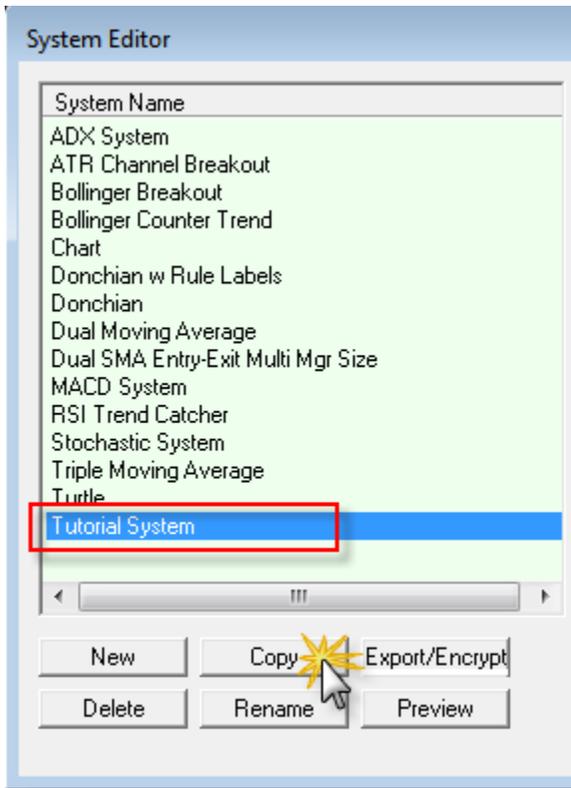
Copy System Items

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:



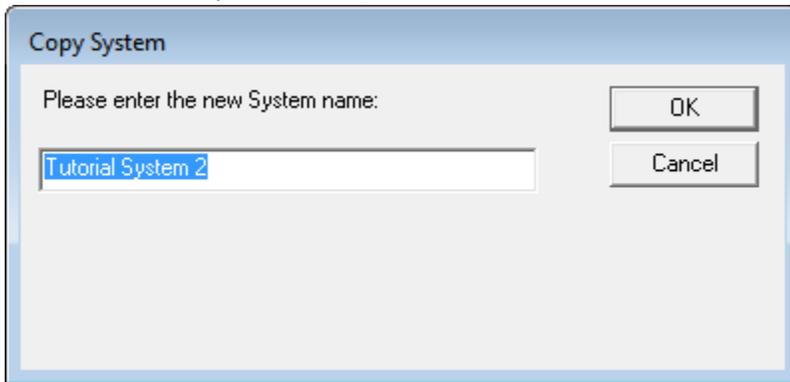
Copy System Items

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



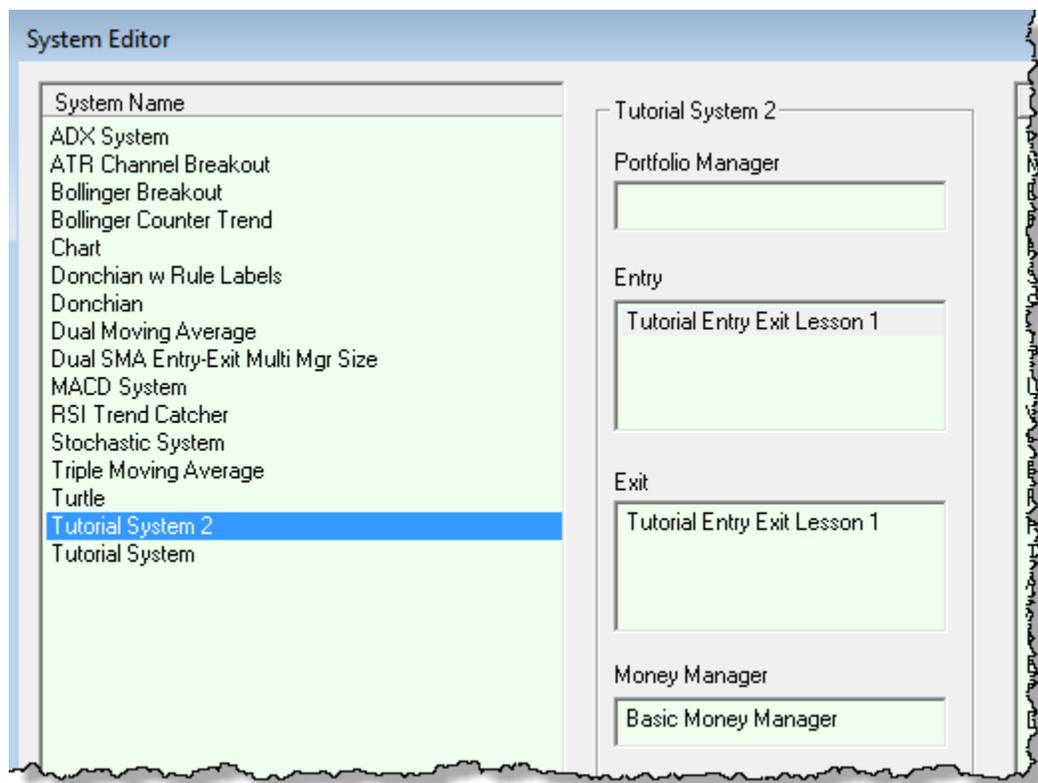
Copy System Items

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.



Copy System Items

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



Copy System Items

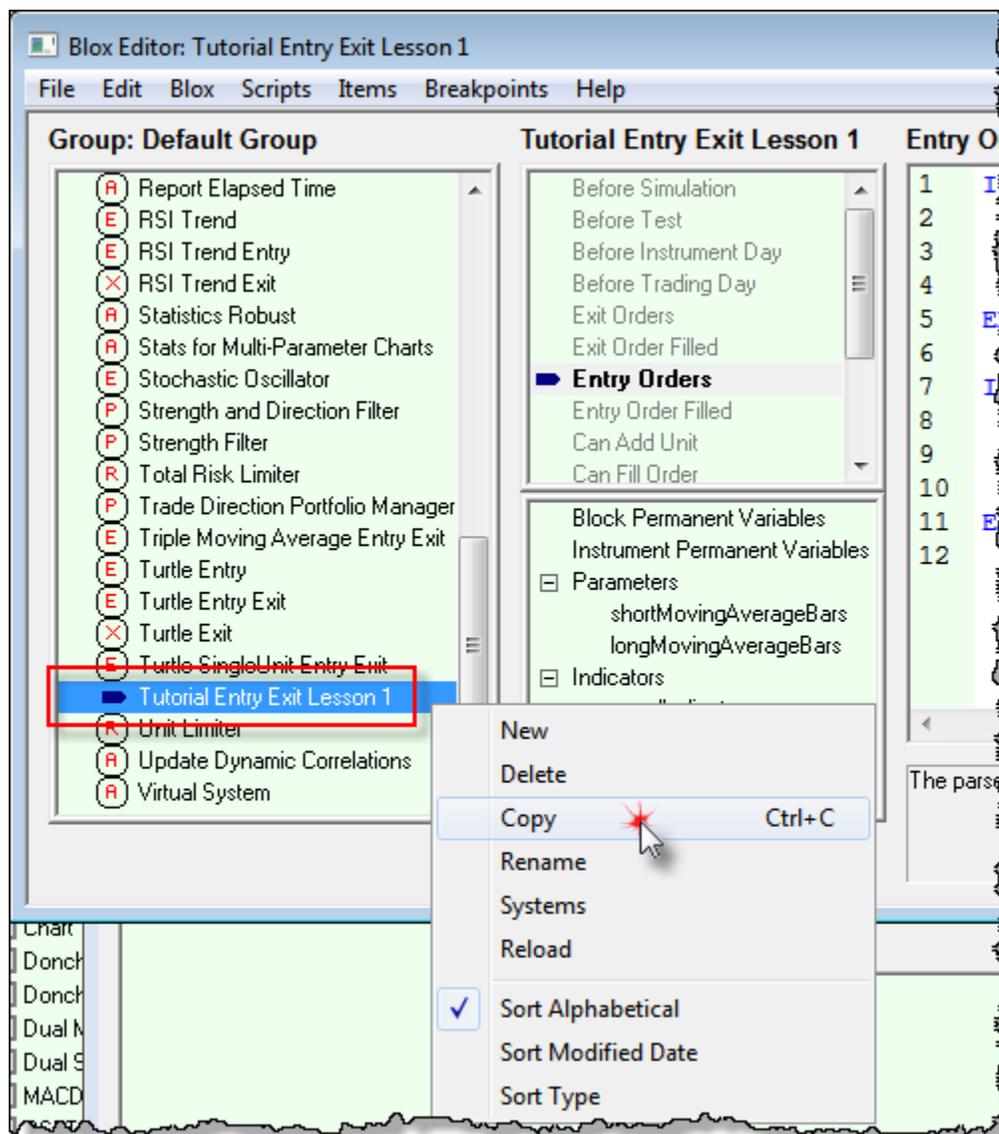
Our new system list will need changes that we don't want in our original entry exit logic. This means we will create a copy of the original tutorial entry exit blox so we can make changes and use the modified module in our new system and suite.

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.

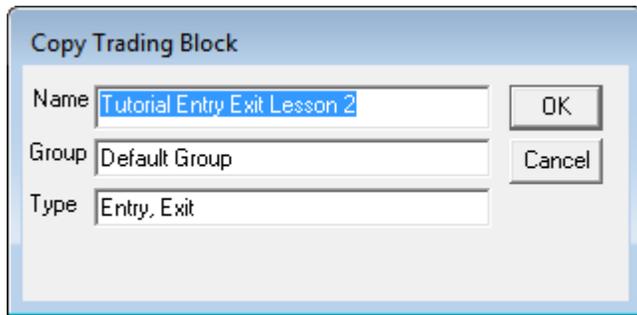
When the editor appears it will look something like this next image where the Tutorial Entry Exit Lesson1 blox module is highlighted.



Copy System Items

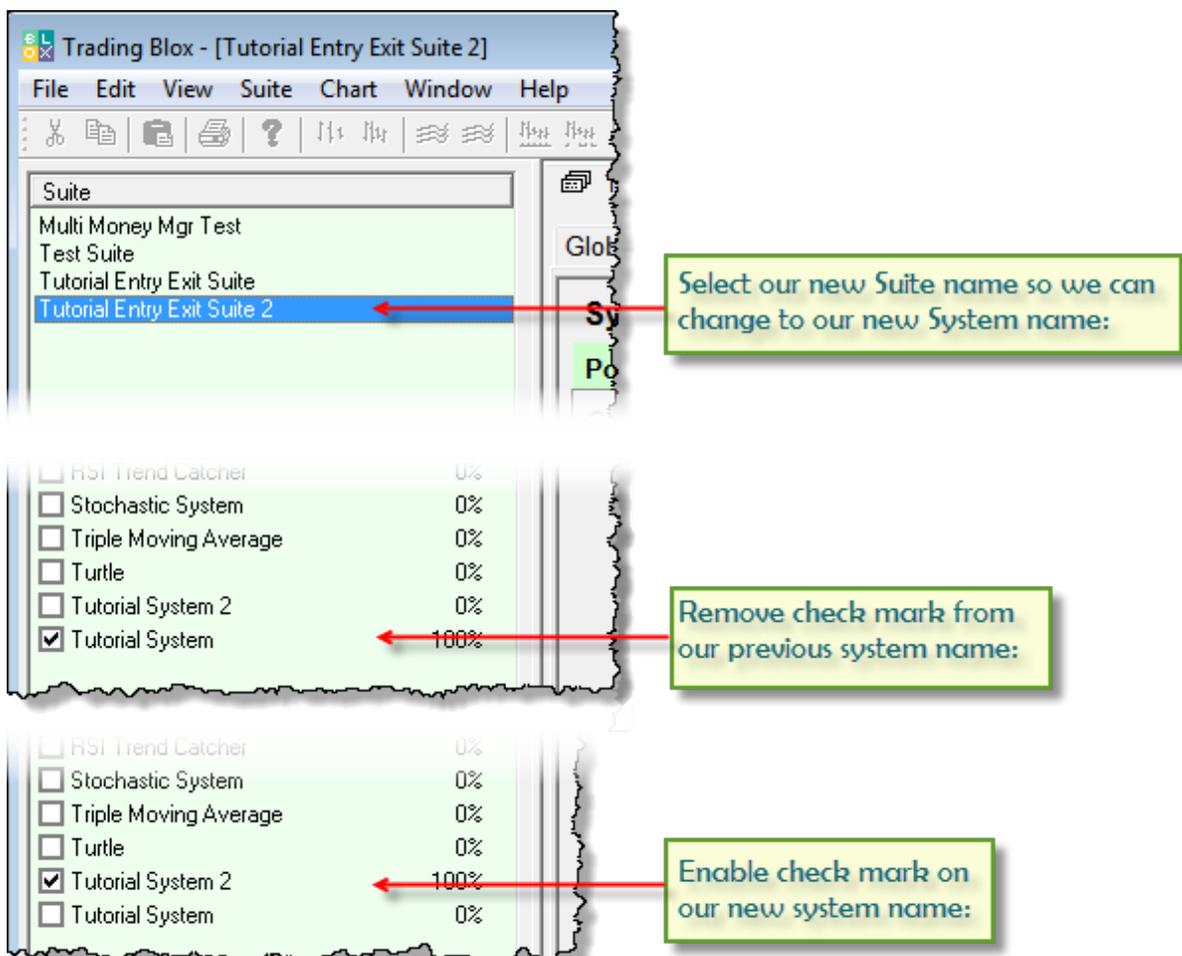
If you used a different name, click on that blox name and when right-click on it using the right-mouse button.

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.

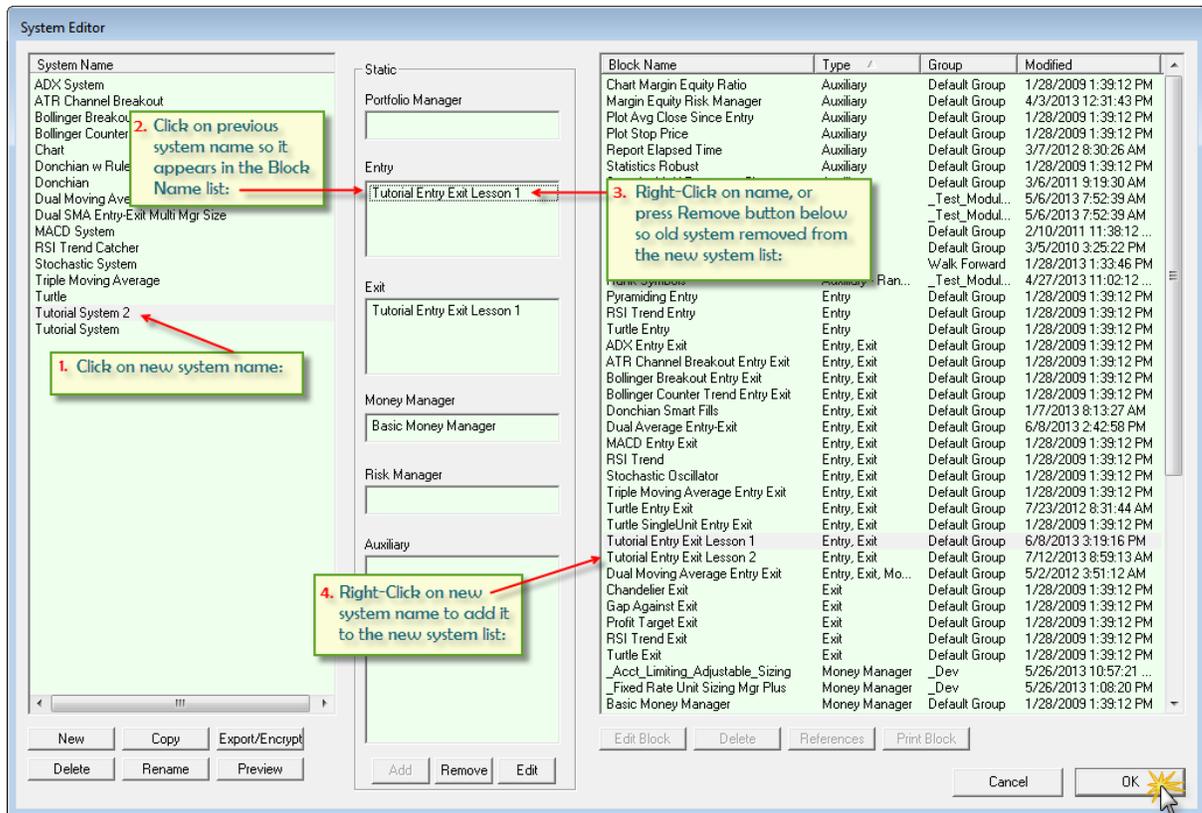


Copy System Items

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



Copy System Items



Copy System Items

This completes this topic.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 221

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.

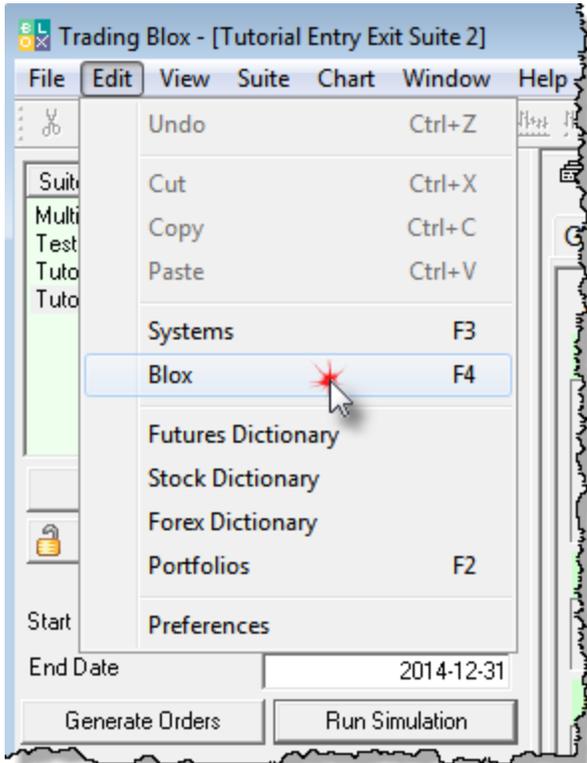
Protective Price Method:

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:

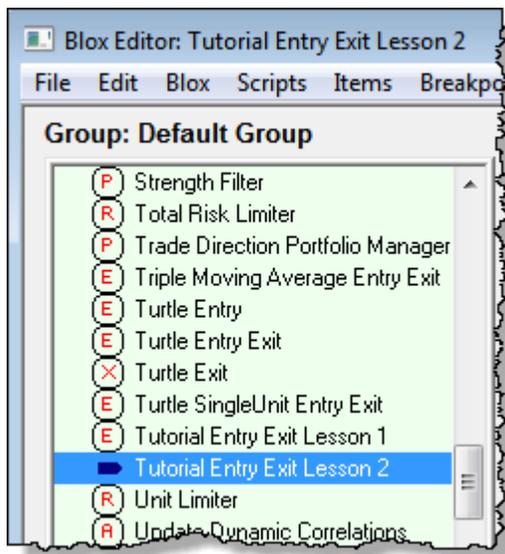
First parameter will be called **atrLength** to our parameter section. Our second parameter will be named **atrAdjustRate** to handle the volatility point adjustment.

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



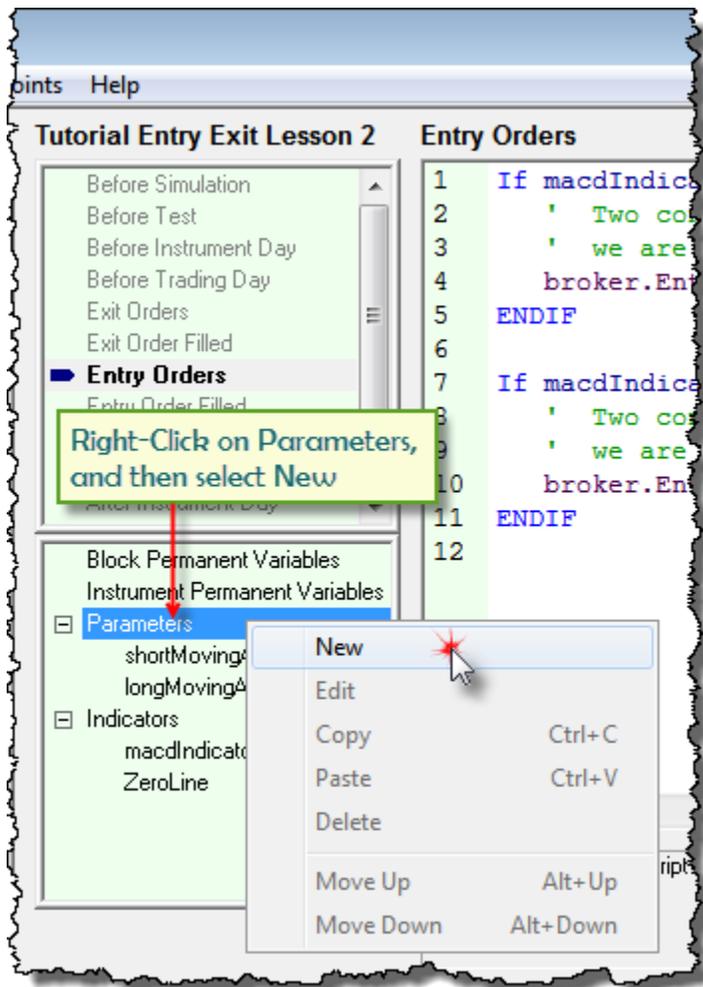
Protective Exit Orders

- 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**:



Protective Exit Orders

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



Protective Exit Orders

- 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:

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

Default Value:

Scope:

Used for Lookback Stepping Enabled

Stepping Priority:

Selector Entries:

Entry	Basic Constant

Add Entry Delete Entry

Move Entry Up Move Entry Down

OK Cancel

Protective Exit Orders

- 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.

New 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

Default Value:

Scope:

Used for Lookback Stepping Enabled

Stepping Priority:

Selector Entries:

Entry	Basic Constant

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

Protective Exit Orders

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:

The screenshot shows the 'Global Parameters' tab for 'Tutorial System 2'. The interface is divided into three main sections, each with a green header and a collapse arrow:

- Portfolio Manager:** Contains three radio buttons: 'Futures' (selected), 'Stocks', and 'Forex'. To the right is a dropdown menu currently set to 'All Futures'.
- Money Manager:** Contains a label 'Size of Unit (contracts or shares)' followed by a 'Step' checkbox and a text input field containing the number '1'.
- Entry, Exit:** Contains four rows of settings, each with a label, a 'Step' checkbox, and a text input field:
 - 'Short Moving Average (Bars):' with a value of '20'.
 - 'Long Moving Average (Bars):' with a value of '40'.
 - 'AvgTrueRange Calculation Length:' with a value of '34'.
 - 'ATR Point Size Adjust Rate:' with a value of '2.5'.

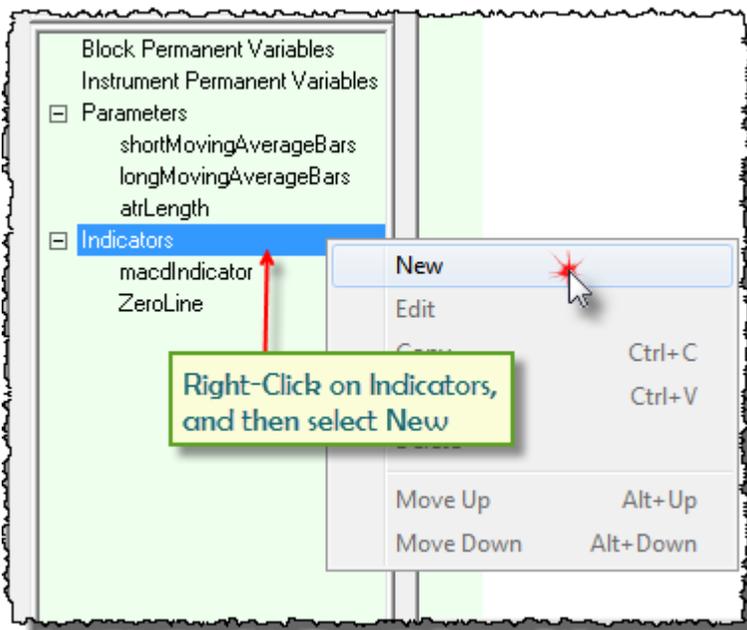
Protective Exit Orders

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**.



Protective Exit Orders

- 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.

New Indicator

Name for Code: avgTrueRange

Type: Average True Range

Value: []

Time Frame: Bar

Moving Average Bars: atLength

Smoothing: Enter Value (1)

Not Applicable: []

Indicator Value Expression: []

The parser likes the expression: []

Scope: Block

Plots

Display Value

Graph Title: ATR

Plot Color: [Orange]

Graph Area: Average True Range

Graph Style: Thin Line

Offset Plot Ahead One Bar

Callouts:

- Enter name: (points to Name for Code)
- Click Arrow and Select this name: (points to Type)
- Click Arrow & select new parameter (points to Moving Average Bars)
- Plots Shows Indicator on Chart (points to Plots checkbox)
- Display Value allows cursor position to display value of indicator in chart's data section. (points to Display Value checkbox)
- Enter name: (points to Graph Title)
- Select a plot color: (points to Plot Color)
- Plot area name: (points to Graph Area)
- Defines plot style: (points to Graph Style)

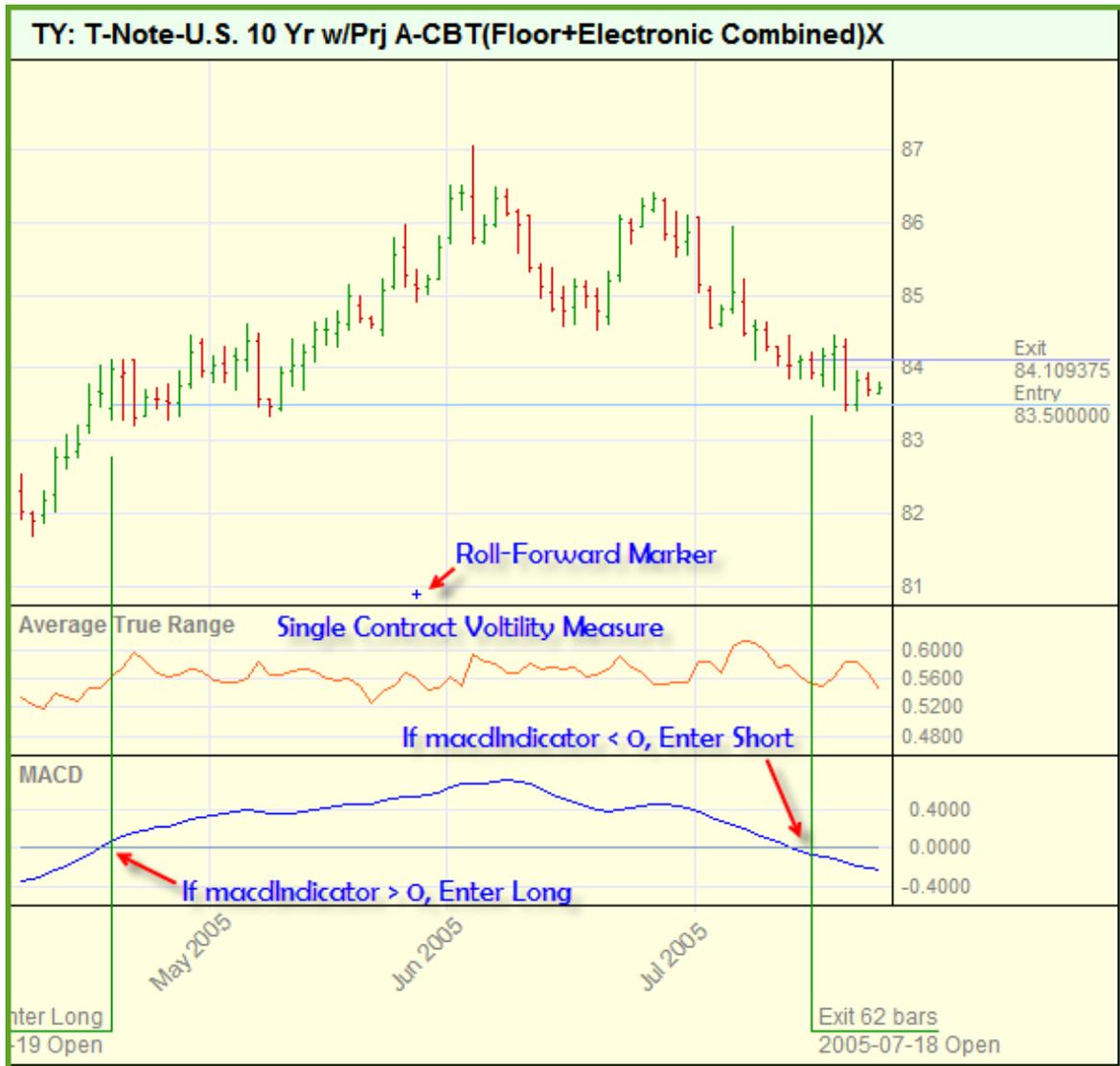
Protective Exit Orders

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.



Tutorial Lesson 2 Avg. TrueRange & MACD Graph Example

We are now ready to create the programming details in our second Entry Exit module so that we can apply a protective price order.

This completes this topic.

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 478

3.4 Entry Order Protection

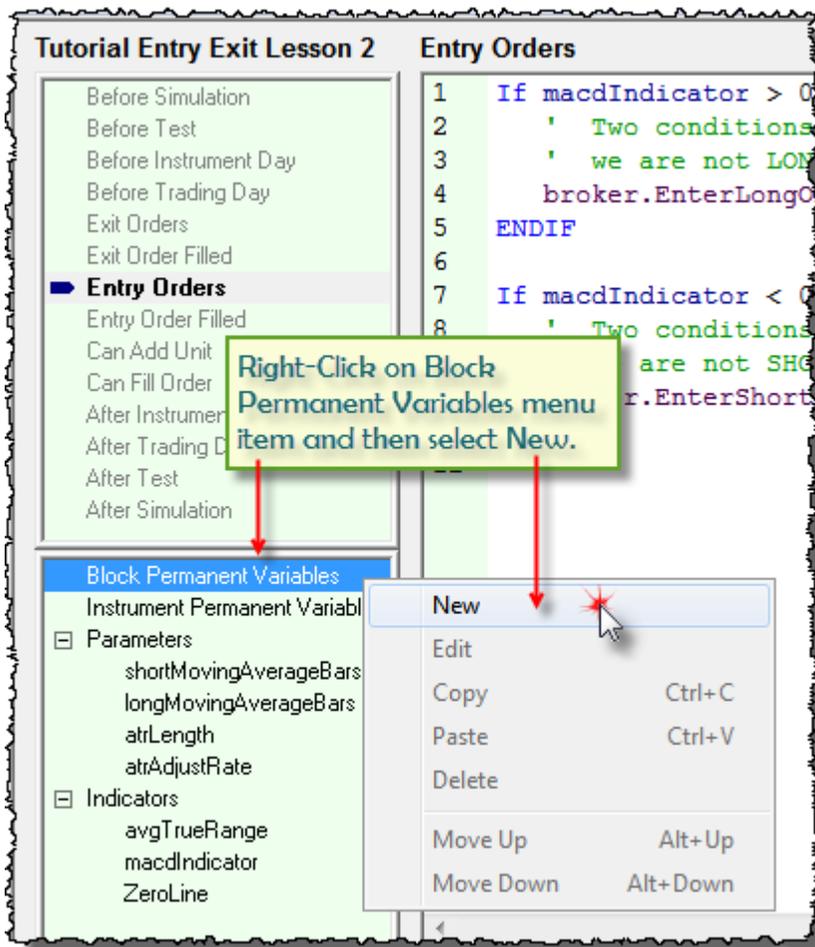
Our **Average True Range (ATR)** indicator is now operating and reporting a volatility measure for each price in the data file after the calculation priming period has primed.

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 **Blox 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:



Entry Order Protection

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

Block Permanent Variable

Name for Code:

Name for Humans:

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:

Scope:

OK

Cancel

Decimal variables require Floating, Price or a number Series so values to right of the decimal are not lost.

Entry Order Protection

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 <-
Modify
  broker.EnterLongOnOpen( instrument.close - exitPoints )      <-
Modify

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

- 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
```

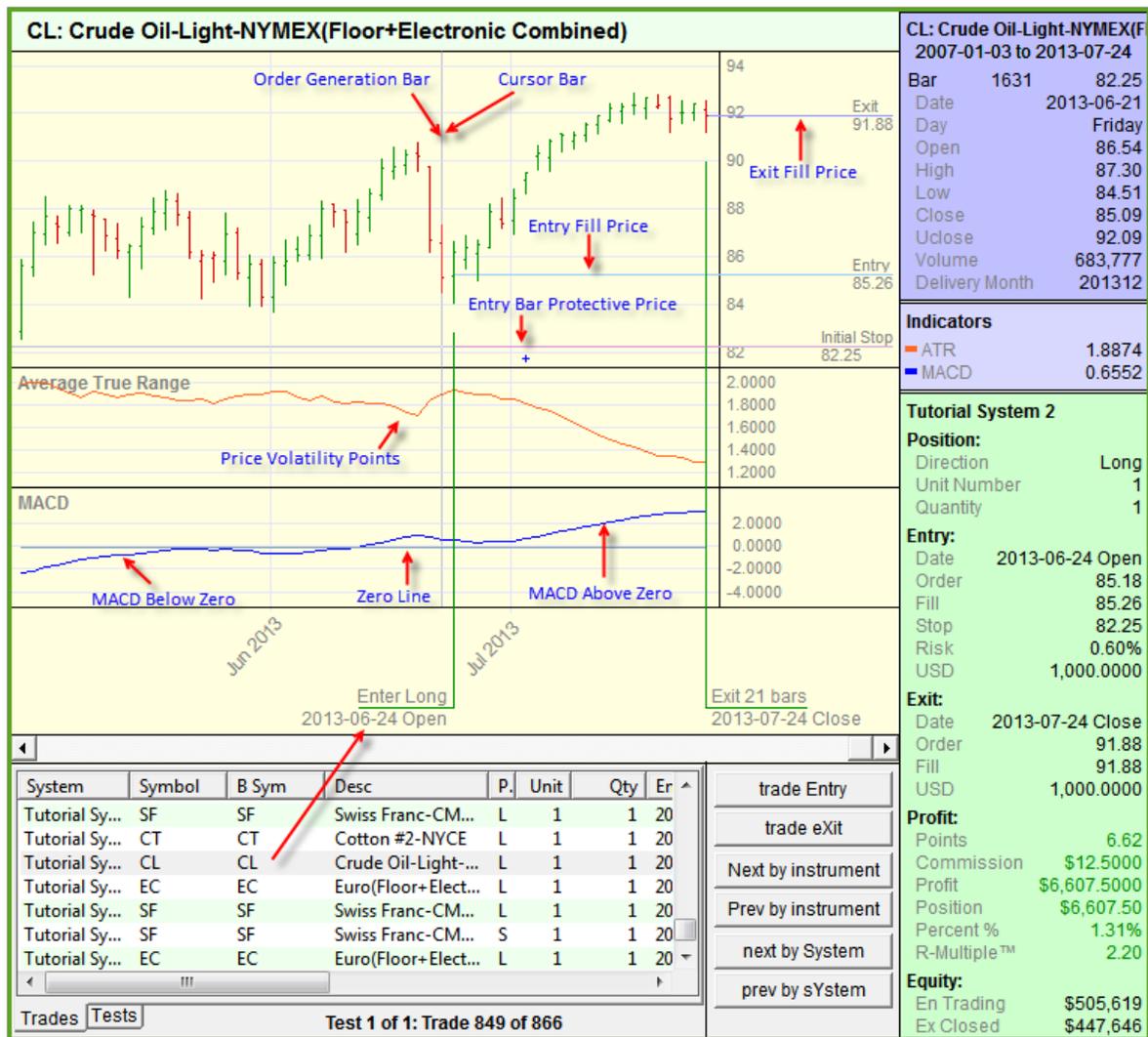
Entry Order Protection

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.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 297

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 7:59:38 AM

Right Click on the Block Name: Plot Stop Price

Active Order Protection

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.

Instrument Permanent Variable

Name for Code:

Name for Humans:

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

Variable Options

Default Value:

Scope:

Auto-Index -- Uses [n] as Lookback from Current Day

Plotting Controls

Plots Display Value

Log Scale

Plot Color:

Graph Area:

Graph Style:

Offset Plot Ahead One Bar

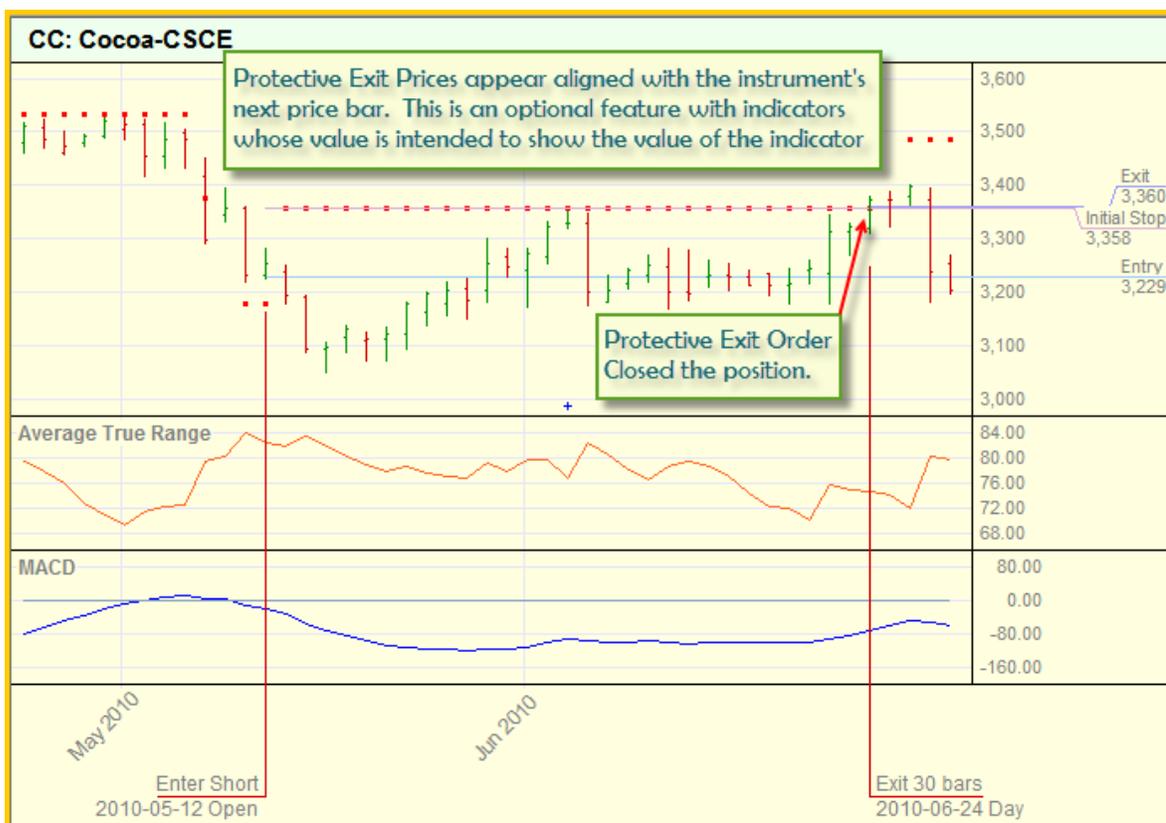
OK Cancel

Active Order Protection

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 is 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:



Active Order Protection

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.

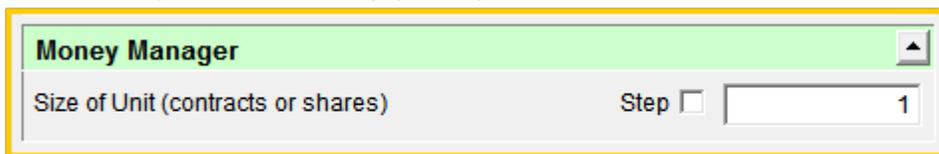
Basic Money Manager:

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 contract. 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

Order Sizing

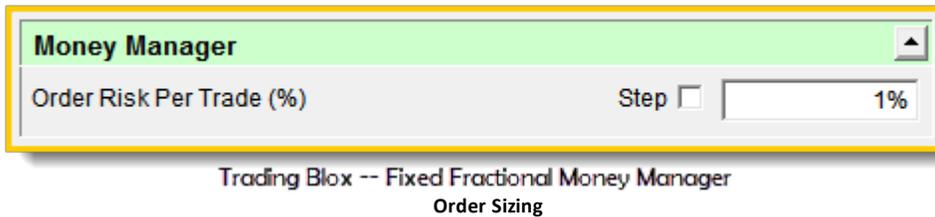
Fixed quantity sizing is often used because it is simple. User picks a quantity, enters that number into the parameter and that is the basis for determining the multiple of the price point difference valuations. While that approach is simple, it doesn't provide the trader with the ability to fix trade 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.



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 x 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 by 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
' 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 Eqt: " + AsString( riskEquity,2 ) _
    + " Order-Risk: " + AsString( dollarRisk, 2 ) )
ENDIF

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

```

Note:

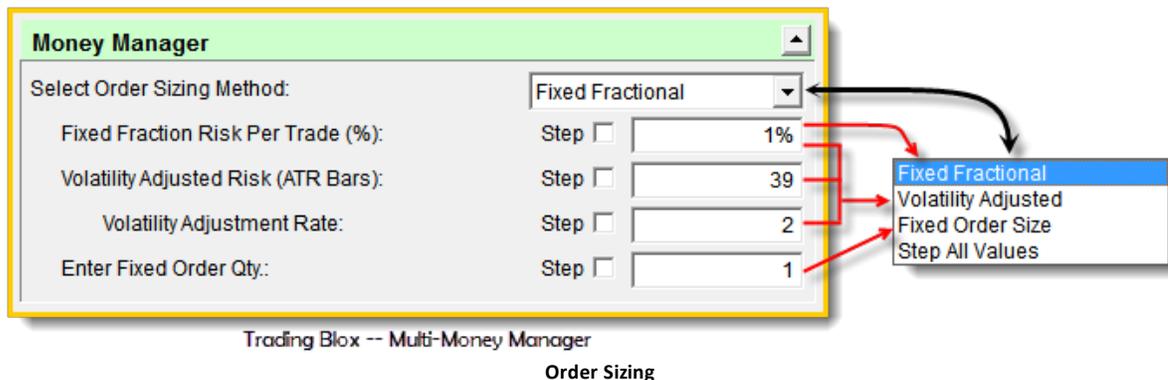
Contracts and Shares must be rounded down to an integer value.

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.



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 %
value.
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
Dollar-Risk
  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.

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 447

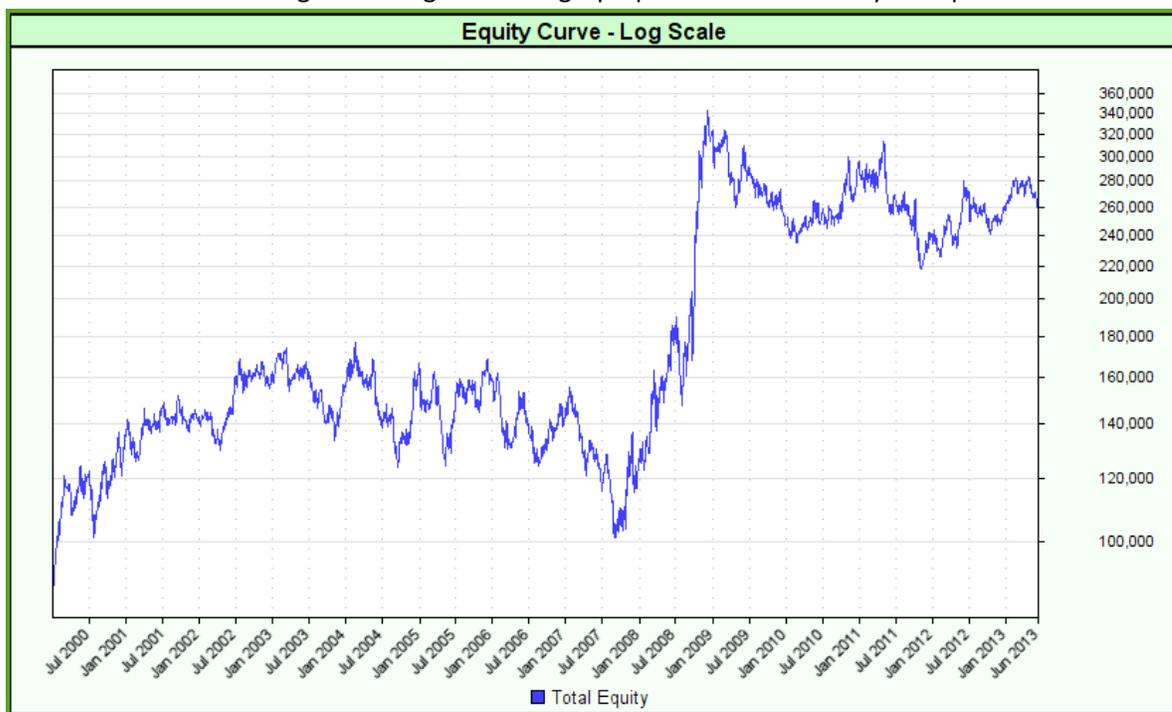
3.7 Trading Risk

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.

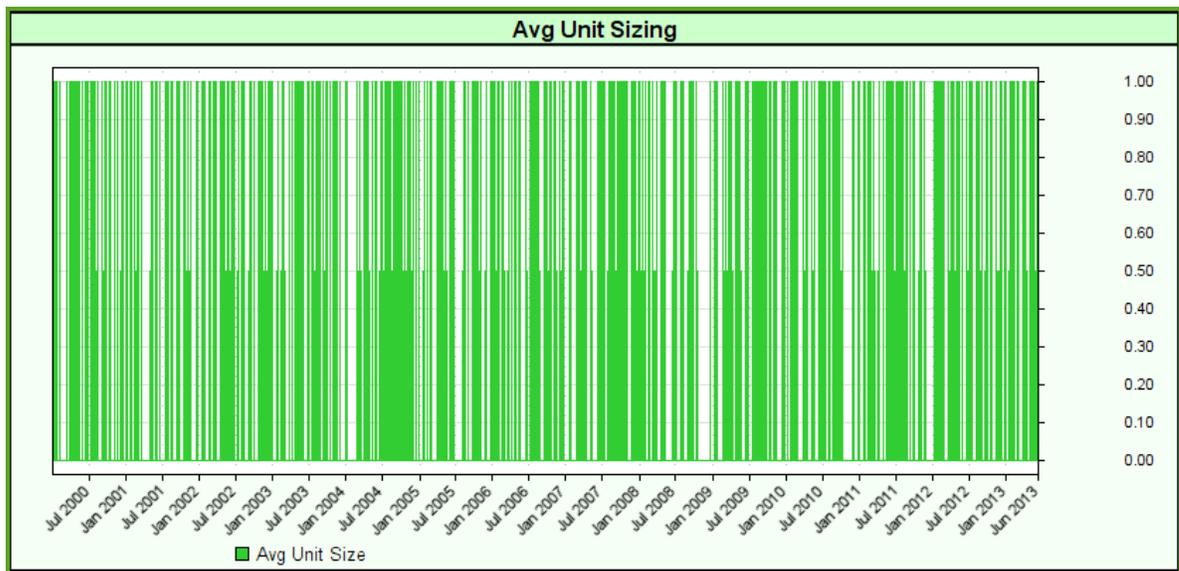
To demonstrate fixed quantity we will use the Basic Money Manager and set the quantity parameter to 1 contract or share, and we will test the trading system from the beginning of 2000 to the current data download date.

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



Trading Risk - Basic Money Manager Fixed Order Size Graph

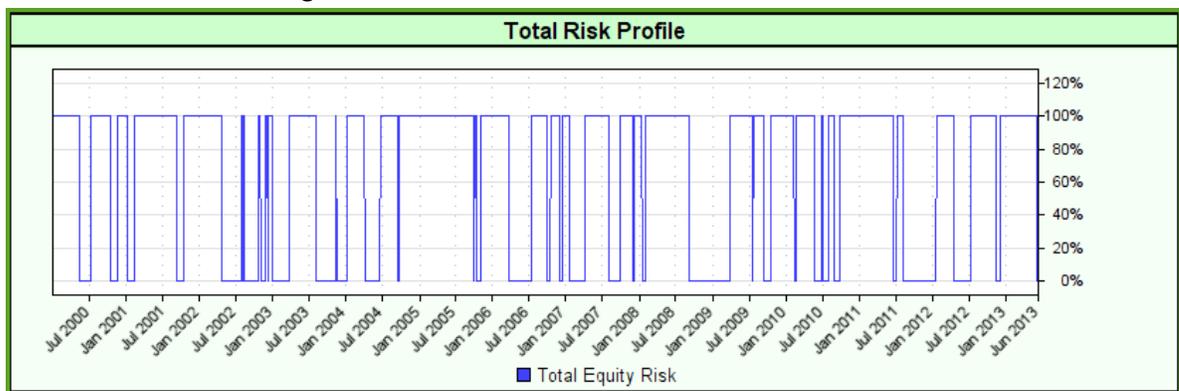
During testing this custom graph created to show the static nature of how the order size of the Basic Money Manager consistently applies the same quantity regardless of how much growth appears in the equity profile graph:



Trading Risk - 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.

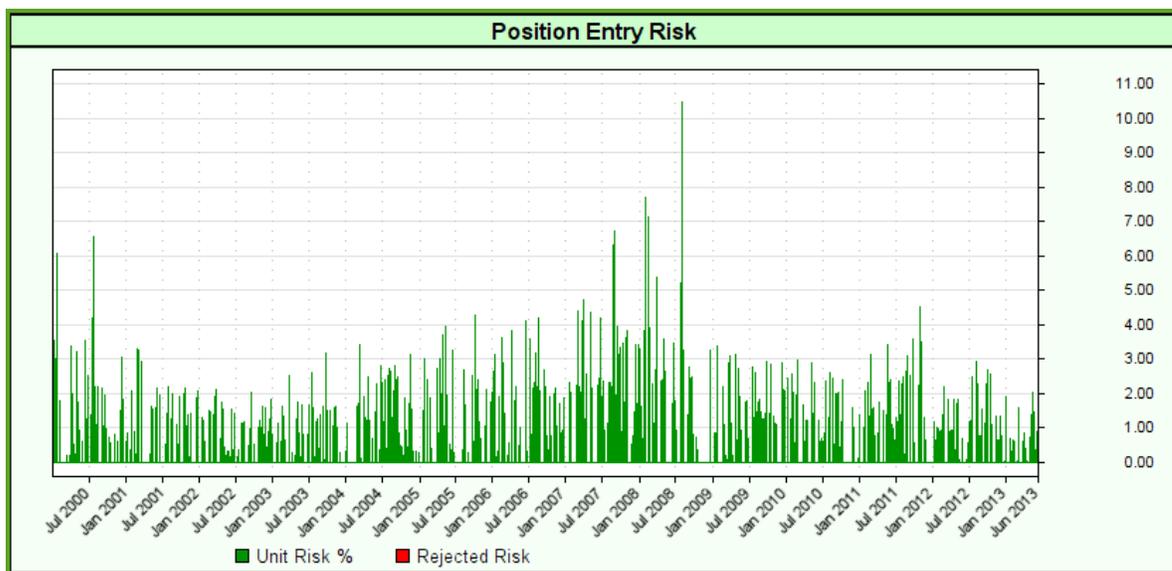
In the standard Performance Results Summary report when the Total Risk Profile option is enabled in the Preference section, the reporting will display an estimate of how much risk that system is creating at all the trade days over the period of dates selected. Position risk is determined by the system's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



Trading Risk - 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.



Trading 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:



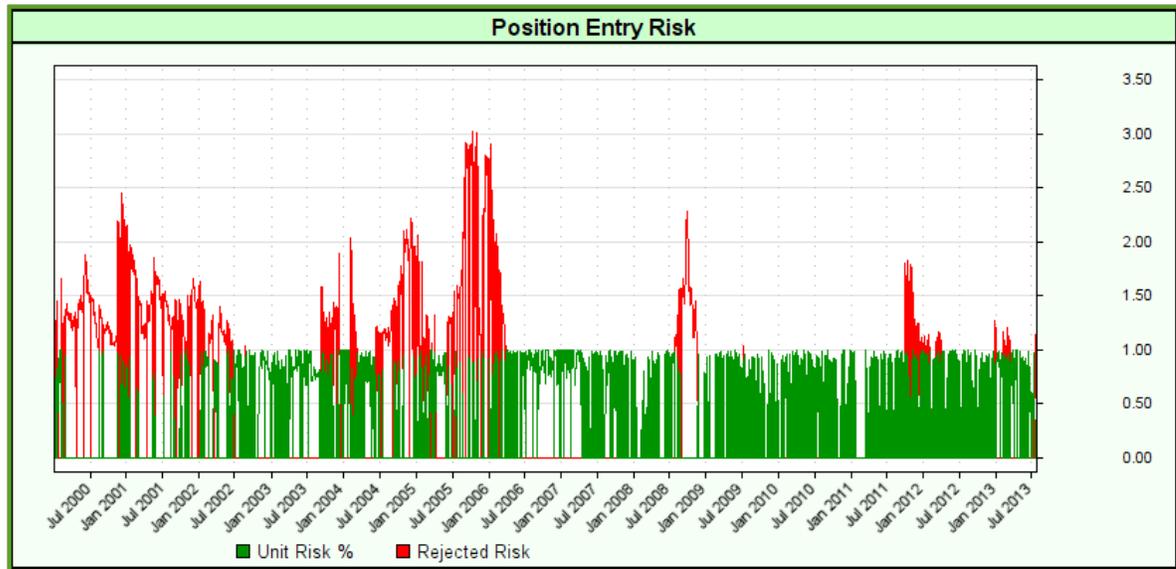
Trading Risk -
Basic Money Manager Size = 1 with Protective Exit Orders

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 for an account valued at 100,000, a 1% risk rate will allow a risk 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.



Trading Risk - Position Entry Risk

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 (see red line spikes). Fixed rate sizing is a method for limiting how much risk a position is allowed to apply to a system, it also enables 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 when the division by the entry risk amount into the allocated amount isn't always be an even number. Results from calculations for size are rounded down to the nearest integer. The variations in the actual entry risk values 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

The account risk of each position is best handled when each position risk is managed to limit the total risk 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 that enable a trading account to lose all or most of its money, or cause the trader to lose confidence in a system.

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. It is important because draw-down periods are going to appear at some point, and long draw-down periods are hard on the account and the trader's confidence in the systems. This means the total risk level a trader exposes to their account must be kept low enough that hard times don't drain the account, or destroy their belief in the system. Instead, it provides a practical level of controlled risk during up and down market cycle period that enhance the trading 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.

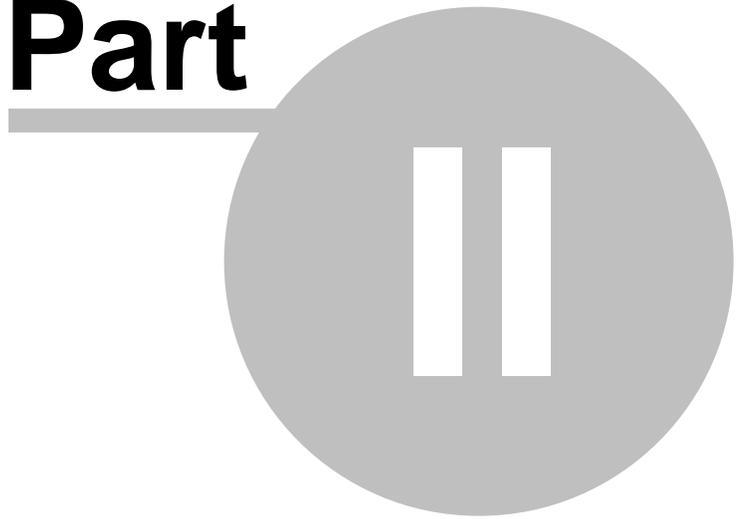
This completes this topic.

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 425

Trading Blox Architecture

Part



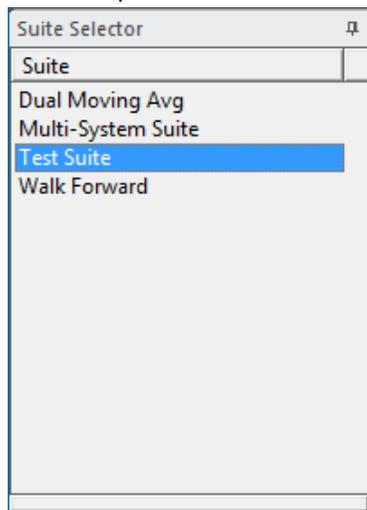
Part 2 – Trading Blox Architecture

Suites:

A test Suite is a collection of systems, the system parameters, and the suite parameters. Let's say you were testing a Dual Moving Average system along 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 a suite. Changes are saved automatically.

Suppose you wanted to work on a new system without disturbing your existing settings. You could create a new test suite and select the desired system. Later by simply switching to the "Dual MA and RSI" suite you can get back the original settings.

Suites can always be seen in the upper left corner of the Trading Blox Builder main menu screen. Controls available to Suites will create new suites, copy or rename existing and remove suites. Suite structures can be locked, and unlocked. Locking prevents parameters from being changed. It does not prevent the blox or systems from being changed.



Suite Name List Examples

Systems:

A system is a collection blox. Each block contains indicators, parameters, variables, and scripts. If you own the Basic 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. Traders who have Trading Blox Builder Builder versions can create your own Blox.

Trading Blox Systems Types:

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

System Component:	Corresponding Block Type:
How Much to Trade	Money Manager
When to get out	Exit Blox
User ideas	Auxiliary

Blox Modules:

Blox are system blocks that encapsulate trading script ideas. Most of the Blocks are self-contained parts of a trading system designed to be connected with other Blocks 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 Blocks. 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 Block module, a package is created that can easily be linked to one or more systems that need the trading idea contained within the Blox. Blocks are trading objects, and while these objects only need to be created once, they can be used many times by other systems to simplify the creation of different system methods.

The same Blox can also be used in multiple systems at once because the parameter setting values of the system's blox are restricted to the system they are assigned. Blox are one of the most powerful features of the Trading Blox Builder versions because it support the concept that makes features created in one blox to be used in many systems at the same time.

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.

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 where the rules of the system method of that Blox are stored. Systems are built by selecting newly created Blox and/or using already available Blox into a system definition file.

When you define a system to contain certain Blox, the functionality contained in those Blox is what defines what the system method. 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).

In the sections that following you will learn how to create systems from Blox and Scripts:

- [Working with Systems](#)
- [Working with Blox](#)
- [Blox Name Changes](#)
- [Working with Scripts](#)

Script Section List:

This list contain the names of script sections where scripted code can be entered. Some [Blox Types](#) will enable instruments by their default nature, and some will require a script that needs to access an instrument to use the [LoadSymbol](#) function to access an instrument.

Add Script

Please select script type to add to block:

Before Simulation
Set Parameters
Before Test
Before Instrument Day
Before Trading Day
Before Bar
Exit Orders
Exit Order Filled
Entry Orders
Entry Order Filled
Can Place Order
Unit Size
Can Add Unit
Can Place Order
Can Fill Order
Filtered Order Notification
After Instrument Open
After Open
Before Close
Update Indicators
Before Order Execution
After Bar
Adjust Stops
After Instrument Day
After Trading Day
After Test
After Simulation
Post Process Utility

Available Script Section Name TBv5.4.1.x

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 653

1.1 Working with Systems

You can use the systems that come with Trading Blox, and you can get other systems and import them. 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. In addition, you should be familiar with this section before a modification to an included system, or a system you have purchased from another source.

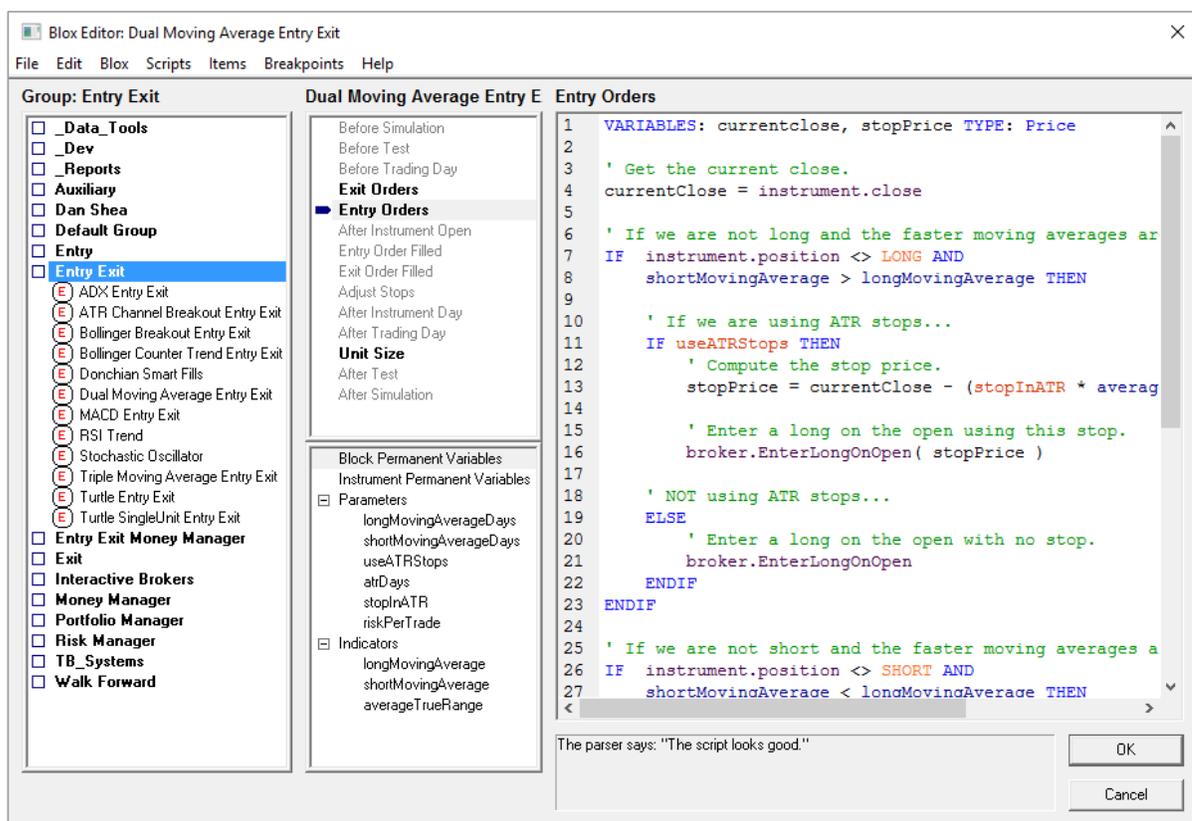
What is a system?

Systems are a collection of Blox. There are six Block types in Trading Blox Builder. Each blox type has a particular purpose in a trading methodology. As described earlier, these blox are basic components of a trading system. This table names the the blox types and the main reason for that blox type:

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
User ideas	Auxiliary

The System Editor:

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



Trading Blox Basic Blox Editor

New System:

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

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

Some of the lists accept multiple Blox, and some lists can accept only one Block type. 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, it cannot be recovered unless you backup your Trading Blox Builder files on a regular basis. When a backup isn't available there is some good news. Recreating a system is

easy because a system is only a collection of Blox. This means, to recreating the deleted system is as simple as creating a new system and then adding the required Blox to the new system. It is still good practice to back up your systems on a regular basis. All Trading Blox Builder systems are stored in the installation folder named, "Systems."

Note:

When deleting a system

When a System contains a Block that can only have one Block of that type, you must remove the installed Block before adding a replacement version. Many of the block in a system can have multiple addition. Review the details in [Blox System Placements](#).

To delete a Block from the system, select the System name, select the Block within the System, and click "Remove Block from System." That action will only delete the highlighted block from the list of selected blocks in the selected system. You will not be removing the selected 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**. Both the Entry and Exit Block need to call the [Broker Object](#) to enter and exit trades. When a Broker Function creates an Entry Order, the Money Manager Block is automatically call to set the trade quantity for the newly created entry order. 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, place that file in your Import folder. When you start-up Trading Blox again, 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.

To understand the timing of Global System scripts, review the details in this topic: [Global Script Timing](#)

1.2 Working with Blox

The **Blox Basic Editor**, explained in more details in the **Trading Blox Builder User's Guide**, provides total control over the creation, edits, renaming and the deletion of any blox module.

The purpose of a blox is to influence, or add one or more behaviors to the system where it is included in the System's list of blox modules. When you create a blox to provide logic to a system, any other system where that blox is assigned will have its behavior changed.

Blox modules are a very powerful convenience in computing terms. It convenience is made possible because it can be "Built once, but used many times." This flexibility means that when a blox is modified all the other systems where this blox is included in the system list of blox, will inherit the abilities that blox provides.

When you only want the blox you modify to have an affect on a specific system, make a copy of the blox before you create any changes. This step will allow you to change the name of the modified blox without it affecting any of the other systems where the original blox logic is assigned. When you change the name of the blox, the previous blox name will still be in the system list of modules. To add the modified blox, remove the previous version and then add the changed versions.

Being able to create a blox once, and then use it in many systems, provides a lot of flexibility in creating systems designs. It also allows you and others to create a blox and offer it as shared blox in the [Blox Marketplace](#) section of the [Roundtable Forum](#).

To understand all the features and abilities of the **Trading Blox Builder's Blox Basic Editor**, review the information in the **Trading Blox Builder User's Guide**.

"Defined Elsewhere in Another Block" option :

Block Permanent Variable

Script Name

Display Name

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

OK

Cancel

Any variable "Defined Externally in Another Blox" Option

Almost all blox data variables are scoped to be entirely independent of other blocks. This variable independence is created when a variable is created using the Scope setting [Block](#). Where this independence isn't true, is when a blox is created with a variable that has a Scope setting of System, Test, or Simulation. These alternate scope options enable the data to be accessible outside of the block where it is created, and a different block uses the same variable name and then enables the **"Defined Elsewhere in Another Block"** option (see above image).

The data in the variable using a Scope setting of System, Test or Simulation can be accessed by the block that has created a variable with the same name and enables **"Defined Elsewhere in Another Block"** option.

This data option item enables the **"Defined Elsewhere in Another Block"** definition option, only the type setting will be displayed in the blox where the data item is named. Blox that have this option enabled must also have another blox in the same system that provides the named data item's scope setting and any default initialization value required.

Data items that use the **"Defined Elsewhere in Another Block"** option, create a dependency that requires another blox to be in the system so the data item's information can be fully referenced for the test to operate successfully.

When the dependent blox isn't included, Trading Blox Builder will stop a test and display a message similar to this one at the bottom of the data window that appears:

"I can't run the '<Any Script Name>' script from the Trading Block '<AnyBloxName>'. The parser can't understand line 1 because variable '<AnyVariableName>' is undefined in this context."

This message is reporting the **"Defined Elsewhere in Another Block"** data item hasn't been fully declared in any of the blox listed in the system being tested.

For more information about the **"Defined Elsewhere in Another Block"** option, review the information in the [Data Scope](#) topic.

Script Sections:

Each blox can have one, or many script sections. The names of the script sections in a blox varies by [Blox Type](#). All the standard scripts section available and their execution timing is explained in this [Blox Scripts](#) topic. [Custom Scripts](#) can be added and called from a standard blox script. Detailed information about Custom Scripts and their functions, is available in the [Script Object](#) section.

Script sections in three of the blox types, will create a blox limitation. That limitation restricts the system to have only one of each of those block types in a system's structure. See the information in [Blox System Placements](#) for more information about how blox can be added to a system.

Script sections with scripted statements become active when statements are entered into their section. Script sections without any statements become inactive and are not sequenced during a test.

Scripts that contain statements control how a blox script section will perform during a Trading Blox Builder suite test.



Data Scope Range Information:

All script sections can support local variables. Local variables are added to a script section when they are declared in the script section where they appear. Local variable scope is limited to the script in which it is declared. This means that the information in a local variable is unavailable to the same name local variable declared in a different script section.

Here is how a local variables can be added to a script section:

Local variables are automatically set to Local to the declared variable's script section.

Local variables are not reset or cleared automatically.

Assigning values to a Local Variables must be handled by scripting statements.

```
' Local Variable Declaration Examples:
VARIABLES: iStepNum, iTotalSteps Type: Integer
VARIABLES: trendRate, trendGain Type: Floating
VARIABLES: sTextMsg, sFileName Type: String
```

Local variables are often used to simplify the creation of a working variables that will have no use outside of the script section where they appear. All local variables need to be cleared, or reset to a default value ahead of the time in which they are used if their previous information might contaminate the results they create.

For example, when a statement section needs to count something, and it steps through a repeating loop structure that counts items, if the local counting local variable at the start of the counting process is not zero, the value of the in that local variable incrementing the value of items counted will not be accurate unless the value at the start of the counting in the local variable is zero.

Resetting local variables is simple. It should always happen before they are used in that script section, unless their previous value is needed in that specific script section. Placement of the variable clearing statements is usually right after where the local variables are declared, or at least before they are used for the first time in the script's section:

```
' Resetting/Clearing of local variables:
iStepNum = 0           ' Integer
iTotalSteps = 0       ' Integer
trendRate = 0.0000    ' Floating
trendGain = 0.0000    ' Floating
sTextMsg = " "        ' String
sFileName = " "       ' String
```

All variable types can be used with functions and properties as long as the variable type will have context in the script section where it is used. For example, when an **IPV** variable is needed in a script section where it doesn't have automatic context, like the Before Trading Day script section, access to an instrument can be made possible using the [LoadSymbol](#) function.

Scope Range:	Variables:	Descriptions:
Local	Local - Script Declared Only	Local variables are restricted to a local script scope. Their information is limited to the script section where they are declared.
Block	BPV, IPV, Parameters, Built-In Indicators	Block scope means that the variable's information can be accessed, or changed anywhere in the blox where the variable has context.
System	BPV, IPV, Parameters, Built-In Indicators	System scope allows the variable's information to be accessed or changed anywhere in the system where that variable with the same name and scope is declared.
Test	Test, Parameters	Test scope allows the variable's information to be accessed, or changed anywhere in the test where a variable with the same name and scope is declared.
Simulation	BPV, IPV	Simulation scope is Test Scope, but the values in the variables are retained throughout the simulation.

Blox Basic Editor:

Creating, Editing, Copying and Deleting a Blox is performed in the Trading Blox Basic Editor. A full description of the editor is available in the <%APPLICATION_NAME%> **User's Guide** topic -> **Blox Basic Editor** topic.



The Trading Blox Builder Blox Basic Editor is accessible in the Builder Editions by pressing F4 or by the Basic Editor menu option in the Editor section of the main screen:



When the Blox Basic Editor appears, any of the listed blox can be changed in the editing area on the right. New blox can be added, copied or removed using the menu options at the top of the Basic Editor.

Editing Blox Scripts:

Before editing a standard blox installed during a setup or update, it is best to make a copy of the Trading Blox Builder installed blox before you make any changes. Blox copies must have text that is different from the original blox. A difference could be a simple number or work added to the original blox. This name difference will prevent a full installation, or an update installation that contains the name of the original blox from replacing the blox you changed.

To create a new Blox module, click on the Blox menu and then select the new Option:



When you click "New" this dialog comes up:

New Blox Creation Dialog

Enter a name for the blox, and then select the required [Blox Type Options](#) option.

Before clicking on the OK button, decide if you want all the default script section, or just the required script section for the selected [Blox Type Script](#). When the above decisions have been made, click the OK button to create the blox module file.

Making Blox Script Changes:

Before Deleting or changing the script in any of the blox module, use the Blox menu System option so you can see where the selected blox is being used. When a blox is assigned to one or more systems, this menu option will display the names of the systems that will be effected by changes or destruction of a blox module.



When a blox is assigned to a system, Trading Blox Builder will not allow you to delete that blox. Instead it will display this warning message letting you know why the blox can't be deleted:



Clicking on the Blox Renaming option will display this dialog:



To copy an existing blox, click on the Copy option in the [Blox Menu Options](#). When the dialog appears, select 'Copy' option and then change the name of the blox so that it is different from the original blox name:



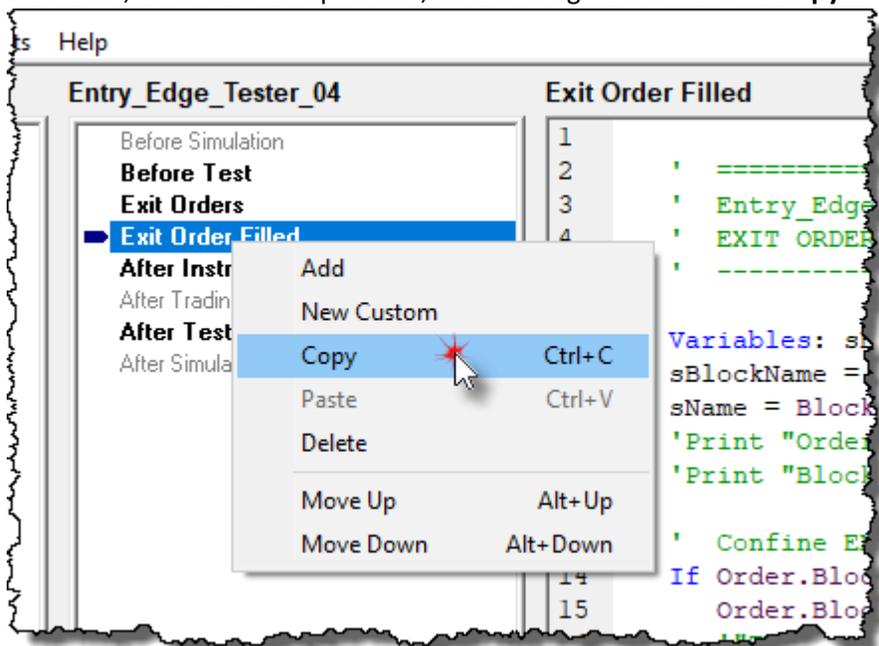
Once you have new Block created, you can edit, or place information in the different scripts by first clicking on a script name, and then add or change the code needed.

Copying Script Statements and Sections:

Script statements can be copied from a block and pasted into a different block by using **Control - C**, to copy the selected statements, and then using **Control - V**, after clicking on the block's section editing area when you want to paste the script you just copied.

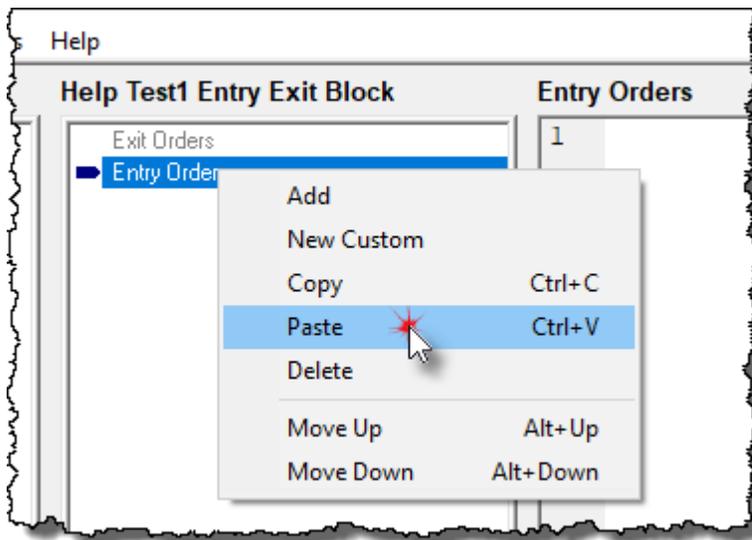
When any script section that is not already part of a blox where you want that script section to be available, and you have a blox with that script section that already has the script statements you want in a block, the script sections can be copied from where you have it, and then pasted into the block where you would like it to appear.

To do this, click on the script name, and then right-click to select **Copy - Ctrl+C**:



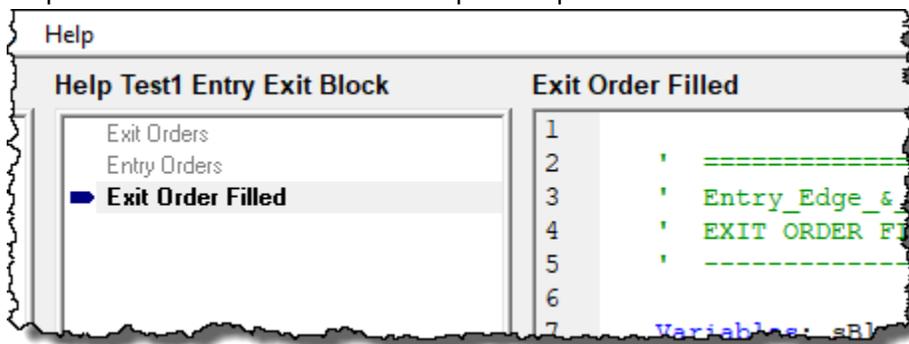
Script Section Name & Contents Copy

To add the copied script name and its contents, click in the destination block's script list display area. Right-click to select the **Paste - Ctrl+V**:



Script Section Name & Contents Paste

After the script name appears in the destination block area, the script code area will display the scripted statement contents of the copied script.



Script Section Name & Contents Paste Results

Custom Script Sections:

You can also add new custom scripts which can act as [custom functions](#). [Custom Functions](#) are designed to create specialized capabilities that are created by the user. In these user defined blox the user can create functions that aren't part of the installed functions. A good reason to create a Custom Function to create a function that is likely to be used by other systems. When this need occurs, the ability to add a function that is written once, but can be used many times adds capability without the need to recreate that scripting process again.

How can you learn more?

Starting from scratch as a beginner, you might find no programming experience daunting. It will create some confusion, but if you fiddle with what is available already by trying small changes, you'll get a feel for what works, and what else you might need to look up to be successful. There is also a lot of help to get you started and through the confusion. Be sure to register with the Trading Blox Builder online [Traders' Round-Table Forum](#). Forum has a huge amount of information and a lot of blox examples in the **Trading Blox Customer Support** sections: [Trading Blox Support](#), [Blox Marketplace](#), and [Feature Requests](#) groups.

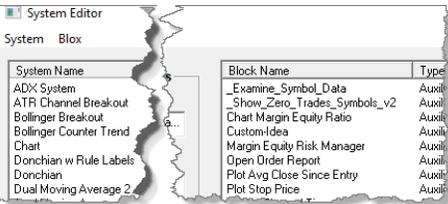
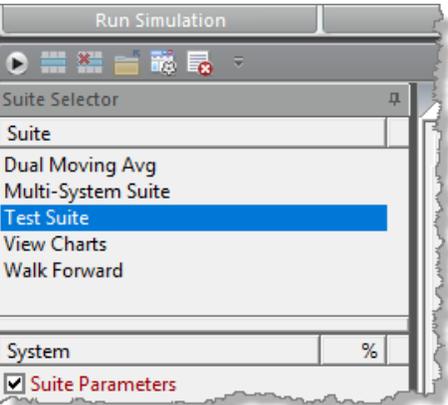
Edit Time: 3/21/2024 10:41:42 AM

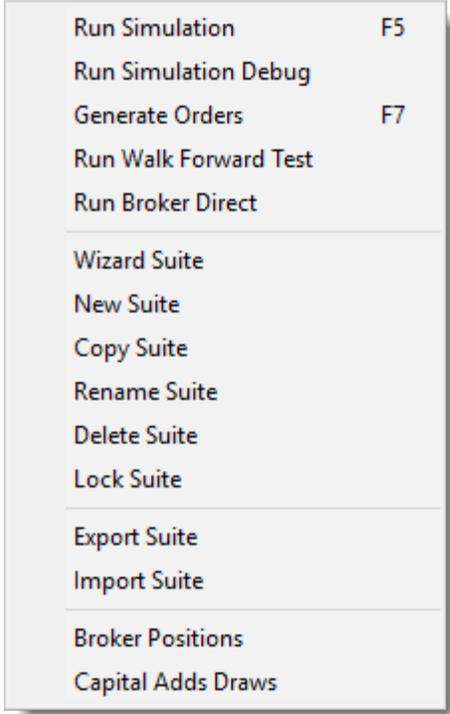
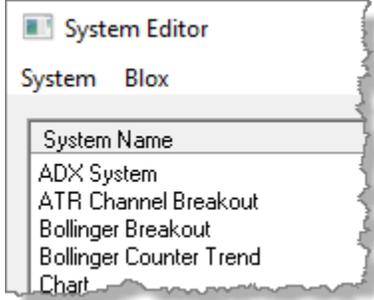
Topic ID#: 651

Blox Name Changes

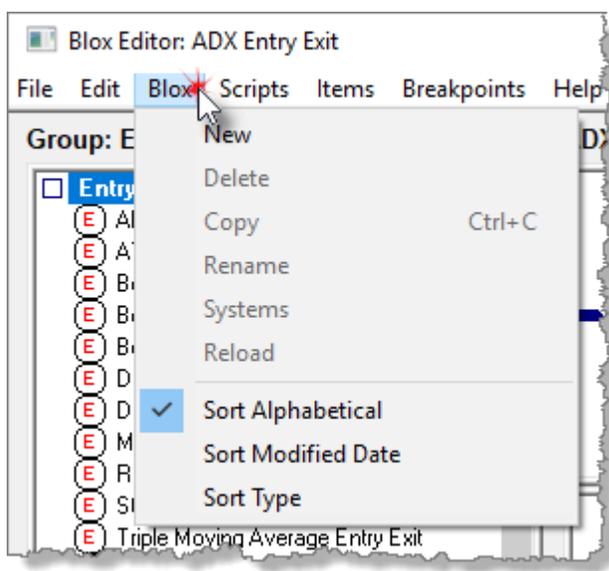
Naming Blox Modules:

The names of a Suite, System and Blox can use any alphanumeric characters. They also contain a space, a dash "-", and an underline "_" symbol as part of the name.

Modules:	Example Image:	Description List:
Blox	 <p style="text-align: center;">System Editor Blox Area List Example</p>	<p>Once a blox is created it can be added to a system module. Available blox are listed in the System Editor's Block Name area.</p> <p>Clicking on a block name, and selecting the Add option, will place a block in the list of blocks in the system.</p> <p>Blocks are also displayed in the Trading Blox Builder User's Guide Topic: Blox Basic Editor. Where all the options are explained.</p>
Suite	 <p style="text-align: center;">Suite Name List Area Example</p>	<p>Suites are only displayed on the main screen of Trading Blox Builder. Suites provide access to the systems that are selected to be in a suite.</p> <p>Suite changes, like the creation, renaming, or deletion of a suite is all done using the Suite menu options (Right-Click Mouse):</p>

Modules:	Example Image:	Description List:
		 <p style="text-align: center;">Trading Blox Suite Menu Options</p> <p>To display the menu, right-click anywhere in the Suite list display area.</p> <p>More information about Suite's list options is in the Trading Blox Builder Global Suite System and User's Guide Topic: Suite Selector List topic.</p>
<p>System</p>	 <p style="text-align: center;">System Name Area List Example</p>	<p>All the System modules options are explained in the Trading Blox Builder System Editor and User's Guide Topic: System Editing.</p>

Blox Name, Group And Section Fields:

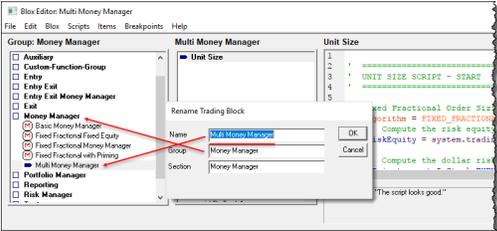


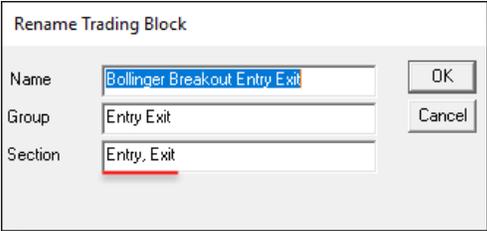
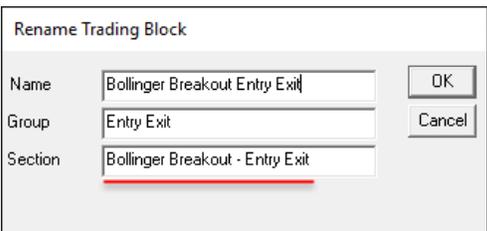
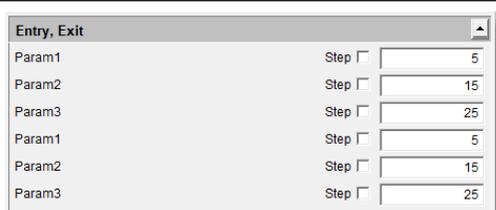
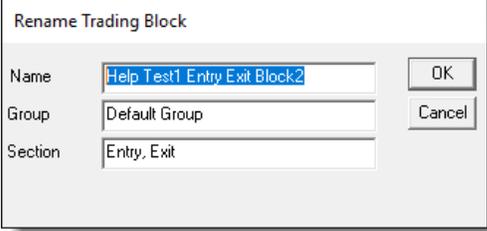
Trading Blox Basic Editor's Blox Menu Options

There are two different dialogs that allow access to the all three of the blox information fields.

Blox Fields:	Description:
<p>Name:</p>	<p>This field is the blox file name. The name of a block is entered in the Type field section of a block. It is also displayed in the System Editor's list Name Column. Blocks listed in the name column represent all the blocks available for a system assignment.</p> <p>When a block is added to a system, the system where they are added will execute the script section in the assigned blox. During execution of blox where the script sections have same name script section names in other blocks, the script sections in will sorted into ascending alphabetical Blox name order. The sorted order of system's blox name determines the order in which the scripts of the same name in a system are executed. To change the order of their execution, change the name of the blox.</p> <p>Blox names are unique and restricted to one block with a specific name. To copy a block in a different group where the original will be left in its current group, the name of the block must be changed in some way. Any change to the name is acceptable. For example, if the block is named "ADX Entry Exit", and a copy of the block is wanted into ADX System Group, the copied block can be "ADX Entry Exit2" or "ADX-Entry-Exit."</p>
<p>Group:</p>	<p>The Blox Group field is assigned when the blox is created. The group name highlighted at the time the user creates a new blox, Trading Blox Builder will be the same name as the current blox that is selected. If the opened area of a group name is selected, the new blox will be save to that group list display, unless the user changes the Group name.</p>

<p>Blox Fields:</p>	<p>Description:</p>	
	<p>Block Group names can be changed using the Trading Blox Builder Blox Basic Editor's Blox Menu Rename option. It can also be changed using the Copy option to duplicate and change the name of a blox. When the Group name is changed, or when the mouse drags the block from its current group to a different group, Trading Blox Builder Blox Basic Editor will automatically change the the new Group name to the destination group name.</p> <p>Group names can be any group name that exist, or user wants to have in their list of Groups. Keeping blox designed for a specific system in the the same group can make finding the blocks used in a system easier to locate.</p> <p>Information about the standard Blox Types is available in the Trading Blox Architecture table.</p> <div data-bbox="435 781 1247 1402" style="border: 1px solid black; padding: 10px; margin: 10px 0;"> <p style="text-align: center;">New Trading Block</p> <p>Name <input type="text" value="New Trading Block"/></p> <p>Group <input type="text" value="Entry"/></p> <p>Section <input type="text"/></p> <p>Block Type</p> <ul style="list-style-type: none"> <input checked="" type="radio"/> Entry and Exit <input type="radio"/> Money Manager <input type="radio"/> Portfolio Manager <input type="radio"/> Risk Manager <input type="radio"/> Auxiliary <p><input checked="" type="checkbox"/> Include Default Scripts</p> <p style="text-align: right;"> <input type="button" value="OK"/> <input type="button" value="Cancel"/> </p> </div> <p style="text-align: center;">Blox Group Types</p> <p>The group is just used to group the blox in the listing on the left of the blox editor. There ability to group block in the list provides a flexibility for the user to organize the modules by block type, or system type, or dates, or customer/clients, etc.</p>	
	<p>Blox Basic Editor Dialog Display Information</p>	<p>Blox Parameter Editing Display Information</p>

Blox Fields:	Description:	
	 <p style="text-align: center;">Blox Basis Editor - Blox Name Details</p>	
	 <p style="text-align: center;">Blox Renaming Dialog</p>	
<p>Section:</p>	<p>Newly created blox will appear with the Section field empty. Earlier versions of Trading Blox Builder, placed the group name into the section field. The section field is now the information listed above the blox parameter area. When more than one blox from a group is added to a system, and it has the same information in a section field, only one instance of the section field is displayed. By replacing, or adding information into the section field that helps the user understand which blox the following parameters are representing, the process of adjusting parameters values is easier to understand.</p> <p>Users might find have more information</p> <p>Previously, the Group name information was automatically entered, but now it is left up to the user who creates the block to decide what information they want to see displayed with the system parameters.</p> <p>The section word or phrase can distinguish which of the Entry and Exit blocks the parameters below the section name are describing. displayed in the Section field determines the words displayed when the system's parameters are displayed. Each block in a system can show the Group name, or it can be more informative and display what the block's create decided was better to inform the user of the block.</p> <p>by the blox when it is in a system, and that system's parameter settings are displayed on the main parameter edit area.</p> <p>The section name is also very flexible and just provides a way to identify the user interface area (section) that the parameters will be in when a system in a suite is selected to show its parameters.</p>	

Blox Fields:	Description:	
	<p>You can group parameters from multiple blox into the same section or have each block have it's own section. How this field is used is mostly up to how the user wants to mange and display the blocks.</p>	
	Blox Basic Editor Dialog Display Information	Blox Parameter Editing Display Information
	 <p>Typical Blox Section Information</p>	 <p>Typical Blox Paramter Display Information</p>
	 <p>Renamed Blox Section Information</p>	 <p>Renamed Blox Section Parameter Display Information</p>
	 <p>Duplicate Section Name Heading Block_1.</p>	 <p>System Parameters Duplicate Section Names Heading Example.</p>
	 <p>Duplicate Section Name Heading Block_2.</p>	<p>Note:</p> <p>When the Block's section names are the same, there is only one Heading title to identify each of the block's parameters.</p>

Blox Fields:	Description:	
	<div data-bbox="435 317 922 548"> <p>Rename Trading Block</p> <p>Name: <input type="text" value="Help Test1 Entry Exit Block"/> <input type="button" value="OK"/> <input type="button" value="Cancel"/></p> <p>Group: <input type="text" value="Default Group"/></p> <p>Section: <input type="text" value="Entry, Exit - Entry Exit Block"/></p> </div> <p style="text-align: center;">Different Section Name Heading Block_1.</p> <div data-bbox="435 590 922 821"> <p>Rename Trading Block</p> <p>Name: <input type="text" value="Help Test1 Entry Exit Block2"/> <input type="button" value="OK"/> <input type="button" value="Cancel"/></p> <p>Group: <input type="text" value="Default Group"/></p> <p>Section: <input type="text" value="Entry, Exit - Entry Exit Block 2"/></p> </div> <p style="text-align: center;">Different Section Name Heading Block_2.</p>	<div data-bbox="964 317 1435 432"> <p>Entry, Exit - Entry Exit Block</p> <p>Param1 Step <input type="checkbox"/> <input type="text" value="5"/></p> <p>Param2 Step <input type="checkbox"/> <input type="text" value="15"/></p> <p>Param3 Step <input type="checkbox"/> <input type="text" value="25"/></p> </div> <div data-bbox="964 443 1435 558"> <p>Entry, Exit - Entry Exit Block 2</p> <p>Param1 Step <input type="checkbox"/> <input type="text" value="5"/></p> <p>Param2 Step <input type="checkbox"/> <input type="text" value="15"/></p> <p>Param3 Step <input type="checkbox"/> <input type="text" value="25"/></p> </div> <p>System Parameters Different Section Names Heading Example.</p> <p>Note: When the Block's section names are the different, each block will display a heading title for each of the block's parameters where the section names are unique.</p>

1.3 Working with Scripts

The following is a chart showing the most basic list of Blox and scripts that will be in most systems.

Each script is called only under specific circumstances after the instrument has been primed. It can be called for every price record in an instrument file. It can also be called instrument file, or at the end of a time period. For example, a daily record file can call the Week or Month data version to access the last record in the week or last record in the month. When this type access is used to access instrument information the values used can be the basis for how the values returned in an order are applied.

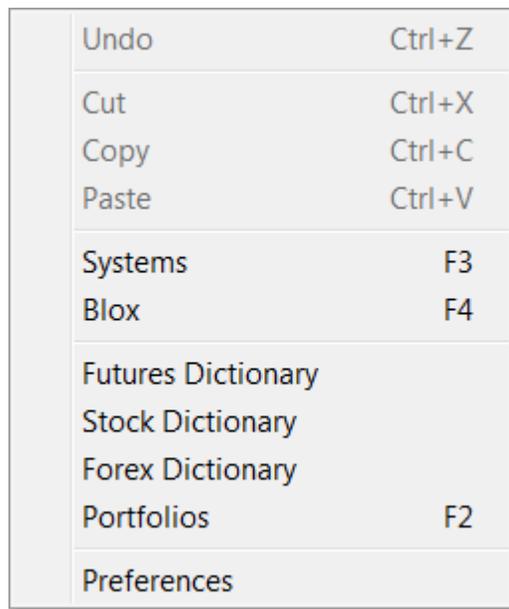
Additional details are available in the [Simulation Loop](#) topic.

Block Type	Script Type	Day	Instrument	Position	Called When
Entry	Entry Orders		•		start of day
				•	
Exit	Exit Orders		•		start of day , when a position is active
				•	
Money Manager	Unit Size		•		Called by broker object function only when an order created

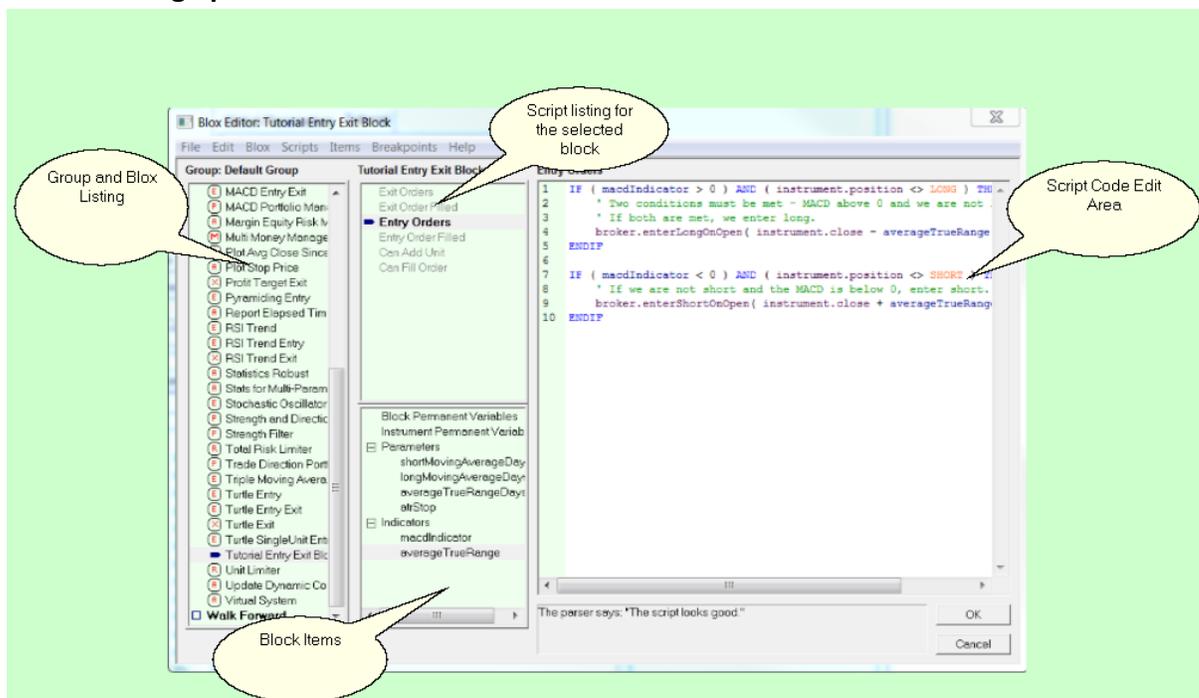
You can build most systems using only these Block Types and these limited number of scripts.

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:



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.

1.4 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				Simulation start
	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				Simulation end
Exit	Before Simulation				Simulation start
	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				Simulation end
Money Manager	Unit Size	<p>Automatically called when any of the broker functions called to initiate a new entry order. Not all initiated orders become broker orders. An order can fail when a portfolio Trade Control Function is enabled to block one or both trade direction orders.</p> <p>An order can also fail the order's quantity order size it too large for the available the available equity. It can also fail when the order's size is smaller than the minimum allowed order size setting.</p> <p>When an order is found to be missing, review the User's Guide topic Order Filtering to understand how orders are filtered.</p>			

You can build very complex and effective systems using only these Block Types and these limited number 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](#).

Last Edit: 3/21/2024

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 161

1.5 Equity Types

System Position Sizing Equity:

All equity methods start out with the same [Test Starting Equity](#) value entered into the [Global Equity Management](#) parameter field. This Global Equity Management section and the adjustments made by the Allocation Slider determine how the system is funded.

Each selectable equity method in this module will create a different result in reported performance. It changes because each equity method applies trade results to each equity method differently. When the Equity Management section is using values in the Draw-Down Threshold and Draw-Down Adjustment fields, those options only effect Trading Equity calculations.

Suite Equity Allocation Distribution:

Table will show the initial equity allocation distribution can be fixed, or it can use trading equity changes made by the other systems in the suite to adjust for equity gain and drops in other systems over the testing period.

Equity Types:	Equity Descriptions:
Trading Equity: ^Top	<p>Amount available for trading by the current system.</p> <p>Trading equity is determined by the Global Setting parameter "Trading Equity Base," System Allocation Slider position, Leverage Rate, and the Draw Down Reduction control settings.</p> <p>For example, if Trading Equity Base is set to "Total Equity", then the value of the system's Trading-Equity is equal to the test.totalEquity multiplied by the Leverage Rate multiplied by the system's Allocation Percentage slider. When Draw Down Reduction controls are used the equity change causes the reduction to reduce, the value made available will be reduced by the reduction threshold amount.</p> <p>For order generation when a value has been entered for Order Generation Equity this number replaces the test.totalEquity amount so the trading equity is calculated from there.</p> <p>Systems in the same suite using Trading-Equity share the gains and losses of other systems in the same Suite.</p>
Total Equity: ^Top	<p>This equity series starts out as system.tradingEquity and is then only affected by the total profits, or losses of trades created by the system where it is being used.</p> <p>For order generation this is the Order Generation Equity times the allocation. Note that this equity value amount is determined after the close of the prior day.</p>
Closed Equity: ^Top	<p>Closed equity is determined after the close of each day and only represents the value of closed trade results up to that date.</p>

Equity Types:	Equity Descriptions:
Core Equity: ^Top	Core Equity is a measure of equity that includes Closed Equity plus the portion of Open Equity that would be realized if all the current positions were exited at their current protected stop price.
Size All Min Qty: ^Top	Manual sizing of orders can create a unit size of at least 1-contract, or the sized entered into the size field of the Basic Money Manager, or any custom sizing method. Fixed quantity sizing is helpful when auditing a system, or a broker's market calculations.

Edit Time: 3/21/2024 10:41:38 AM

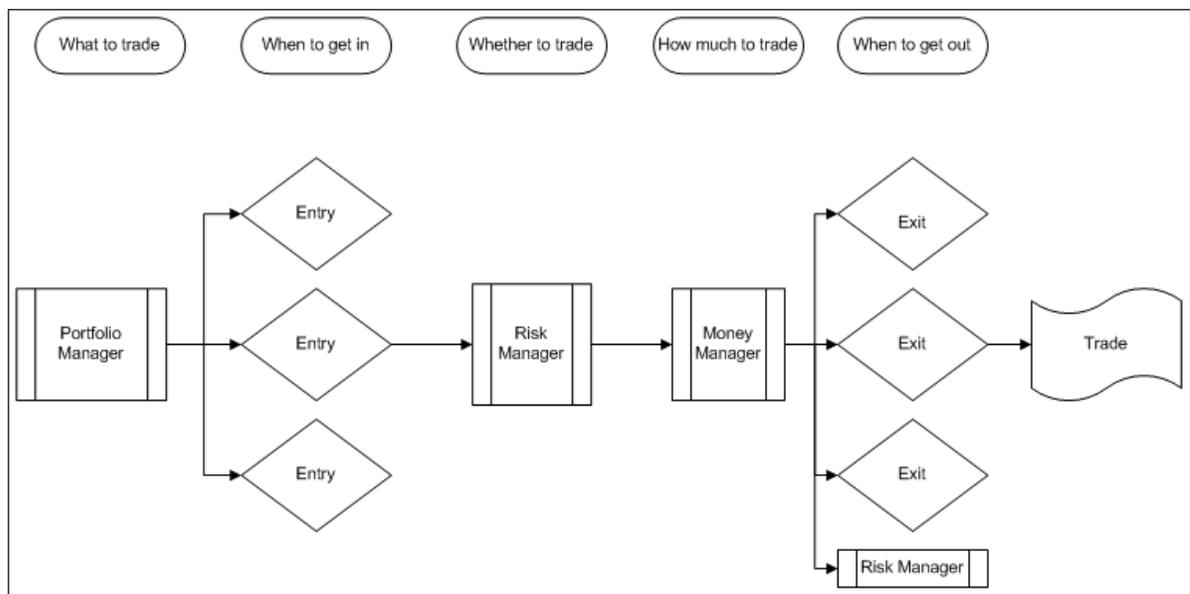
Topic ID#: 304

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.



Trading Blox Process Flow Diagram

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:

```

broker object          call Unit Size script when order is created by
allowed              call Can Add Unit script to check if trade is
                    next ( system )

                    for ( each system )
call Before Order Execution script
                    next ( system )
                    call Before Order Execution script for the Global Suite System, if
available.

                    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

```

Simulation Loop:

```

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

```

Simulation Loop:

```
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

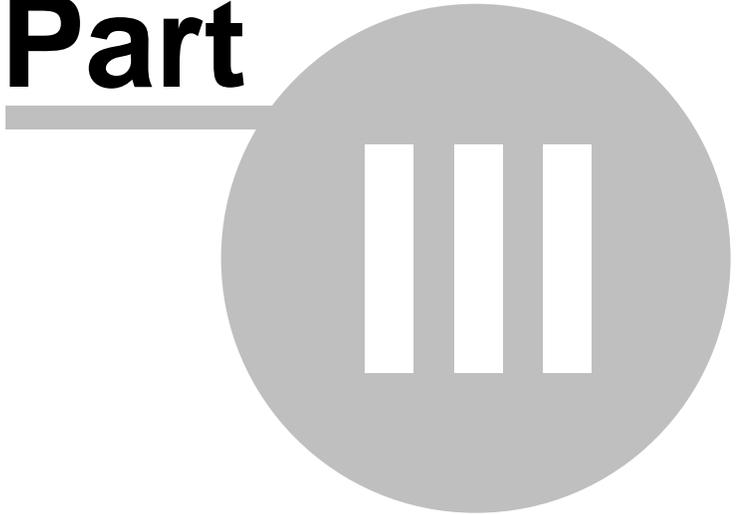
if generating orders
    Generate Orders

next ( test in simulation )

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

Blox Module Reference

Part



Part 3 – Blox Module Reference

Trading Blox Builder has three native modules that are needed to exercise a trading idea:

Trading Blox Modules:	Description:
Suite	Test Suite modules contain the name of one or more systems. Each system can trade the same or different type of instruments. Suites can also contain dedicated test-level Blox designed to work at the top level of a testing exercise.
System	<p>Each system must have a unique name, and a system can function in one of two ways. One way a system can function is to contain a list of modules that can test a system idea. As second way for a system to function is to contain one or more system modules that are designed to work at the top-level of how a Suite test supports one or more of the systems assigned to a specific Suite name.</p> <p>When a System name is shown in the system list in a different color than the other system names shown with black colored text, the different color system name is designated to be a Global Suite System (GSS). Global Script Timing in GSS Blox execute differently than how the Blox are timed to execute. See Script Section Types.</p>
Blox	Trading blox has six Blox Types designed to handle different system responsibilities needed to test most of trading ideas. Blox assigned to a Global Suite System are restricted in which of the possible Blox script sections allowed to execute.

Blox Creation Information:

Each Blox is identified by its names and the type of block that the creator intended. In a directory, the name of the block, can be arranged in ascending or descending order. It can also be ordered by its designed section type. All block will have a last save date, making the sorted order of all block by date ascending or descending order.

When a blox is created, these fields should be entered to ensure the process intended is how the blox will perform:

New Trading Block

Name:

Group:

Section:

Tab:

Block Type

- Entry and Exit
- Money Manager
- Portfolio Manager
- Risk Manager
- Auxiliary

Include Default Scripts

OK

Cancel

New Blox Creation Information

The Blox scripts that are added to a new block can easily increased and reduced. To see which scripts will be added to a block when the option is enabled with a check-mark is available here: ["Include Default Scripts"](#)

During block creation, the full-name of a block, and the Type of block that is intended, are the most important details. A name can be easily changed, but the section type information will need to be changed so the blox can perform in a different manner.

The section type of the blox determines the type of system actions and their sequential test timing during a simulation.

New System

Portfolio Manager

Portfolio Manager Blox

Entry

Entry Blox
Entry Exit Blox

Exit

Exit Blox
Entry Exit Blox

Money Manager

Money Manager Blox

Risk Manager

Risk Manager Blox

Auxiliary

An Auxiliary can be any block that isn't one of the above. This is where Custom Blox will appear.

Add Remove

System Blox Section Placement Types

Links:

[Blox Types](#), [Blox Script Access](#), [Blox Script Timing](#), [Blox System Placements](#), [Global Script Timing](#), [Script Section Reference](#)

See Also:

[Trading Objects Reference](#)

Section 1 – Blox Types

Blox Modules:

Blox are system blocks that encapsulate trading script ideas. Most of the Blocks are self-contained parts of a trading system designed to be connected with other Blocks 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 Blocks. 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 Block module, a package is created that can easily be linked to one or more systems that need the trading idea contained within the Blox. Blocks are trading objects, and while these objects only need to be created once, they can be used 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	<p>This block is the script section is positioned to be the best place to create Entry Orders.</p> <p>Note: This Entry-Type name doesn't appear as an option when a new blox is being created, but any Entry, Exit blox create where the Exit script is removed will be an Entry Block.</p>
Exit	<p>This script section is only called when a position is active. It is also the best place to create orders to Exit Orders designed to remove an entire position, or remove a some of the quantity in an existing position.</p>

Script Name:	Description:
	Note: This Exit-Type name doesn't appear as an option when a new blox is being created, but any Entry, Exit blox that exists where the Entry script is removed will be an Exit Block.
Entry, Exit	This block type is an option that can be selected when a new block is being created. It will have both an Entry Orders and an Exit Orders script section.
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, custom function groups, statistics, reporting blox and many other types of process that might be needed in a system.

1.1 Portfolio Manager

The **Portfolio Manager** script section is used to filter the instruments a system portfolio. This is analogous to a screen process used often in stock trading. Each **Portfolio Manager** contains the [Rank Instrument](#) and [Filter Portfolio](#) script sections. These script sections can only be in the a **Portfolio Manager** blox.

A **Portfolio Manager** blox applies its logic to each instruments data record. As each instrument record is reviewed the instrument's ability to create an order can be allowed or denied, and it can also be assigned a ranking value. When an instrument is filtered from trading it can be prevented from being allowed to trade in a Long, Short, or in both Long and Short direction. This directional or bi-directional filtering option is built-in **Trade Direction Portfolio Manager** blox installed during Trading Blox installation. Only the **Portfolio Manager** can contain the [Rank Instrument](#) script section. This script section sees every instrument with a record for each date in the instrument file. Each date is passed through this script as that date is incremented during testing or order generation.

Ranking an instrument is most often a process by which an instrument is given a numerical value for each date processed by the [Rank Instrument](#) script section. A ranking value is applied as the [Rank Instrument](#) script is called for each record once in each instrument. This allows the blox logic to use the [Ranking Functions](#) available in the [Instrument Object](#) (i.e. `instrument.SetLongRankingValue` and `instrument.SetShortRankingValue`, etc.).

After all of the instruments have been ranked in the [Rank Instrument](#) script section, the [Filter Portfolio](#) script section is called once for each instrument. In this script section is used to determine the Instrument filtering. Filtering is performed by one or more of the [Instrument Object Trade Control Functions](#) that allow or deny order generation based upon the logic in the [Filter Portfolio](#) script section.

After all the portfolio's instruments have been assigned a ranking value the portfolio can be sorted. Sorting in the **Portfolio Manager** is performed automatically right after all the instruments have been processed by the [Filter Portfolio](#) script section as long as there is code in the [Filter Portfolio](#) script section. Sorting changes the order in which the instruments are sequenced during testing and order generation.

Ranking, Filtering and Sorting methods can easily be performed in other script sections. Ranking and Filtering can be performed in other script sections. For example script sections that also process each instrument for each record in a data series like [Before Instrument Day](#), [Update Indicators](#) and [After Instrument Day](#) they get the same exposure that is provided by the [Rank Instrument](#) script section. In addition script section [Before Trading Day](#) and [After Trading Day](#) script section.

Ranking, Filtering and Sorting can also be performed in each of the [Before Trading Day](#) and [After Trading Day](#) script sections. When sorting is performed in either of these two scripts they can use any of the many [System Object's](#) sorting functions to change how instruments are sorted. When no code is listed in the [Filter Portfolio](#) section and the rules of the system require the portfolio to be sorted, sorting can be performed in the [Before Trading Day](#) script section the [Rank Instrument](#) script was used to provide a ranking value. When a ranking is performed in [Update Indicators](#) or

After Instrument Day script sections instead of in [Rank Instrument](#) script, ranking can also be performed in the After Instrument Day script section

NOTE:

These scripts section **will execute** for a test date where an instrument date record is **not available**. This means when a test date is being processed and an instrument does not have a matching instrument data date record on which an order can be filled, or a position can be adjusted this script section **will process the last know instrument date**.

See Also:

[Ranking Properties](#)

[Ranking Functions](#)

[Trade Control Properties](#)

[Trade Control Functions](#)

1.2 Entry

This Entry Block is the main script section for creating entry orders/positions, changing position direction, or adding units to an existing position.

NOTE:

This scripts section will not normally execute when a instrument specific date record is not in the instrument's data file.

A current record is a record date for the same test date on which this script is executing. This means a test date must have an instrument date price record in order for this script section to execute.

When it does have a missing instrument record any orders that were active for the active test date, orders will not be filled or expired by the brokerage process. However, when an order is still active after the brokerage process, it will be available to execute an order fill or allowed to expire on the next available instrument date trading session.

1.3 Exit

Exit Block script section is the main script responsible for processing Exit Orders and Adjusting Position Quantities. This script only execute when an instrument has an active position that is either **LONG** or **SHORT**.

NOTE:

This scripts section will not normally execute when a instrument specific date record is not in the instrument's data file.

A current record is a record date for the same test date on which this script is executing. This means a test date must have an instrument date price record in order for this script section to execute.

When it does have a missing instrument record any orders that were active for the active test date, orders will not be filled or expired by the brokerage process. However, when an order is still active after the brokerage process, it will be available to execute an order fill or allowed to expire on the next available instrument date trading session.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 319

1.4 Money Manager

A Money Manager Block contains the [UNIT SIZE](#) script section. This is the primary script section for determining [unit order quantities](#). It is most often the script section that rejects an order when insufficient equity allocation doesn't support any quantity.

The [UNIT SIZE](#) script section is called by the [Broker Objects Entry Order Functions](#) so that an [order](#) can be given a quantity size.

Note:

The [UNIT SIZE](#) script can be **Added** to any block as integrated money manager when desired. However, there can only be one [UNIT SIZE](#) script in any of the system's list of blox.

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 426

1.5 Risk Manager

The 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 not in an active position.

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

Any Auxiliary Blox can be used to create custom indicators, reports or statistics. An Auxiliary block might have the Update Indicators script, or perhaps other scripts that do not change the block type. Adding scripts like the Can Add scripts or Before/After Trading Day scripts is not an uncommon use of this block.

Think of this block designation as being flexible in their purpose. For example, if you create some functions that you might want to use with many systems, this Auxiliary blox is a good place to create these new capabilities because they can be in any system. There is no limitation on how many Auxiliary blox are listed in a system.

Edit Time: 9/11/2020 4:48:24 PM

Topic ID#: 155

Section 2 – Blox Scripts

This next table shows the Trading Blox Builder script sections for each of block modules that might be needed in a system.

Default script sections can also include many of the typical block scripts when the "**Include Default Scripts**" option is enabled when the block is created. When that option isn't enabled, the block will only contain the defining block script sections for that block type.

Adding additional script section to an existing block is an easy process. Details for Adding and Removing script section is available the "[Managing Block Scripts](#)" topic.

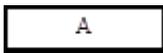
Some script sections define, and can change a blox type after a defining script section is added. For example, any blox that contains Rank Instruments or Filter Portfolio will be defined as a Portfolio Manager blox. Any script that contains any of the Risk Manager's scripts (see rows 20 to 23 in the table image below) will be defined as a Risk Manager blox.

The labels shown in the next image are explained in the table below the image:

Defining Blox Module Scripts + Optional Script Additions								
Count	Script Section Name:	Portfolio Manager	Entry	Entry, Exit	Exit	Money Manager	Risk Manager	Auxiliary
1	Before Simulation	A	A	A	A	A	A	A
2	Set Parameters	O	A	D	A	O	O	O
3	Before Test	A	A	A	A	A	A	A
4	Rank Instrument	D						
5	Filter Portfolio	D						
6	Before Trading Day	A	A	A	A	A	A	A
7	Before Instrument Day	A	A	A	A	A	A	A
8	Before Bar	I		I		I	I	I
9	Exit Orders	O	O	D	D	O	O	O
10	Entry Orders	O	D	D	A	A	O	O
11	Can Place Order	O	O	O	O	O	O	O
12	Unit Size		A	A	A	D		O
13	Can Add Unit		A	D	A	A	D	O
14	Before Order Execution		O	O	O	O	O	O
15	Filtered Order Notification		O	O	O	O	O	O
16	Update Indicators	O	O	O	O	O	O	D
17	Can Fill Order		A	D	A	O	D	O
18	Exit Order Filled		A	D	A	O	O	O
19	Entry Order Filled		A	D	A	O	O	O
20	After Instrument Open		O	O	O	O	O	O
21	After Bar	I	I	I	I	I	I	I
22	Adjust Stops		O	O	O	O	O	O
23	Initialize Risk Management						D	
24	Compute Instrument Risk						D	
25	Compute Risk Adjustments						D	
26	Adjust Instrument Risk						D	
27	After Instrument Day	A	A	A	A	A	A	A
28	After Trading Day	A	A	A	A	A	A	A
29	After Test	A	A	A	A	A	A	A
30	After Simulation	A	A	A	A	A	A	A
31	Post Process Utility	O	O	O	O	O	O	O
32	<Custom Name Script>	O	O	O	O	O	O	O

Defining Blox Script and Additional Scripts Available

Legend:	Description:
	<p>This label identifies the block scripts that determine the block type when created will appear only with the scripts name that are identified with this label when the "Include Default Scripts" option is not selected.</p> <p>When the Exit Orders script is removed from the "Entry, Exit" block, the block type name will change to an Entry block.</p> <p>When the Entry Orders script is removed from the "Entry, Exit" block, the block type name will change to an Exit block.</p>

Legend:	Description:
	When the "Include Default Scripts" is enabled, the script names shown with this label are the additional scripts that will be added to a new block when it is created.
	Cells where this label is displayed show other script sections that can be added manually to an existing block. Adding a script section is easy when the block is displayed in the Blox Basic Editor so the scripts in the block are visible. To add a script section that isn't in the block already, click on the script section. Custom script execute when their names are called using the Script Object's Execute function
	Intraday script section executes for each intraday record.

NOTE:

Blox Script name rows in the above image that have a Legend 'D' mark entered in the Instrument column, those script names will automatically provide **IPV** variables with context so they can be accessed and changed.

Instrument scripts that automatically provide context, will execute each instrument symbol in the portfolio that has a date record in its file that aligns with the system testing date in progress. For example, if there are ten-instruments in the portfolio, each instrument script will execute one time on each of the system testing dates. When an instrument doesn't have a date that aligns with the current system test date, the instrument object [instrument.tradesOnTradeBar](#) property, will be **FALSE**, and that instrument won't be tested again until has date record that aligns with the system testing date. When an instrument has a date record that aligns with the next system testing date, the [instrument.tradesOnTradeBar](#) property will be **TRUE**.

There are exceptions with the script sections that contained a red Legend 'D' and a green background cell. These scripts have dependencies that are explained in the Legend notes.

Instrument scripts without any Legend symbols, do not automatically provide instrument context, and they do not process each instrument in the portfolio. However, instruments can be processed in these script section, but they must be loaded and processed using the [LoadSymbol](#) function using a repetitive loop structure, and they must use **Instrument-BPV** type variable that is created in the **BPV** variable creation dialog.

Links:
Blox Script Timing , Blox System Placement , Blox Types , Trading Objects References
See Also:

Edit Time: 3/21/2024 10:41:38 AM

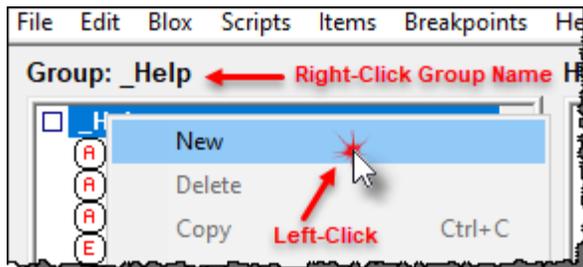
Topic ID#: 172

2.1 New Block Creation

Adding new blocks with different methods is an easy way to discover improvements to your systems. It is easy because the ability to create a new block is simple, and modifying an existing block that is copied, given a new name gives the developer a known set of scripts that can be modified to work in a different way. At the heart of all systems are various ideas about how a system should follow markets.

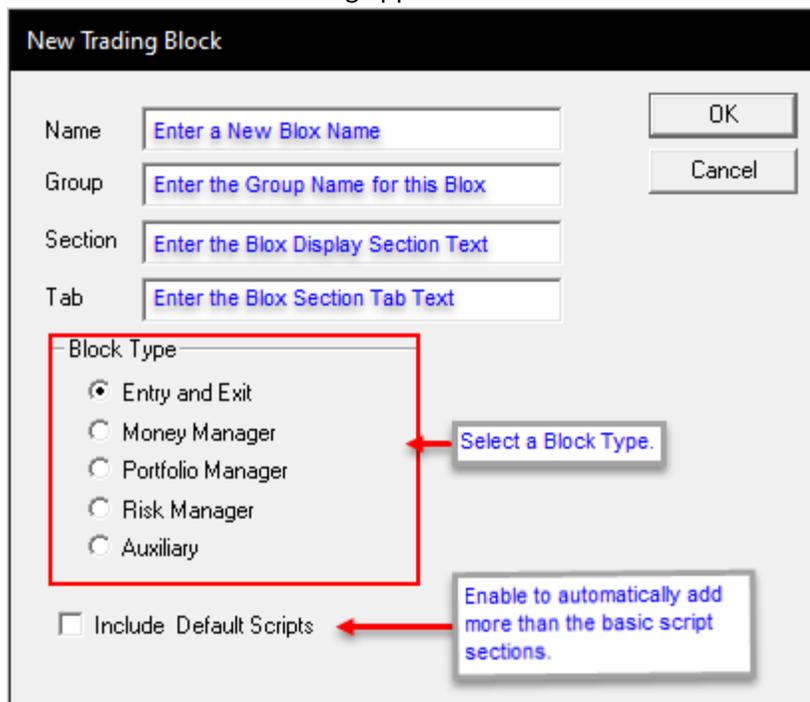
Creating a New Block:

Clicking on the **Blox** tab that is on the top **Menu**, or **Right-Click** the mouse on the **Group Section-Name** where the block will be stored.



How to Create a New Block.

When the pop-up selection dialog appears, Click on the **New** option. That New selection will have the new block creation dialog appear:



New Blox Creation Dialog

- This dialog shows four information text-fields that need to be filled out.
- The Block Type area determines the show five different block types that determine which script sections will appear in the block by default.

- The "Include Default Scripts" option provides a means to add some of the more common scripts sections that user tend to use when the same block was created for them.
- When all the information is provided, and you click OK, the block will appear in the Group that was active when you created the block.
- To copy an existing block, click on the Copy menu selection and modify the details in the text fields. When copy and renaming is complete, save the block.
- When all the information is provided, and you click OK, the block will appear in the Group that was active when you created the block.

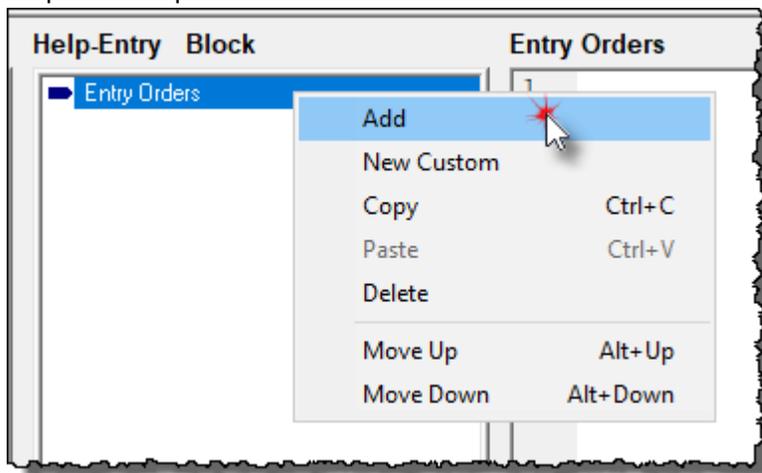
Adding Block Scripts:

Adding one or more script sections to an existing blox is simple and quick. If the block where more scripts need to be available is a new block, the "**Include Default Scripts**" will often add many of the scripts users tend to add, but that option might add more than you need.

NOTE:

Script sections where no script statements are shown, that script section will not execute because there is nothing for those scripts to perform. While this makes the process faster, if the empty scripts are available in other blocks in the system and those scripts sections contain script statements, those other block scripts will execute.

To add a script section, **Right-Click** the mouse click on a script section to display the script pop-up script action options.



Add Script Section to a Block

Script Menu Options:	Keys	Descriptions:
Add		Adds a selected script section below the highlighted script name.
New Custom		Adds a User's Custom Script section.

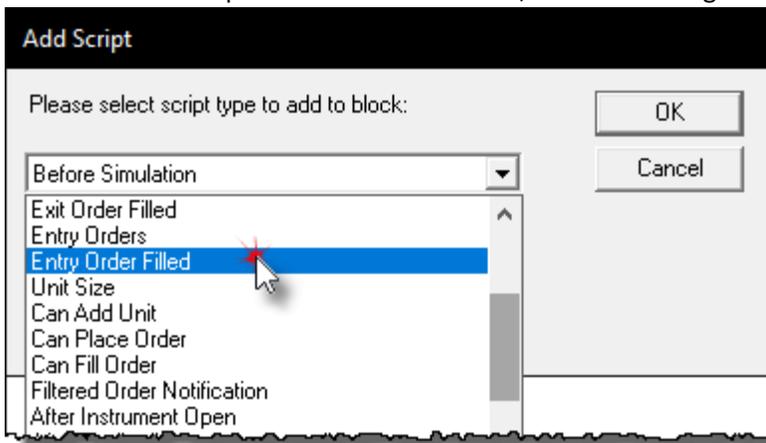
Script Menu Options:	Keys	Descriptions:
Copy	Ctrl+C	Allows the user to copy a script section they can then use to paste into a different block
Paste	Ctrl+V	Allows the user to Paste a copied script section into a different block.
Delete		Removes the highlighted script section from the block.
Move Up	Alt+Up	Allows the user to user move a script section Up when the Alt+Up-Arrow keys are pressed.
Move Down	Alt+Down	Allows the user to user move a script section Down when the Alt+Down-Arrow keys are pressed.

NOTE:

The ability to Move Script section to different locations in a block can happen when the user highlights a script section and the uses the Alt-Up or Alt-Down keys to move the script section.

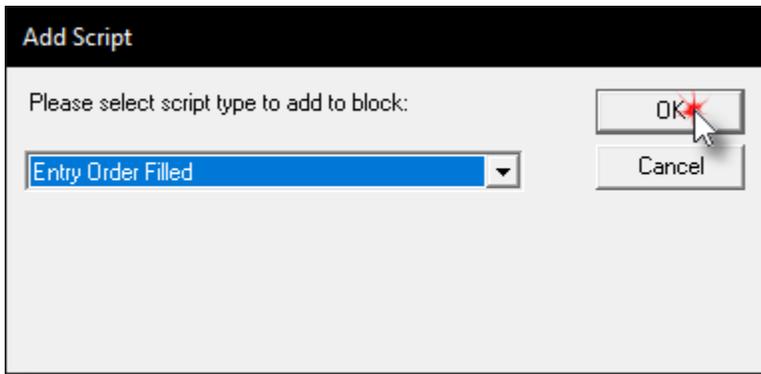
Adding a Script Section Example:

When the Add script menu item is selected, this next dialog will appear.



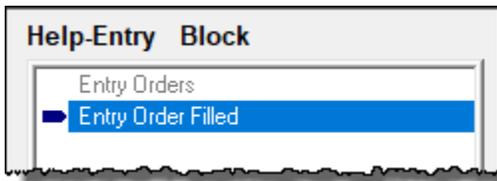
New Blox Creation Dialog

The items listed in this list of script sections will be added to a block once the new script section is highlighted.



New Blox Creation Dialog

When the script section to add is selected, the Add Script dialog closes the list of Script Sections available so the user can click on the OK button so the new script section will appear.



New Blox Creation Dialog

After the script section is added, it will then appear in the block where it was not available before.

Links:
Blox Script Timing , Blox System Placement , Blox Types , Trading Objects References
See Also:

2.2 Managing Block Scripts

Scripts with the same name will execute in the order the Suite process added a system to its list of systems. The first system added to a Suite, will be assigned an index = 1, the second system index = 2, etc.

Global System Blocks will be assigned an index = 0.

See the information in these two topics for more information.

- [Blox Scripts](#)
- [Blox Script Timing](#)
- [Global Script Timing](#)

Links:
Blox Script Timing , Blox System Placement , Blox Types , Trading Objects References
See Also:

Section 3 – Blox Script Timing

This table shows how each Trading Blox Builder script section is available during testing, and when it will execute for each test date and or time. For example, **Before Simulation** and **After Simulation** script section will only execute once during the selected Suite test. In a stepped test, **Before Test** and **After Test** will execute once for each step in a test.

Before Trading Day and other scripts identified in the table column named: **Runs Once Each Bar** will execute for each test date or date-time record in the data.

Script section identified in the **Runs Once Each Instrument** will execute once for each instrument for each date or date-time record in the data file. Instrument identified in this column not assigned a "Y" designation, but instead show a reference notation (see table below), will only execute when the instrument test condition for that instrument requires execution.

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

Blox Execution Timing & Frequency

All the Trading Blox Builder module scripts are shown in this table:

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 Builder Basic** statements will execute. Script that are blank are not executed.

Only the scripts displayed in blox are 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. Script execution for section identified with a notation also require the notation's condition be met to enable execution.

Each notational exception is describe in the legend below the table.

NOTE:

Blox Script name rows in the above image that have a Legend 'D' mark entered in the Instrument column, those script names will automatically provide **IPV** variables with context so they can be accessed and changed.

Instrument scripts that automatically provide context, will execute each instrument symbol in the

portfolio that has a date record in its file that aligns with the system testing date in progress. For example, if there are ten-instruments in the portfolio, each instrument script will execute one time on each of the system testing dates. When an instrument doesn't have a date that aligns with the current system test date, the instrument object [instrument.tradesOnTradeBar](#) property, will be **FALSE**, and that instrument won't be tested again until has date record that aligns with the system testing date. When an instrument has a date record that aligns with the next system testing date, the [instrument.tradesOnTradeBar](#) property will be **TRUE**.

There are exceptions with the script sections that contained a red Legend 'D' and a green background cell. These scripts have dependencies that are explained in the Legend notes.

Instrument scripts without any Legend symbols, do not automatically provide instrument context, and they do not process each instrument in the portfolio. However, instruments can be processed in these script section, but they must be loaded and processed using the [LoadSymbol](#) function using a repetitive loop structure, and they must use [Instrument-BPV](#) type variable that is created in the [BPV](#) variable creation dialog.

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. They also require an entry order was filled. 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.
R-1	Filtered Order Notification will execute after every order rejection. If there is script in this script section, the script can send information showing why the order was rejected to the Log Window display. Note: The Log Window must be open for the print information to appear.
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	<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

Legend:	Execution Description:
Scripts	return to line in the script from which they were called.

Links:
order.continueProcessing , order.isEntry , System Object , Accessing Instruments , LoadSymbol
See Also:
Script Section Type Details , Global Script Timing Table

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 173

Section 4 – Blox System Placements

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 and support how the system operates. Some script names are only executed when position is active, or when a broker object statement is executed.

Note:

Systems in a Suite, and Blox in a System are sorted in alphabetical order when the Suite and when the Systems are created. This means the the system with letters that sort ahead of all the other systems in the Suite, will be assigned to system index 1. The system that is second in the name sorted order will be assigned, system index 2, etc.



System Block Editor

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 where they are displayed will determine the order in which the blox script section with the same name 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 one module of that type are allowed to be placed within a system, will follow the same alphabetical blox name order.

Section 5 – Global Script Timing

Trading Blox Suites support simultaneous testing of multiple systems so the overall strategy can include multiple trading methods. This means that systems with the same name and with different portfolios or with an different instrument class can be tested in a Suite so their performance is analyzed.

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, or when some scripts can have multiples of some script name section as other modules in the system the order in which these scripts are executed is important to how the test is performed.

All script sections 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 display list area (See [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 is sorted in the suite. The system name shown on the left-most system-name tab will have the system index 1. The system in the next tab to the right, will have system index 2.

Suites can also contain a Global System Scripts ([GSS](#)) when the name of the system name is 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 6 – Script Section Reference

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	This script section is the first script section to execute once a Suite Test is executed.
Set Parameters	This script section runs after Before Simulation , and ahead of when instruments are loaded. In this script, instrument can be Disabled trading before they are loaded, and before indicators are calculated in the Before Test script section.
Before Test	This script section begins to execute after the Before Simulation and Set Parameters script sections. Instruments are loaded and indicator calculations happen in this script section.
Rank Instruments	
Filter Portfolio	
Before Trading Day	
Before Instrument Day	
Before Bar	
Before Close	Before Close script section runs only once for each bar and it does not have automatic instrument context access.
Exit Orders	
Entry Orders	
Can Place Order	Executes after a Broker Function Call and before the Unit Size script executes.
Unit Size	
Can Add Unit	
Filtered Order Notification	This script section provides access to the order rejection message so that additional information can be added. When this script section is active

Script Name:	Description:
	<p>it will automatically provide instrument and order context at the same time.</p> <p>This script section becomes accessible when an order is in process of being rejected.</p>
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	
Post Process Utility	

Further information on a specific blox script can be found in [Blox Script Access](#) and in [Blox Script Timing](#). In some cases the descriptions in the [Blox Reference](#) section for each block type will have information specific to that block type if applicable.

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 519

6.1 Before Simulation

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.

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:

Example:

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
```

Links:

[LoadSymbol](#), [totalInstruments](#), [System Properties](#)

See Also:

[Set Parameters](#), [GetSteppedParameter](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 209

6.2 Set Parameters

This script section executes before the instrument data is loaded. At this stage of testing, Trading Blox Builder enables access to all instruments that will be loaded when the Before Test Script section is operating.

Note:

In a stepped simulation, the threads are locked. This is done so the script sections"

- Set Parameters
- Before Test
- After Test

In each thread each thread will complete without overlap from a different thread.

This script section enable or disable one or more symbols in the portfolio. Symbols disabled prior to loading, will not be loaded in an inactive state. When an instrument is not loaded, the computer's memory space will remain available. The time to load the data, would be used. In simple terms, reducing the active symbols in a portfolio reduces the memory required and the time it would take to load the symbols and calculate the systems indicators.

Set Parameters script executes:

- After the Before Simulation script and ahead of the Before Test script.
- Before the system's indicators are computed..
- This script section support the script.InstrumentLoop function that will automatically loop through all the symbols in the portfolio.
- Enabling and Disabling indicators and custom **IPV** and **BPV** series disabled will reduce some test time.
- Setting the Type Value of an indicator, before the indicator is computed. i.e. Changing from a SMA of the Close to an SMA of the Open can also be handled in this section.

How to change an instrument's ability to Load:

The process uses this function when called from this script section.

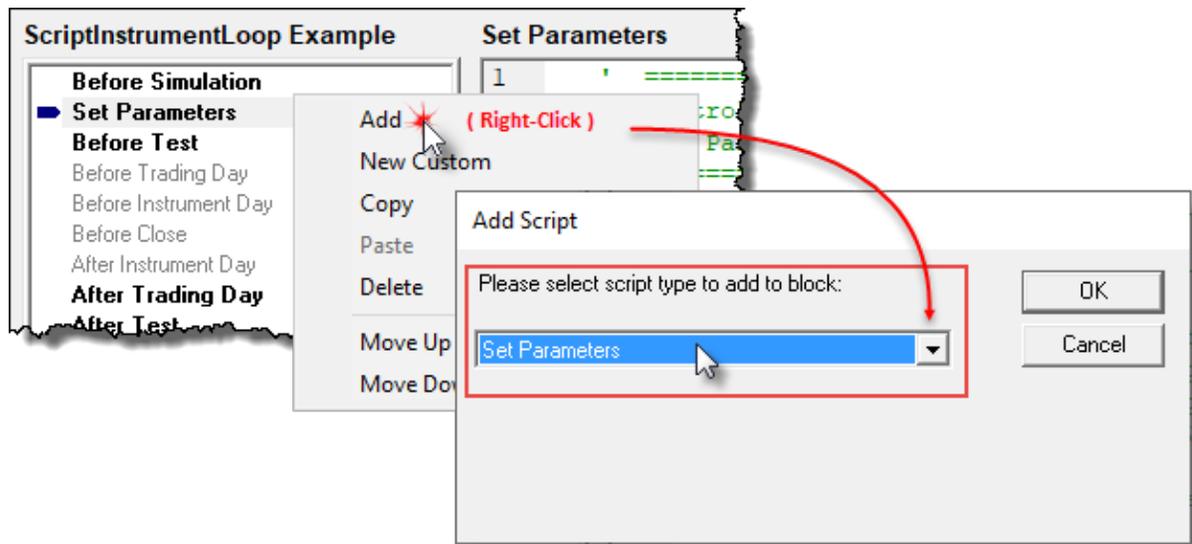
```
script.InstrumentLoop( "anyCustomInstrumentScriptName" )
```

When called from another script section, the example in the [instrument.DisableTrading](#) topic can be used.

Note:

- Regular Indicators that use Price or Volume as a data source are not recomputed for each stepped test run unless an indicator dependent parameter changes value.
- Calculated Indicators and Regular Indicators that use another indicator as the source data, are still recomputed for each test.
- Multi threaded testing computes indicators for each thread separately, which uses memory and time as initial overhead.

When the Set Parameters script section isn't displayed in a blox, the following sequence of steps will add it after the script that is right-clicked to enable an additional script section to be loaded.



Add Set Parameters Custom Script Section Steps

The [Set Parameters](#) script section will always run after the Before Simulation script is finished executing. It will also run before the Before Test script section is run.

Variable Portfolios:

Before Trading Blox Builder can run a test, it needs to know all the market symbols allowed to trade. Usually, each symbol in the portfolio loads into memory. Every symbol in memory is available for order generation analysis. The key to creating a variable portfolio is to prevent some symbols from being loaded into memory.

At the beginning of a test, the [Before Simulation](#) script section initializes all the test settings. All the portfolio symbols are recognized and enabled so they can load into memory. After the [Before Simulation](#) script section, this [Set Parameters](#) script section will run when scripting is entered its script area.

The [Set Parameters](#) script section has access to all the symbols in the portfolio. It uses a [Custom Script](#) section that can change a symbol from being available to trade to being unable to trade.

A symbol with its `instrument.inPortfolio` property value set to `TRUE` is allowed to trade. Changing the property to `False` by applying the `instrument.DisableTrading` function stops the symbol from trading. It can also reverse that condition by assigning the `instrument.inPortfolio` property to `TRUE`.

These indicators and series can be set as enabled or disabled by default in the **Trading Blox Builder Basic** editor.

Example:

This is an example of how a custom script can be created to disable symbols in a portfolio from being used by the system. The name of the custom script created is "`iOnOff_Instrument`"

Example:

This script detail is placed Set Parameters script section so the custom script "iOnOff_Instrument" can be executed when the Set Parameter script section executes.

Notes:

- Creating a specific set of instruments from a portfolio that has all the possible instrument you wish to examine should be used as the Symbols Supply.
- Before using the Symbol Supply portfolio, use the `system.SortInstrumentList(5)` to ensure all the symbols are in alphabetical order. This also true for target list of symbols that have been created.
- Specifying which symbols in the Symbol Supply are needed for a test should be placed in a normal portfolio set file, or an independent text file that can be accessed and used as a stepped process to identify which symbols should be allowed, and which symbols should be blocked from trading.
- Adding a Selector Parameter with the names of the variable portfolio in use can be used to determine which target-portfolio symbols are to be used for the selected portfolio name.

```

' =====
' Control Active Symbols
' Set Parameters - SCRIPT START
' =====
' ~~~~~
' Set Parameters script runs after parameters
' are set For the test run, but before the
' indicators are computed.
' -----
' This script Allows For:
' 1) Setting custom parameter values TO
'    be used when computing indicators.
' 2) Enabling AND Disabling indicators
'    AND custom IPV AND BPV series TO
'    save time AND memory
' 3) Setting the Type Value of an
'    indicator, before the indicator
'    is computed. Changing from a
'    SMA of the Close TO an SMA of
'    the Open, For example.
' -----
' Runs a custom script For each enabled
' instrument in the portfolio. Sets the
' default instrument object.
script.InstrumentLoop( "iOnOff_Instrument" )
' ~~~~~
' =====
' Set Parameters - SCRIPT END
' Control Active Symbols
' =====

```

In this example, the Symbols to disable are hard coded into the conditional:

```
If ( sSymbol = "CD" ) OR ( sSymbol = "DJ" ) THEN
```

Example:

That can be changed by adding a list of symbols into a string series. If all the symbols in the series are enable for trading, then any symbol not in the series can be disabled from loading into memory. The following example scripts will automatically loop all the instruments in the portfolio so they can be trading-disabled, or allowed to trade.

```

' =====
' Control Active Symbols
' iOnOff_Instrument - SCRIPT START
' =====
' script.InstrumentLoop( "iOnOff_Instrument" )
' This Script Section Disables specific Instruments in a
' Portfolio. It then Display Current Date, Symbol, and its
' Trading State condition.
' ~~~~~
' Create Temp Variables for Print & Debug Stepping Display
VARIABLES: iDate, iInPortfolio      Type: Integer
VARIABLES: sSymbol                  Type: String
' -----
' Temp Print & Debug Assignments
iDate = instrument.date
sSymbol = instrument.symbol
iInPortfolio = instrument.inPortfolio
' -----
' Display the loaded instrument.
PRINT instrument.date, instrument.symbol, instrument.inPortfolio
' If the symbol is listed here, turn make it InActive
If ( sSymbol = "CD" ) OR ( sSymbol = "DJ" ) THEN

  ' Remove From Portfolio
  instrument.DisableTrading

  ' Example code in a custom script called
  ' "iOnOff_Instrument"
  iDate = instrument.date
  sSymbol = instrument.symbol
  iInPortfolio = instrument.inPortfolio

  ' Display Current Date, Symbol, and its Trading State
  PRINT instrument.date, instrument.symbol, instrument.inPortfolio
ENDIF
' Add a Space between each symbol
PRINT
' ~~~~~
' =====
' iOnOff_Instrument - SCRIPT END
' Control Active Symbols
' =====

```

Returns:

System's Portfolio has four symbols:

Returns:**AD, DJ, ED, CD**

Prior to this custom script running, all of the instruments were active and would load if this script had not been run.

Calling Script Name: Set Parameters

Output shows the Date when the instrument's `inPortfolio` property was changed.

`instrument.inPortfolio` at start, shows all the symbols will show **TRUE**.

When the `instrument.DisableTrading` is executed, it will change to **FALSE**.

Date:	Symbol	<code>inPortfolio</code> -State
20140402	AD	1
20140402	DJ	1
20140402	DJ	0
20140402	ED	1
20140402	CD	1
20140402	CD	0

Links:

[SetSeriesEnable,](#)

See Also:

Before Simulation, Before Test

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 94

Set Parameters Methods

This script section executes before instrument data is loaded. This means it can change which instruments will get loaded during the Before Test Script section.

This script section can enable, disable one or more symbols in the portfolio. Symbols that are disabled, will not be loaded in their inactive state. When an instrument is not loaded, the data it would have consume will remain available. The time to load the data, would be used. In simple terms, reducing the active symbols in a portfolio reduces the memory required and the time it would take to load the symbols and calculate the systems indicators.

The Set Parameters script executes:

- After the Before Simulation script and ahead of the Before Test script.
- Before the system's indicators are computed..

- This script section support the script.InstrumentLoop function that will automatically loop through all the symbols in the portfolio.
- Enabling and Disabling indicators and custom **IPV** and **BPV** series disabled will reduce some test time.
- Setting the Type Value of an indicator, before the indicator is computed. i.e. Changing from a SMA of the Close to an SMA of the Open can also be handled in this section.

How to change an instrument's ability to Load:

The process uses this function when called from this script section.

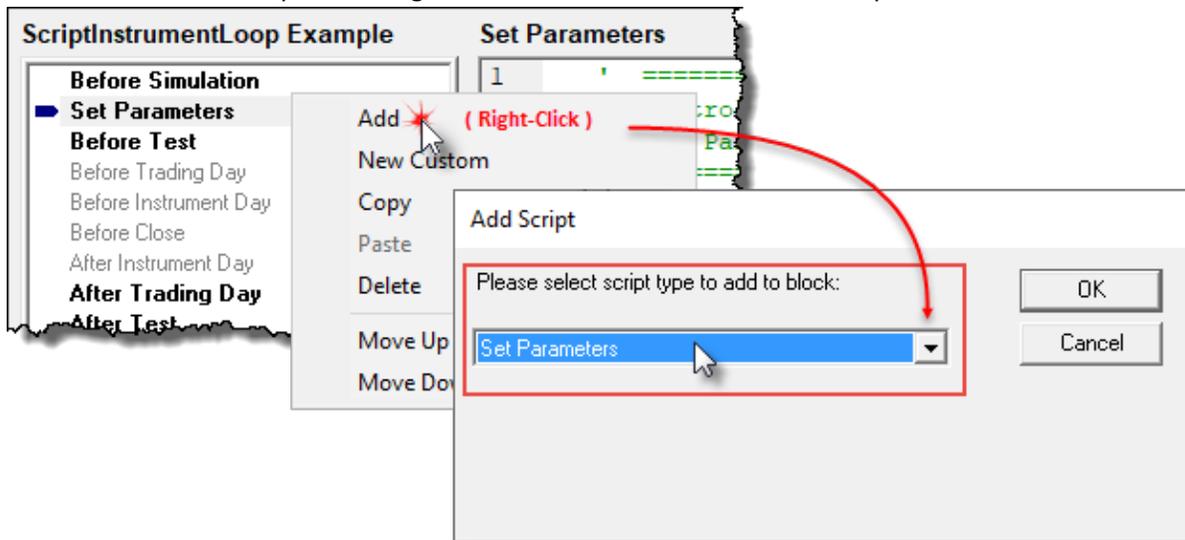
```
script.InstrumentLoop( "anyCustomInstrumentScriptName" )
```

When called from another script section, the example in the [instrument.DisableTrading](#) topic can be used.

Note:

- Regular Indicators that use Price or Volume as a data source are not recomputed for each stepped test run unless an indicator dependent parameter changes value.
- Calculated Indicators and Regular Indicators that use another indicator as the source data, are still recomputed for each test.
- Multi threaded testing computes indicators for each thread separately, which uses memory and time as initial overhead.

When the Set Parameters script section isn't displayed in a blox, the following sequence of steps will add it after the script that is right-clicked to enable an additional script section to be loaded.



Add Set Parameters Custom Script Section Steps

The [Set Parameters](#) script section will always run after the Before Simulation script is finished executing. It will also run before the Before Test script section is run.

These indicators and series can be set as enabled or disabled by default in the **Trading Blox Builder Basic** editor.

Example:

This is an example of how a custom script can be created to disable symbols in a portfolio from being used by the system. The name of the custom script created is "iOnOff_Instrument"

This script detail is placed Set Parameters script section so the custom script "iOnOff_Instrument" can be executed when the **Set Parameter** script section executes.

```

' =====
' Control Active Symbols
' Set Parameters - SCRIPT START
' =====
' ~~~~~
' Set Parameters script runs after parameters
' are set For the test run, but before the
' indicators are computed.
' -----
' This script Allows For:
'   1) Setting custom parameter values TO
'       be used when computing indicators.
'   2) Enabling AND Disabling indicators
'       AND custom IPV AND BPV series TO
'       save time AND memory
'   3) Setting the Type Value of an
'       indicator, before the indicator
'       is computed. Changing from a
'       SMA of the Close TO an SMA of
'       the Open, For example.
' -----
' Runs a custom script For each enabled
' instrument in the portfolio. Sets the
' default instrument object.
script.InstrumentLoop( "iOnOff_Instrument" )
' ~~~~~
' =====
' Set Parameters - SCRIPT END
' Control Active Symbols
' =====

```

In this example, the Symbols to disable are hard coded into the conditional:

```
If ( sSymbol = "CD" ) OR ( sSymbol = "DJ" ) THEN
```

That can be changed by defining a list of symbols the custom script can use to learn what needs to be disabled.

```

' =====
' Control Active Symbols
' iOnOff_Instrument - SCRIPT START
' =====

```

Example:

```

' script.InstrumentLoop( "iOnOff_Instrument" )
' This Script Section Disables specific Instruments in a
' Portfolio. It then Display Current Date, Symbol, and its
' Trading State condition.
' ~~~~~
' Create Temp Variables for Print & Debug Stepping Display
VARIABLES: iDate, iInPortfolio      Type: Integer
VARIABLES: sSymbol                  Type: String
' -----
' Temp Print & Debug Assignments
iDate = instrument.date
sSymbol = instrument.symbol
iInPortfolio = instrument.inPortfolio
' -----
' Display the loaded instrument.
PRINT instrument.date, instrument.symbol, instrument.inPortfolio
' If the symbol is listed here, turn make it InActive
If ( sSymbol = "CD" ) OR ( sSymbol = "DJ" ) THEN

    ' Remove From Portfolio
    instrument.DisableTrading

    ' Example code in a custom script called
    ' "iOnOff_Instrument"
    iDate = instrument.date
    sSymbol = instrument.symbol
    iInPortfolio = instrument.inPortfolio

    ' Display Current Date, Symbol, and its Trading State
    PRINT instrument.date, instrument.symbol, instrument.inPortfolio
ENDIF
' Add a Space between each symbol
PRINT
' ~~~~~
' =====
' iOnOff_Instrument - SCRIPT END
' Control Active Symbols
' =====

```

Returns:

System's Portfolio has four symbols:

AD, DJ, ED, CD

Prior to this custom script running, all of the instruments were active and would load if this script had not been run.

Calling Script Name: Set Parameters

Output shows the Date when the instrument's `inPortfolio` property was changed.

`instrument.inPortfolio` at start, shows all the symbols will show **TRUE**.

When the `instrument.DisableTrading` is executed, it will change to **FALSE**.

Returns:		
Date:	Symbol	inPortfolio-State
20140402	AD	1
20140402	DJ	1
20140402	DJ	0
20140402	ED	1
20140402	CD	1
20140402	CD	0

Links:
SetSeriesEnable,
See Also:
Before Simulation, Before Test

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 700

6.3 Before Test

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. For example, when there is a need access additional **IPV** properties and update them with specific information that will be the same throughout a test, this script is a good place to perform that task.

This script has access to the current parameter settings for the test. It is also the script section where all the symbols in the portfolio are loaded into Trading Blox Builder memory.

Note:

Before Test, Set Parameters, and After Test to be thread locked so the script for each thread will complete without overlap from another thread.

[GetSteppedParameter](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 210

6.4 Rank Instruments

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

The [Rank Instruments](#) script is available in a [Portfolio Manager](#) blox type. The [Ranking Functions](#) are for creating an order list of symbols.

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 can sort 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 )
```

Links:
Trade Control Functions
See Also:

6.5 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.

When there is no scripting code in the Filter Portfolio script section, then the execution of the Ranking Function can 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
```

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 343

6.6 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 purposes.

This script does not have automatic-context access to Instrument properties, or Instrument Permanent variables. Instruments can be called in this script section using the [LoadSymbol](#) function to provide instrument context access.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 211

6.7 Before Instrument Day

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

Before Instrument Day runs once per day. If you want to run a script on every bar, use the [Before Bar](#) script section.

This **Before Instrument Day** section executes on every test date record in a data file for each instrument in the portfolio. When a date record isn't in a symbol's data series, the information in the properties will be from the last date record in the symbol data. where a date record exist and that instrument's property reports `instrument.tradesOnTradeDate = TRUE` when the amount of data required for its indicators to be primed shows the symbol property `instrument.IsPrimed = TRUE`.

Execution for each instrument happens after the **Before Trading Day** script section completes execution.

Script can be an alternate location to calculate instrument-specific variables or custom indicators each day.

Executing this script when the instrument does not have a date for the current instrument date being processed, the property value information will be from the most recent instrument date.

Use the [instrument.tradesOnTradeBar](#) property return value to determine if the instrument record being processed is for the current period, or an earlier period. When [instrument.tradesOnTradeBar](#) return value is TRUE, the instrument information is current. When it is FALSE, values available will be from the most recent price record.

Use the [instrument.tradesOnTradeBar](#) property when it is important to know, or to not allow calculation changes.

6.8 Before Bar

This script does not have automatic instrument context by default. Instrument access can be achieved using the [LoadSymbol](#) Function.

This script section is available just ahead of the next price bar in an Intraday data file test. If you need to change your signal with intraday data, this script section is where it should happen.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 162

6.9 Before Close

Before Close script section runs only once for each bar and it does not have automatic instrument context access.

Note:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 98

6.10 Exit Orders

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

When the current instrument in the portfolio is selected and it also has an existing active position this script's access is enabled for execution.

Exit Orders Script section is used to create order that:

- Exit a one or more units in a multiple unit position
- Exit and entire position
- Adjust a protective price for an entire position
- Adjust the protective any, or only some of the units in a position.
- Increase a unit or a position quantity
- Decrease a unit or a position quantity

Most of the position's need shown above are created when the Exit Order script is called.

NOTE:
This scripts section will not normally execute when a instrument specific date record is not in the instrument's data file.
A current record is a record date for the same test date on which this script is executing. This means a test date must have an instrument date price record in order for this script section to execute.
When it does have a missing instrument record any orders that were active for the active test date, orders will not be filled or expired by the brokerage process. However, when an order is still active after the brokerage process, it will be available to execute an order fill or allowed to expire on the next available instrument date trading session.

Example:

```
' 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
```

NOTE:

This sample script has two common features.

1. It is able to check against an active position use resolving the `instrument.position` property to determine if the active position is **LONG** or **SHORT** positions.
2. By assigning its Exit Stop order price to the **instrument.unitExitStop price** its previously protective price is available as a worse case Exit Price Stop value.

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.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 323

6.11 Entry Orders

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

Use this script section to create orders for new positions, or for adding units to a position. Entry Orders is called for each instrument with a current date record.

NOTE:

This scripts section will not normally execute when a instrument specific date record is not in the instrument's data file.

A current record is a record date for the same test date on which this script is executing. This means a test date must have an instrument date price record in order for this script section to execute.

When it does have a missing instrument record any orders that were active for the active test date, orders will not be filled or expired by the brokerage process. However, when an order is still active after the brokerage process, it will be available to execute an order fill or allowed to expire on the next available instrument date trading session.

Example:

```
' Following Entry Orders script is copied 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
' instrument is not LONG, Then create a LONG order.
broker.EnterLongOnOpen
ENDIF

If macdIndicator < 0 AND instrument.position <> SHORT THEN
' Two conditions must be met - MACD below 0 and
' instrument is not SHORT, Then create a SHORT order.
broker.EnterShortOnOpen
ENDIF

' This script has the broker object enter new long
' and short positions depending on the macdIndicator
' value to determine if is positive or negative.

' 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.
```

NOTE:

Unless you wish to add to an additional Unit to an existing position you should always check for an existing position before creating a new order to enter in the same direction.

6.12 Can Place Order

This script section is similar to the [Can Add Unit](#) script section. It often applies various information and filters to an Active-Order to ensure the system's portfolio goals are supported. However, where it is different is how it executes before the [Unit Size](#) script section adds size to the unit. The Can Add Unit script section runs after the Unit Size script section is completed.

In this script section will execute right after an order generated by [Broker-Object](#) function. Orders are always active because it is the order that caused this script section to be available. Because this script section executes ahead of the [Unit Size](#) script, any missing protective exit price can be applied or changed, and text that provides order-reason can be added to one or more units being looped. Filters can be applied to ensure that the additional unit additions do not create risk boundaries greater than how the design requires the system rules of how those script sections are managed.

This [Can Place Order](#) script, which runs for entries and exits as they are being created and placed, can be used as a generic replacement for the [Can Add Unit](#) script.

NOTES:

In the Roundtable forum here: <https://tinyurl.com/ybct9ast>

The following conversation about how to add a unique reason to each unit transpired.

[Re: Rule labels with multiple units](#) (link)

[Post](#) by [Tim Arnold](#) » Tue May 05, 2020 11:48 am

Thinking outloud, I guess I would start with the idea of setting an IPV flag before the adjustment, and using that in the Can Place Order script to know where this order is coming from and set the rule accordingly. If you have multiple rules that exit using the adjust position, you could expand this concept so the adjustFlag is a number from 1 to 10. Or, even a string that contains the rule label that you want to apply to the order.

```
adjustFlag = TRUE
AdjustOnOpen
adjustFlag = FALSE
```

Links:

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

See Also:

[Can Add Unit](#), [Unit Size](#), [system.OrderExists](#)

6.13 Unit Size

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

This script runs when one of the Broker Entry Functions executes. While this Unit Size script is active, the Entry-Order quantity is applied before it is sent to the Can Add Unit script section.

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:

6.14 Can Add Unit

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

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
```

6.15 Filtered Order Notification

This script section provides access to the order rejection message so that additional information can be added. When this script section is active it will automatically provide instrument and order context at the same time.

This script section becomes accessible when an order is in process of being rejected.

Rejected message can happen for a range of conditions established in the [Unit Size](#) or [Can Add Unit](#) script sections. What the rejected message reports is included and determined by the script section's statements. Orders are rejected when the conditions created to reject an order are found. The rejections of an order uses the [order.Reject](#) function. This function handles the rejection details and it also carries the information that explains why the order was rejected.

The [order.processingMessage](#) property is the method for how an order rejection message is added to the Trading Blox Builder Filter Log.csv file that is in the Results Folder where Trading Blox Builder is installed. After every test, the **Filter Log.csv** file is updated after a test is completed.

The **Filter Log.csv** file has a record of every rejected order. The **Filter Log.csv** is a ".csv" (comma separated values) file that will easily open into a spreadsheet. The message that appears in this script section when there is script statements to display the rejection message, this script section will execute each time an order is rejected.

Access to this rejection message property provides an opportunity to adjust a rejection messages with additional details so those changes can appear in the log window, and in the **Filter Log.csv** file.

Adding message details to the Filter.Log can improve understanding about why an order that should not have been rejected, or specific order rejection that is a problem, can be accessed and adjusted before the rejection-message is added to the Filter.Log.

Order Rejection Conditions Example:

```

' ~~~~~
' Test Quantity Size: Reject Zero Qty Orders - See Filter Log
' -----
' 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.
' -----
' When the Quantity is less than the Instrument's Round-Lot value,...
If order.quantity < instrument.roundLot THEN
' Place a rejection reason record in the Trading Blox Filter Log.
If dollarRisk <= 0 THEN
' Calculate Risk Value
dollarRisk = riskEquity/Max(dollarRisk, 1)
' Generate Insufficient Equity Error Message
sMsg = "Qty:" + AsString(dollarRisk, 1 ) _
      + " < Minimum-Round Lot: " _
      + AsString( instrument.roundLot, 2 ) _
      + " Risk Eq: " + AsString( riskEquity,2 ) _
      + " Order-Risk: " + AsString( dollarRisk, 2 )

' When Added Calculation Size is Zero Error,...
If iErrQty THEN
sMsg = sMsg + " Qty-Error = " + AsString(iQty, 1)
ENDIF
ELSE
' Generate Below Round Lot Size Error Information
sMsg = "Qty: " + AsString( iQty, 2 ) _
      + " < Min-Lot: " _
      + AsString( instrument.roundLot, 2 ) _
      + " Eqty: " + AsString( riskEquity,2 ) _
      + " Risk: " + AsString( dollarRisk, 2 )

' When Added Calculation Size is Zero Error,...
If iErrQty THEN
sMsg = sMsg + " Qty-Error = " + AsString(iQty, 1)
ENDIF
ENDIF

' The Order.Reject function is given a message
' when an order is flagged for rejection.

' Send Order Rejection Message
order.Reject( sMsg ) ' <=== Order Rejection Function
ENDIF
' ~~~~~

```

When a rejected order happens, this **Filtered Order Notification** script section is executed so the rejection message can be seen at the time of an order rejection. When there are scripts in this section, those script will execute and the **PRINT** statements in the script will send the rejection information to the Main Screen's Log window display.

Log Message Script Example:

```

' =====
' FILTERED ORDER NOTIFICATION EXAMPLE
' SCRIPT - START
' =====
' ~~~~~
' Order rejection when a Filtered Order Notification Script Section
' included in a blox connected to a System, the order rejection
' messages will be displayed in the Main Screen's Log window when
' the Log window is open.
' ~~~~~
' When the Order is an Entry,...
If order.isEntry THEN
' -----
If iRejectionTitles = FALSE THEN
'   Send Column Titles to Log
PRINT "Test Date",_
        "Rejected",_
        "Reason",_
        "Symbol",_
        "Price",_
        "Qty",_
        "Stop"

'   Send Column Titles only Once
iRejectionTitles = TRUE
ENDIF
' -----
'   Display this event information
PRINT test.currentDate,_
        "Rejected",_
        order.processingMessage,_
        order.symbol,_
        order.orderPrice,_
        order.quantity,_
        order.stopPrice
' -----
ENDIF
' ~~~~~
' =====
' SCRIPT - END
' FILTERED ORDER NOTIFICATION EXAMPLE
' =====

```

Example Messages:

Log Message Script Example:

```

Log Window → Log Window in Floating Mode
Starting Simulation at 14:11:29
Test Date Rejected Reason Symbol Price Qty Stop
20170103 Rejected Average volume of 0.00 is less than requi
20170103 Rejected Qty: 0.38 < MinLot: 1.00 EqtyAmt: 4000.01
20170103 Rejected Average volume of 43.00 is less than requ
20170104 Rejected Average volume of 10.00 is less than requ
20170104 Rejected Qty: 0.37 < MinLot: 1.00 EqtyAmt: 4019.68
20170104 Rejected Average volume of 37.67 is less than requ
20170105 Rejected Average volume of 6.67 is less than requi
20170105 Rejected Qty: 0.38 < MinLot: 1.00 EqtyAmt: 4080.56
20170105 Rejected Average volume of 52.44 is less than requ
20170106 Rejected Average volume of 9.78 is less than requi
20170106 Rejected Qty: 0.36 < MinLot: 1.00 EqtyAmt: 4020.57
20170106 Rejected Average volume of 34.96 is less than requ
20170109 Rejected Average volume of 6.52 is less than requi
20170109 Rejected Qty: 0.35 < MinLot: 1.00 EqtyAmt: 4019.14
20170109 Rejected Average volume of 41.31 is less than requ
20170110 Rejected Qty: 0.35 < MinLot: 1.00 EqtyAmt: 3983.84
20170110 Rejected Average volume of 51.54 is less than requ
20170111 Rejected Qty: 0.35 < MinLot: 1.00 EqtyAmt: 4006.75
20170111 Rejected Average volume of 40.36 is less than requ
20170112 Rejected Average volume of 90.82 is less than requ
20170112 Rejected Qty: 0.34 < MinLot: 1.00 EqtyAmt: 4005.77
20170112 Rejected Average volume of 71.91 is less than requ
20170113 Rejected Average volume of 60.55 is less than requ
20170113 Rejected Qty: 0.33 < MinLot: 1.00 EqtyAmt: 3966.91
20170117 Rejected Average volume of 40.36 is less than requ
20170117 Rejected Qty: 0.33 < MinLot: 1.00 EqtyAmt: 3927.30
20170118 Rejected Average volume of 27.24 is less than requ
20170118 Rejected Qty: 0.32 < MinLot: 1.00 EqtyAmt: 3946.50
20170119 Rejected Average volume of 18.16 is less than requ
20170119 Rejected Qty: 0.31 < MinLot: 1.00 EqtyAmt: 3905.89
20170120 Rejected Average volume of 12.11 is less than requ
20170120 Rejected Qty: 0.30 < MinLot: 1.00 EqtyAmt: 3852.60
20170123 Rejected Average volume of 8.07 is less than requi
20170123 Rejected Qty: 0.31 < MinLot: 1.00 EqtyAmt: 3888.89
20170124 Rejected Average volume of 5.71 is less than requi

```

Floating Log Window Message Example

This display shows the Log Window when it is pulled from the main screen using the mouse and placed away from how it typically appears at the bottom of the System Parameter Display on the Main Screen.

To return the Log Window to its position at the bottom of the main screen, reduce the height of the Log Window and grab the title bar as you move it down to the bottom of the Main Screen. As you get within reach of the bottom, a button will appear with a down arrow that will allow you to click so that it pops back into its original location.

Portfolio Manager Direction Conditions:

The Trading Blox Builder **MACD Portfolio Manager** and others use script to control in which the Entry Orders cannot create an order that will be able to go in a **Long** or a **Short** direction.

When this happens in a system, and the Entry Order script in the system does not check to see if the **Portfolio Manager** has blocked a **Long** or **Short** order, the order direction created the Portfolio Manager intended to block, will cause a Portfolio Manager Rejection message when the [Broker](#) Function attempts to create the order.

This automatic rejection of a huge amount of orders that should not have been created in the first place, will fill the Log Window Order Rejection display. This is a problem when the task of watching order rejections is to find a problem quickly. In many cases, there wouldn't be any rejected orders if the Entry Orders script section had tested the symbols properties to see if the Portfolio Manager had blocked orders in the direction that the Entry Order section will attempt to create an order.

To stop rejecting orders the Portfolio Manager blocked order direction, the fix is simple:

```

Entry Orders      Turtle SingleUnit Entry Exit
1
2  ' Set the stop width
3  stopWidth = stopInATR * averageTrueRange
4
5  ' If we are not long (we are short or out) then place orders for tomorrow
6  IF instrument.position <> LONG AND instrument.canTradeLong THEN
7      broker.EnterLongOnStop( offsetAdjustedEntryHigh, _
8                              offsetAdjustedEntryHigh - stopWidth )
9  ENDIF
10
11 ' If we are not short (we are long or out) then place orders for tomorrow
12 IF instrument.position <> SHORT AND instrument.canTradeShort THEN
13     broker.EnterShortOnStop( offsetAdjustedEntryLow, _
14                              offsetAdjustedEntryLow + stopWidth )
15 ENDIF
16

```

Portfolio Manager Rejection Change to Disable Filter Trade Log Messages

Allowing automatic rejection of orders will fill the Filtered Log window with so much information that shows the Portfolio Manager didn't intend an order be created with that symbol in the direction today it is being blocked. Take a look at the above example. It shows how the current the **Donchian's Turtle SingleUnit Entry Exit** block to show how easy it will be to remove those **Portfolio Manager** Rejections.

In the above image, the script-text in each red box outline will prevent an order in a direction that the Portfolio Manager attempted to prevent. It can stop the rejections because the [Trade Control Properties](#) (click to see the trade control property details).

When an Entry Orders statement test the status of these trade-direction properties, the orders in the direction the Portfolio Manager blocks will not be created.

6.16 Before Order Execution

This script section called once for each test day. It becomes available just before orders are used to creating market signals so that orders can be sorted, adjusted, removed or added.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 163

6.17 Update Indicators

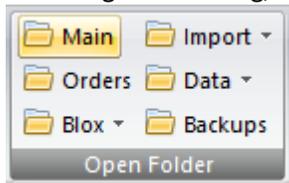
This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

The **Update Indicators** script is designed to run each instrument in the system that is enabled and **Primed**. This means script statements will have the ability to run scripts before the **Suite's Test Start Date** is reached. This earlier execution will happen when the default execution setting in the **Trading Blox.Ini Block Basic Group** parameter shows:
Allow Priming Update Indicators=TRUE

All new installations of Trading Blox Builder will show the above setting set to TRUE. That setting is what enables **Update Indicator** script section to execute script statements for each instrument that shows the **Primed** property is set to TRUE.

However, if your Trading Blox Builder installation has been updated over many versions using the **Main Menu's Help** tab option "**Check for New Version**", that setting might show: "**Allow Priming Update Indicators=FALSE**". When the setting is **FALSE**, early **Primed Before Test Start Date** calculations are disabled.

To change the setting, open the folder where Trading Blox Builder is installed.



Main Menu - File Menu - Open
Folder Group items

Click on the **Main** button to open a dialog that displays all the files and folders in the TB installation folder. Exit Trading Blox Builder

before you make and changes to the file. If Trading Blox Builder is left open, it will over-write any changes made to the file. Scroll down to the group of files near the bottom of the installation folder the file until you find: **Trading Blox.Ini**. The **Trading Blox.Ini** file is a text file that will open with any editor like Windows **Notepad.exe**.

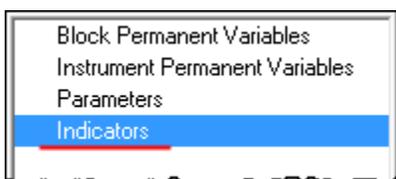
Open the file and scroll down until the **Block Basic** group is found. In my NotePad++ editor, the [Block Basic] group is on line 209 in a file where the max line count is 254. So the scrolling from the

top down is longer than the jump down, Ctrl-End key combination will be longer to get to [Block Basic] parameter **Allow Priming Update Indicators** on line 224:

```
[Block Basic]
Keyword Color=16711680
Function Color=0
Operator Color=0
Comment Color=37632
String Color=65535
Trading Object Color=4718664
Built-In Color=2124031
Temporary Color=0
Block Permanent Color=9109504
Instrument Permanent Color=9109504
Indicator Color=9109504
Parameter Color=13781
Include Default Scripts=FALSE
Automatic Blox Script Parsing=TRUE
Allow Priming Update Indicators=TRUE
Blox Editor Font Size=14
Blox Editor Font=Courier New
Use Jelly Fish Editor=FALSE
Blox Directory=
Parameter Tab Size=6
```

Trading Blox.Ini - Block Basic group

In the group of parameters, if the **Allow Priming Update Indicators** shows **FALSE** and you want the system's Custom Scripted Indicators to start calculations as soon as they are primed, change the setting to **TRUE**. Save the file and restart **Trading Blox.exe** with a double-click that appears in the open dialog.



Data Types and Built-In Indicator Sections

This ability is essential because all built-in **Indicator Section** indicators are primed and calculated with access to all of the instrument's data as soon as an instrument is primed. This **Update Indicator** ability allows **Custom Scripted Indicators** to access instrument data as soon as each instrument is **Primed**. In simple terms, this **Update Indicators** script section provides the same data access ability to the **Custom Indicators** as is built-in the indicators selected in the **Indicator Section**.

This is also true for Intraday records that must process the same date's Intraday time records. The first **Intraday Time Value** to process will be the first time record in the time record series of each record date.

If you update your custom indicators in this script section the values will be available to use in all other scripts and blox, like the [Adjust Stops](#) script, After Instrument Day script, [Before Instrument Day](#), and of course the [Entry Orders](#) and [Exit Orders](#) scripts. To access provide access to other blox in the same system set the property's Scope to System.

Indicator Priming - Chart Plotting:

If you want to see indicators plot during the period when an instrument is priming, you will need to enable the ability in the Update Indicators script section so the "**Allow Priming Update Indicators=TRUE.**"

NOTE:

This scripts section will not normally execute when a instrument specific date record is not in the instrument's data file.

A current record is a record date for the same test date on which this script is executing. This means a test date must have an instrument date price record in order for this script section to execute.

When it does have a missing instrument record any orders that were active for the active test date, orders will not be filled or expired by the brokerage process. However, when an order is still active after the brokerage process, it will be available to execute an order fill or allowed to expire on the next available instrument date trading session.

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 640

6.18 Can Fill Order

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

Called as part of the fill process, the *Can Fill Order* script let's 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

- o **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
```

NOTE:

This script section will not normally execute when a instrument specific date record is not in the instrument's data file.

A current record is a record date for the same test date on which this script is executing. This means a test date must have an instrument date price record in order for this script section to execute.

When it does have a missing instrument record any orders that were active for the active test date, orders will not be filled or expired by the brokerage process. However, when an order is still active after the brokerage process, it will be available to execute an order fill or allowed to expire on the next available instrument date trading session.

6.19 Exit Order Filled

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

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 Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

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
  ' Process Exit Filled Calculations when needed.
EndIF
```

NOTE:

This scripts section will not normally execute when a instrument specific date record is not in the instrument's data file.

A current record is a record date for the same test date on which this script is executing. This means a test date must have an instrument date price record in order for this script section to execute.

When it does have a missing instrument record any orders that were active for the active test date, orders will not be filled or expired by the brokerage process. However, when an order is still active after the brokerage process, it will be available to execute an order fill or allowed to expire on the next available instrument date trading session.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 321

6.20 Entry Order Filled

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

This **Entry Order Filled** script is called each time an entry order is filled. It lets the user to perform any calculations or adjustments, or other 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.

NOTE:

This scripts section will not normally execute when a instrument specific date record is not in the instrument's data file.

A current record is a record date for the same test date on which this script is executing. This means a test date must have an instrument date price record in order for this script section to

execute.

When it does have a missing instrument record any orders that were active for the active test date, orders will not be filled or expired by the brokerage process. However, when an order is still active after the brokerage process, it will be available to execute an order fill or allowed to expire on the next available instrument date trading session.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 300

6.21 After Instrument Open

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

NOTE:

This scripts section will not normally execute when a instrument specific date record is not in the instrument's data file.

A current record is a record date for the same test date on which this script is executing. This means a test date must have an instrument date price record in order for this script section to execute.

When it does have a missing instrument record any orders that were active for the active test date, orders will not be filled or expired by the brokerage process. However, when an order is still active after the brokerage process, it will be available to execute an order fill or allowed to expire on the next available instrument date trading session.

Indicator calculations are updated to the data date being used for test date. The instrument date is equal to the test date in the **After Instrument Open** after the Open. This means the **OHLC** and all indicators for the the test date are updated and available.

From a test date perspective, the [Entry Orders](#) script runs using the instrument data prior to the test date. Then the access within the After Instrument Open script follows these rules:

1. The last nearest Instrument date is brought forward to the active test date.
2. [Update Indicators](#) script updates all custom indicators are synchronized with the OHLC and the regular and calculated indicators.
3. **After Instrument Open** script runs with access to all this test date data.
4. As mentioned, active trading orders on order generation day are run using the very last bar.
5. The the data process stops at the [Before Order Execution](#) script.
6. All the next day orders are computed for the test date and are output to the order report.
7. All the subsequent scripts that require the trade day data are not run.

WARNING:

The **After Instrument Open** script is an advanced script that is not recommended for most users. It is available 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 in the instrument file is available. For this reason, it is possible to write systems with postdictive information that are unavailable in live trading. They are unavailable in live trading because the market is still open, but the file data that is the source of the instrument prices has not been released and loaded into the data file.

In simple terms, making decisions using postdictive information that is not be available in active trading when the market is strill trading leaves a data hole in the strategy. . If you want to create orders which are based on a bar's open, use the [tradeDayOpen](#) instrument property in the Entry Orders script.

6.22 After Bar

This script does **NOT** have automatic instrument context by default.

Instrument access can be achieved using the [LoadSymbol](#) Function.

Available at the end of a price bar in Intraday data file test.

Edit Time: 9/11/2020 4:48:24 PM

Topic ID#: 135

6.23 Adjust Stops

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

Adjust Stops will always execute when an instrument has an active position. Script executes after all the instruments with active positions have been updated with the new trade date information.

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

NOTE:

Script will execute for each instrument in the portfolio on each test date.

Executing this script when the instrument record is earlier than the current test date will find instrument property values will contain data from the nearest earlier instrument record.

To know if there is a instrument date that can match the current test date use the [instrument.tradesOnTradeBar](#) property. When the current test matches the instrument date this property will return **TRUE**. When the instrument date is earlier this property will return **FALSE**.

Use the [instrument.tradesOnTradeBar](#) property when it is important to know, or to not allow calculation changes to this instrument when test and instrument dates do not match.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 129

6.24 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:

```
' Initialize Total Value for each time this script is executed.  
totalRisk = 0
```

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 384

6.25 Compute Instrument Risk

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

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
```

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 215

6.26 Compute Risk Adjustment

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

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.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 216

6.27 Adjust Instrument Risk

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

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
```

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 130

6.28 After Instrument Day

This script has Automatic Instrument Context by default. This means, when this script section executes, it is given direct access to all **IPV** Properties and Functions.

Script is called before the After Trading Day script section. Script executes on every test date for each instrument in the portfolio where a date record exist and that instrument's property reports `instrument.isPrimed = TRUE`. This script can be an alternate location to calculate instrument-specific variables or custom indicators each day.

NOTE:

Script will execute for each instrument in the portfolio on each test date.

Executing this script when the instrument record is earlier than the current test date will find instrument property values will contain data from the nearest earlier instrument record.

To know if there is a instrument date that can match the current test date use the `instrument.tradesOnTradeBar` property. When the current test matches the instrument date this property will return `TRUE`. When the instrument date is earlier this property will return `FALSE`.

Use the `instrument.tradesOnTradeBar` property when it is important to know, or to not allow calculation changes to this instrument when test and instrument dates do not match.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 205

6.29 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 Automatic-Context access to instrument properties or Instrument Permanent variables.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 208

6.30 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.

Note:

Before Test, Set Parameters, and After Test to be thread locked so the script for each thread will complete without overlap from another thread.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 207

6.31 After Simulation

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.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 206

6.32 Post Process Utility

This new "**Post Process Utility**" script section runs after all reports and files are processed and closed.

In this script section the **PDF** file that contains the suite testing reports can be sent as a Summary Results document. The timing of this script, makes it safe to attach the information to an email, or to create a copy that can be named with date and time information as necessary. Use the sample code below as an example of how to handle the output that is available and can be added.

Example:

```

VARIABLES: reportPath, messagebody Type: String

' Use the PDF version to attach to email as it contains all the graphics
and charts.
reportPath = ReplaceString( test.summaryReportFilePath, ".htm", ".pdf" )

If NOT EmailConnectSSL( "mail.tradingblox.com", _
                        + "Customer
Testing<customertesting@tradingblox.com>", _
                        + "Customer
Testing<customertesting@tradingblox.com>", _
                        + "customertesting+tradingblox.com", _
                        + "xxx", 587) THEN ERROR "Unable to connect"

' Send a simple email with the report as an attachment.
' IF NOT EmailSend( "tim@tradingblox.com", "Summary Report",
' "", "", "", reportPath ) THEN ERROR "Unable to send"

' Add the equity chart as an image. Only images can be
' added for use in the message body html.
' This first image will be referenced by cid:message-root.1
If FileExists( test.resultsReportPath _
                + "\LinearEquityDrawdownGraph_P1.png" ) THEN
    EmailAddImage( test.resultsReportPath _
                    + "\LinearEquityDrawdownGraph_P1.png" )
ENDIF

' This second image will be referenced by cid:message-root.2
If FileExists( test.resultsReportPath + "\MonthlyReturnsGraph_P1.png" )
THEN
    EmailAddImage( test.resultsReportPath + "\MonthlyReturnsGraph_P1.png" )
ENDIF

messagebody = "<HTML>Here is the <STRONG>Equity Chart:" _
              + "</STRONG><BR><BR><IMG SRC='cid:message-root.1'>" _
              + "<BR><BR>AND here is the <STRONG>Monthly Returns " _
              + "chart:</STRONG><BR><BR><IMG SRC='cid:message-root.2'>"

' Send the HTML message with the message body referring to each attached
image. More than one can be included, and the HTML can be as complex as
necessary.
If NOT EmailSendHTML( "tim@tradingblox.com", _
                        + "Summary Report Graph", messagebody, "", "",
reportPath ) THEN ERROR "Unable to send"

EmailDisconnect

```

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:
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.
Data Variables	In Blox Basic you can create variables which have names and can hold values just like a spreadsheet cell can.
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
Indicator Pack 1	Additional indicators and functions are available in this section.
Indicator Reference	You can use a variety of indicators in a trading system. Trading Blox provides a long list of available indicator methods and function. All of these can easily be selected and used as a standard indicator for display, or as component to a custom indicator in a system. Built-in functions can also create a value in a custom indicator you create.
Operator Reference	Shows the types of operators and expressions you can write in a Blox Basic script
Script Problem Information	This section contains information for solving some of the more common script problems.
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.

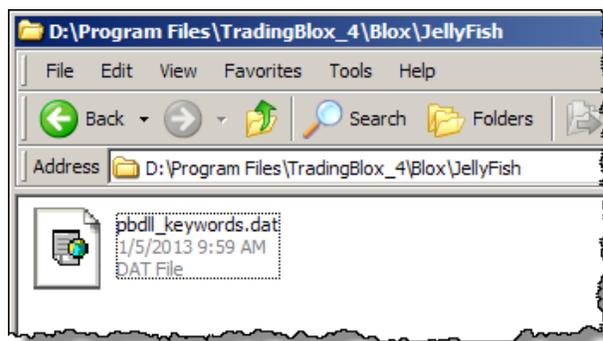
Basic KeyWords Listing:

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 – Data Variables

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

Section Name:	Description:
About Data Variables	Brief outline of how Trading Blox uses variables.
Data Comments	Method for assigning comments to scripts.
Data Groups	List of Trading Blox Data Groups.
Data Names	List of Trading Blox
Data Scope	Data value information reach in a Blox, System, or Suite
Data Types	Types are Numerical, Boolean, String/Text, and Series/Array variables.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 255

2.1 Colors

Selecting, applying and changing colors used on a chart, or graph is under the users control methods.

Chart area of display colors 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. Multiple colors can be assigned to a single indicator based upon the script logic created to control color assigned.

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.

Control of the plot color of both series is the same. To color a plot series, use the Color option in the Plot section of the series creation dialog, or assign the color using the above function.

When using the Plot section in the series creation dialog, the Colored-button is clicked. A matrix table of common colors will display. Select a color displayed and click OK. If the color you want, click the "**More Colors...**" button to display additional color option methods.

When the "**Custom Color Selection Dialog**" 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 is 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

Color Property Name:	Default Color Number:
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

```

' ~~~~~
'   R , G , B
ColorItem[ 1 ] = ColorRGB( 0 , 0 , 255 ) ' Blue
ColorItem[ 2 ] = ColorRGB( 255 , 0 , 0 ) ' Red
ColorItem[ 3 ] = ColorRGB( 0 , 255 , 0 ) ' Green
ColorItem[ 4 ] = ColorRGB( 0 , 0 , 0 ) ' Black
ColorItem[ 5 ] = ColorRGB( 49 , 133 , 155 ) ' Aqua
ColorItem[ 6 ] = ColorRGB( 0 , 0 , 192 ) ' Dark Blue
ColorItem[ 7 ] = ColorRGB( 84 , 141 , 212 ) ' Light Blue
ColorItem[ 8 ] = ColorRGB( 227 , 108 , 9 ) ' Orange
ColorItem[ 9 ] = ColorRGB( 235 , 117 , 123 ) ' Coral
ColorItem[ 10 ] = ColorRGB( 94 , 162 , 38 ) ' Dark Green
ColorItem[ 11 ] = ColorRGB( 95 , 73 , 122 ) ' Purple
ColorItem[ 12 ] = ColorRGB( 160 , 106 , 88 ) ' Brown
ColorItem[ 13 ] = ColorRGB( 255 , 255 , 0 ) ' 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 ]

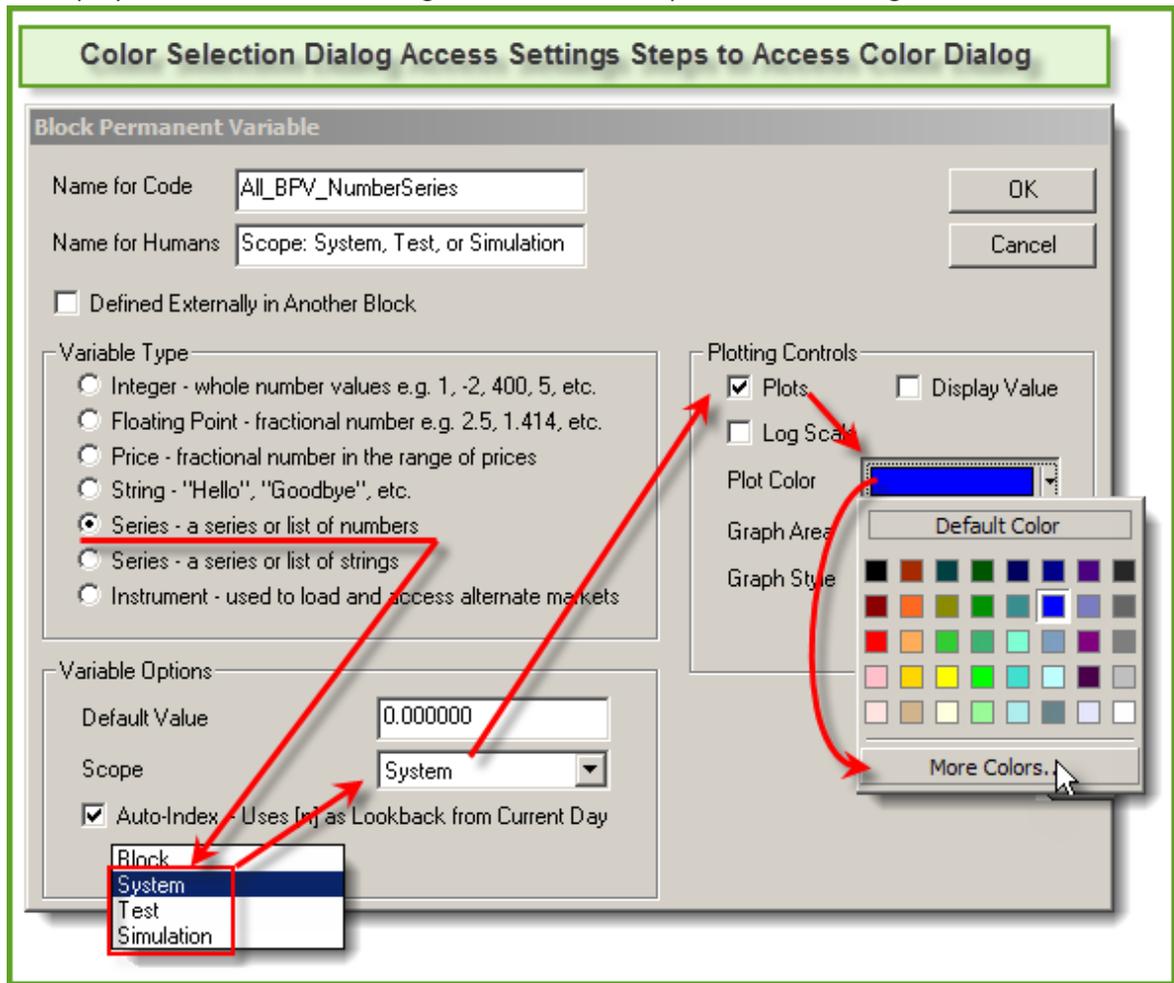
' Place this system's test-date total equity percentage net change
' value in the chart space using the new color
chart.AddLineSeries( AsSeries(systemEquity), elementCount, _
                    alternateSystem.name, plotColor )

```

Trading Blox Color Selection Dialog:

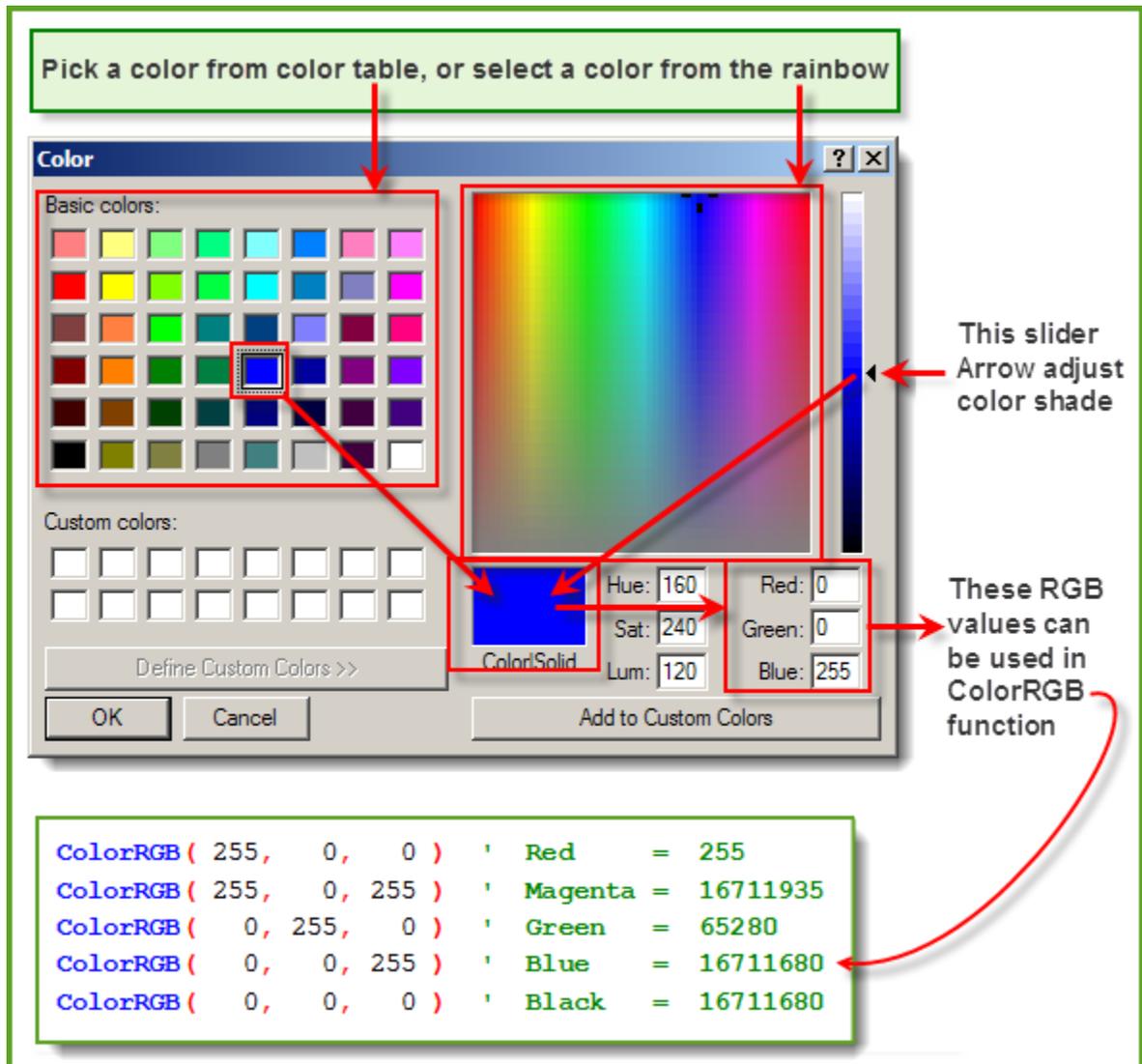
All BPV Series will provide access to the Trading Blox Color Selection Dialog when the BPV series uses a System, Test or Simulation Scope setting with a BPV numeric series.

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



Color Selection

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



Color Selection Options

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 the Trading Blox [ColorRGB](#) function requires, place the color numbers using Blue, Green and Red as the first, second and third parameter locations

Script Color Assignment Examples:

```

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

2.2 Constants Reference

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

Constant Name:	Constant Value:
PI	3.14159265358979321
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

Trading Blox Preferences:

Preference Setting Values:	Description:
CHARTNOVALUE	Constant. Assign to an IPV plotting series when no value is to be plotted on the instrument chart.
LoadUnadjustedClose	TRUE / FALSE value.
LoadVolume	TRUE / FALSE value.
NumberOfExtraDataFields	Number value of the Extra Data Fields setting field.
ProcessDailyBars	TRUE / FALSE value.

Preference Setting Values:	Description:
ProcessMonthlyBars	TRUE/FALSE value.
ProcessWeekends	TRUE/FALSE value.
ProcessWeeklyBars	TRUE/FALSE value.
RaiseNegativeDataSeries	TRUE/FALSE value.
UNDEFINED	Constant. When assigned to an IPV series, the negative value will be outside of the plotting range of the instrument chart.
YearsOfPrimingData	Number value of the Extra Data Fields setting field.

Using constants improves code understanding.

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
```

2.3 About 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.

Variable Type Names:	Descriptions:
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.
Series	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](#)

2.4 Data Comments

Methods available to annotate your scripts so they will be understandable later. Data comments are sections of code that are intended for human reading, and are ignored by Blox Basic script interpreter.

Commented scripts will help you document as you are creating it, and provide an aid in understanding the code later on. These same 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!
```

2.5 Data Groups

Trading Blox provides four data group sections where the different data types of data a group can be applied to how variables share the variables information.



Data Group Name Types



User Created Local Variables

Group Types:	Description:
Block Permanent Variables - (BPV)	All Block Permanent Variables are the same regardless of instrument that is being processed. They retain their value in all the scripts, and are accessible from any script section. two dimensional bpv string arrays, works the same way as two dimensional number arrays
Instrument Permanent Variables - (IPV)	Instrument Permanent variables can contain a different value for each instrument. Use these when you need the value to be available when an instrument is in context. Instrument Permanent variables are only accessible using normal means when the script section has obtained instrument context. Instrument Permanent variables can be brought into context in any script using the Block Permanent variable: <symbol-name>.LoadSymbol(reference) method.
Parameters	Parameters create a powerful feature for controlling systems. They allow a user with the ability to create systems that present user interface controls Most Parameters enable easy value setting value that can be incremented so a series of a stepped series of test can generated. These stepped variables are the primary means control of how Stepped Optimizations are control, and how many iterations will be generated.
Indicators	Indicators are a form of Instrument Permanent variables and follow the same access. Indicators are created using the Indicator Wizard process that makes custom indicator creation easy and fast.
Local Variables	Local variable scope is limited to the script section in which the User Created Local Variable is created. Each Local variable must be declared in the script section where they are to be used. Any value assigned to a Local variable is can only be accessed within script section where it is declared using the Variables Function. Local are not cleared. This means the value in the variable is not cleared to zero or a value unless the user provides the scripted

Group Types:	Description:
	code to clear or seed a value that variable will need. and cleared from these variables must be part of the sections scripting statements.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 253

Block Permanent Variables

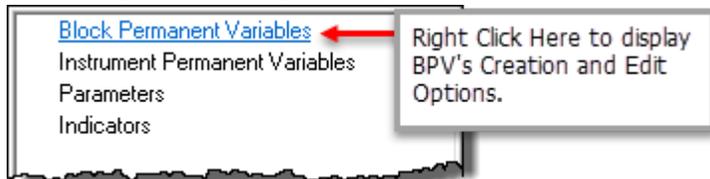
Block Permanent Variables (**BPV**) have the same value regardless of instrument.

At the start of a test, all **BPV** variables are initialized to a user specified value. The default value is zero. This initialization value is retained until the value changed by scripts when a test performed.

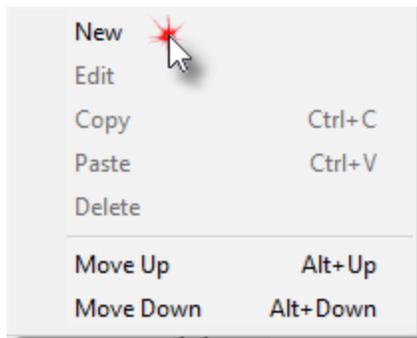
All **BPV** variables are accessible to all the block script sections in a block. That access can be increased to other blocks when the scope setting for a **BPV** variable selects a scope value that is not [Block](#). **BPV** variables work faster than script-section created [Local Variables](#). **BPV** variables initialize to the established default value that was entered when the **BPV** variable was created. BPV Initialization happens in the Before Test script section at the beginning of a test.

The initialization value, or the latest assigned value in a **BPV** variable is the same regardless of which instrument is the focus of a test. For example, the scripting assigns a day value to a **BPV** value early in the test process, that value will be the same for all the instruments being tested and will stay the same until scripting is used again to change the value.

Creating & Editing BPV Variables:



Block Basic Editor's BPV Access Menu



Block Basic Editor's BPV Menu Options

Double-Clicking on the **Block Permanent Variables** menu option above, or by Right-Clicking the **Block Permanent Variables** will display [BPV Variable Editor Dialog](#) shown next.

BPV Variable Editor Dialog:

Block Permanent Variable (BPV) Editor

Variable Script Name & Display Name:	Descriptions:
Script Name	<p>This is the name seen in the BPV Variable List section and in the scripted statements where it is used.</p> <p>Variable Name Rules: Variable names cannot have a space or any of the special characters in the name. However, an underscore character "_" are allowed.</p> <p>Use variables names that make sense to you when you create them or change them. When a variable name is changed, it will need to be change every place where it is used in script sections where it is used. Variable names that are descriptive will improve the understanding of why they are created and applied. Descriptive names also help to reduce the mystery of a statement's purpose. For example, if you want the variable to hold the current day-of-month value</p>

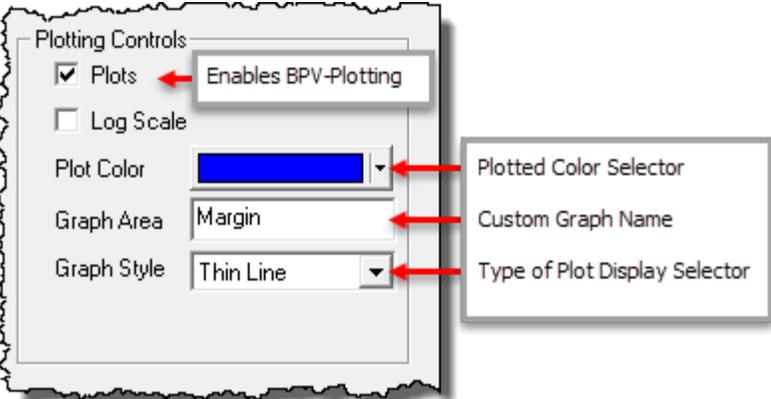
Variable Script Name & Display Name:	Descriptions:
	<p>(dayInMonth), that name reflects the variables purpose. Better understanding of a variable's purpose is a key factor in preventing and reduce mistakes.</p> <p>Trading Blox Builder Common naming convention:</p> <ul style="list-style-type: none"> • Use a Lower-Case letter as the first character in a name for Object Property variables. • Uses an Upper-Case first letter for functions.
Display Name	<p>This field is optional, but when it is used, consider using text that provides a description of the BPV variable's purpose.</p> <p>This field is displayed in the variable declaration section of the block, when it is Previewed or Printed using the option in the Editor's File menu.</p>
Defined Externally in Another Block	<p>This information explains the: Defined Elsewhere option.</p>

Variable Types:	Descriptions:
Integer	Any Integer numeric value (e.g. 1, -2,400, 5, etc.)
Floating Point	Any Decimal/Fractional numeric value (e.g. 2.5, 1.414, etc.)
Price	Any instrument fractional number in the symbol's range of price values
String	<p>Any keyboard character. Value passed as letters must be bounded by quote marks (e.g. "Hello", "Goodbye", etc.)</p> <p>two dimensional bpv string arrays, works the same way as two dimensional number arrays</p>
Numeric Series	<p>All data elements in a Numeric series are a Floating Point variables. A series, also know as an array, are an indexed list of numeric value locations that can hold the same value, or a different a value in each of the series elements. Auto-Index Series match the instrument's date information and use the same index offset that the instrument's data use. i.e., an instrument.close[0], represents the current close price. An instrument.close[1] represents the previous date's close price. That same offset number process is how the information can be accessed in an Auto-Index numeric series.</p> <p>A numeric series can be Manually-Index. Manually-Indexed series require the user to determine the size of the series, this means how many variable places will</p>

Variable Types:	Descriptions:
	<p>be needed, and how to access the series requires the user to create the indexing and searching scripts to access information.</p> <p>Numeric Series/Arrays, can only contain a list of numbers. These can be tied to the test day using the "Auto-Index" feature described below. They can also be a fixed manually sized series that can be expanded or reduce using scripting. Fixed series are access by an index that is managed by scripted statements.</p>
String Series	<p>All data elements in a String series are a text character containers. A String Series, also know as an array, are a list of String locations that can hold the same, or a different Strings in each of the element locations.</p> <p>Numeric Series/Arrays, can only contain a list of numbers. These can be tied to the test day using the "Auto-Index" feature described below. They can also be a fixed manually sized series that can be expanded or reduce using scripting.</p> <ul style="list-style-type: none"> - a series or list of strings <p>A Series of Strings can be used for many purposes, but also as the x axis label of custom charts.</p>
Instrument	<p>A BPV Instrument type variable is a series variable that emulates the properties and methods associated with all the Instrument Permanent variables (IPV). In simple terms, they become containers for an instrument's data, properties and functions so the instrument can be accessed in a script section that does not automatically provide instrument access.</p> <p>A BPV Instrument type variable provides a container that allows the LoadSymbol function to get an instrument so the instrument's data, properties and functions can be accessed and used.</p> <p>To create a BPV Instrument, the user selects Variable Type Instrument and gives it a name. Fpr tjos example, I'll use the name Mkt. I could have any other name like Corn, Bonds, or inst. The only limitation in naming is to not use a Trading Blox Builder Keyword. The name selected will act like an Object reference that enables access to the instrument contents.</p> <p>A BPV Instrument can be used in a For-Next loop to add information to all or many of the instrument's in the portfolio. More than one BPV Instrument can be used to calculate differences, or change some property values.</p> <p>Click on this link to learn how to use the LoadSymbol function.</p> <p>The Trading Blox Builder Roundtable Blox Market Place has more examples that show different purposes for loading IPV instrument information into a BPV instrument.</p>
Note:	

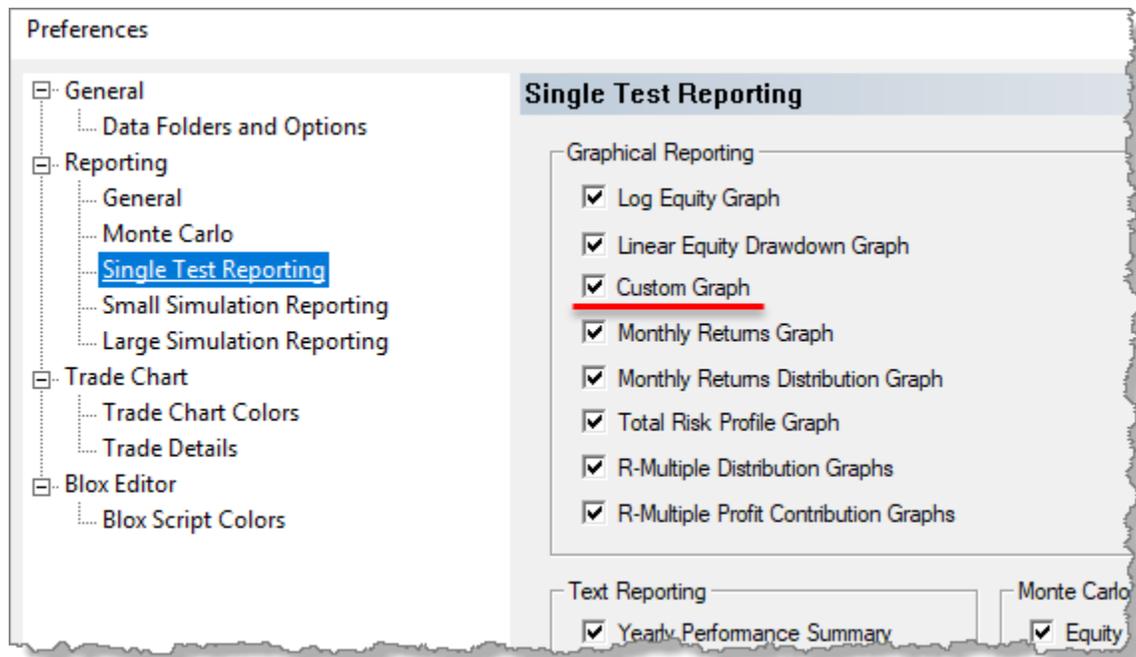
Variable Types:	Descriptions:
<p>Variable Names and Types cannot be changed using scripts. To make a change to either, that change must be done using the BPV Variable Editor. There is more information In the About Data Variables topic and some variable use examples.</p>	

Variable Options:	Descriptions:
Default Value	The value entered into this field will be the initialized value this variable will have at the beginning of a test. Initialized values and be changed by script statements or functions.
Scope	<p>The selected Data Scope determines where this variable can be accessed and changed.</p> <p>BPV variables are allowed to use the following scope settings:</p> <ul style="list-style-type: none"> • Block -- This scope allows the variable to accessible only within the blox where it is declared. • System -- Allows this variable to accessible from within any of the blocks in the System where this block is listed. • Test -- Allows this variable to be accessible in any of the block in any of the systems within the testing Suite. This scope is the same as Test scope, except the value will not be reset when the Suite is performing a multi-step parameter test. This means the last value of this variable will be the value that appears at the beginning of the next parameter testing step.
Auto-Index Series	<div style="border: 1px solid gray; padding: 5px; margin-bottom: 10px;"> <p><u>Variable Options</u></p> <p>Default Value <input type="text" value="0.000000"/></p> <p>Scope <input type="text" value="System"/></p> <p><input checked="" type="checkbox"/> <u>Auto-Index -- Uses [n] as Lookback from Current Bar</u></p> </div> <p style="text-align: center;">BPV Series Indexing Options</p> <p>A BPV Series with Auto Index checked, will track the test.currentDay index. When <u>Auto-Index</u> is NOT Enabled, the series will be a static series that is managed and a accessed with scripting.</p>
Plotting Controls	<p>This option will not appear unless the BPV numeric-series type is selected and is using System-Scope setting.</p> <p>The numeric variable type that has a System scope setting will display this Ploting Control area where the user can create a chart graphic to display the numeric series information:</p>

Variable Options:	Descriptions:
	 <p data-bbox="638 745 1114 772">Appears when a BPV is System-Scope, or Test-Scope.</p> <ul data-bbox="480 814 1461 1388" style="list-style-type: none"> • The color of a graphed plot is assigned using the Plotted Color Selector. • The graph name where the plotted information is to appear is determined by the name entered into the Graph Area field. • The type of plotted display is based upon the selection appearing in the Graph Style selector. • When more than one series is being plotted in a custom graph, and one of those series has enabled "Log Scale" option, then all series plotting in the same Graph Area will plot as log scale. • When you plot more than one BPV variable in a custom graph, be sure the ranges of the all the series is within a reasonable range for all the series being plotted in that same graph area so the display is readable and consistent. • When a blox is used to create a custom graph, and that blox is inserted in multiple systems, there will be multiple plotted graphs displaying information from each of the systems. This is also true when you use multiple plotting statements in in your test, then multiple Summary Custom Charts will be created for each one.
Reset Before Test	

Plotting BPV Series variables on the Summary Custom Chart

Be sure to enable the **Custom Graph** check-box in the **Preferences' Single Test Reporting** section. See the Trading Blox Builder **User's Guide** manual **Reporting Information's Graphical Reporting** topic for more information.



BPV Series Preference Custom Graph Ploting Option

Graph Plotting:

To plot a BPV-Series variable, the defined Scope setting must select either the [System, or the Test scope](#) option. This setting will enable the BPV-Variable Editor to display the [Plotting Control](#) area.

Block Permanent Variable [Block: EndOfWeek Test Idea | Group: _Help]

Script Name

Display Name

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

Plotting Controls

- Plots
- Log Scale
- Plot Color
- Graph Area
- Graph Style

Variable Options

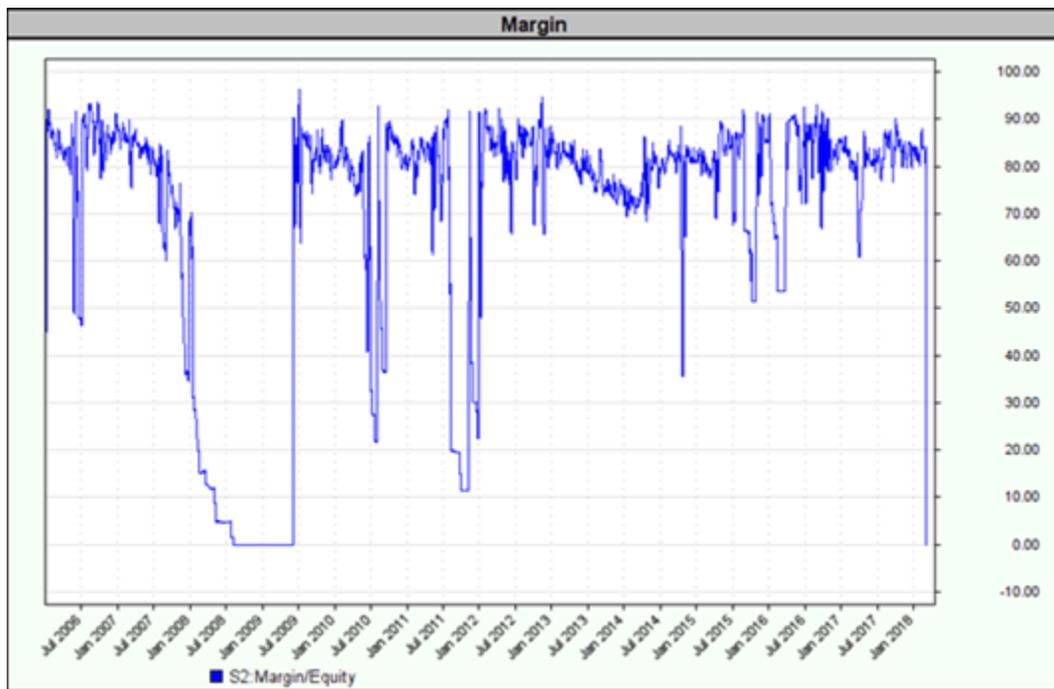
Default Value

Scope

Auto-Index -- Uses [n] as Lookback from Current Bar

BPV Series Indexing Options

BPV variables defined to create a Custom Graph image at the end of test Will appear something like this next graph image:



BPV-Variable Custom Graph - Margin/Total Equity %

The above graph is an example that display the amount of **Margin** as a percentage of the **Test's Total-Equity**. This graph is created and displayed in the Custom Graph area of the **Performance Summary report**.

Graph's created for a **Performance Summary Report** are saved in the **Results** section. In the Results section there will be a folder that matches the Performance Summary file date and time information. Contained in that folder are all the graphs displayed in the **Performance Summary report**.

Custom Graph are static images used for reporting. This needed approach for reported prevents the image from display any of the plot location values that a cursor might select. However, the data used to create a Custom Graph is easily be exported to the Print Output.csv file, or to a custom file that can be used to see the values as a text, or a spreadsheet table or plotted chart.

Script Statement 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
```

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 310

Series Data Information

Data Series, also known as an Array, can create series variables using the Block Permanent Variable (**BPV**) dialog, or the Instrument Permanent Variable (**IPV**) dialog.

Both (BPV) & (IPV) Data Series types are available to support two different data types:

- Numeric series will work with Floating Point & Integer values in all of its elements.
- String Series are design to contain any String or Text value in all of its elements.

Both (BPV) & (IPV) Data Series can be Auto-Index or Manually-Index:

- Data Series intended to provide an indicator instruments, must be Auto-Index so the alignment of the instrument record testing is aligned to an element in the instruments data series.
- Data Series intended to track how a system many positions are active at one time must be Auto-Index so the number of positions that are active can be recorded and aligned to each test record.

Both (BPV) & (IPV) Data Series can use a Manual-Index:

Manual-Index needs the user of a Manual-Index decides how many element are included in the Manual-Index series.

The process that manages the Manual-Index control must be created by the user so the information. There are two examples of how to a manual-index can be managed in this next Trading Blox Builder Forum Link:

[Manual Array Sizing Demo](#)

BPV Data Information:

All (BPV) variable data is not dependent on any instrument.

When you need to create a variable that needs to not be dependent on any instrument, a (BPV) must be where you create your variables.

Knowing the characteristic of the data type you select, makes a significant difference in how it is used and what is available from that data type. For example, if the data stored needs to be accessible with the same information in all of the test-operating script sections, a (BPV) is the data-type to use.

IPV Data Information:

All (IPV) variables are dependent on a specific instrument symbol that is in the system's list of instruments.

When you need to create a variable that needs to be dependent on a specific symbol instrument, an (IPV) must be where you create that variable.

All data elements in a Numeric series are a Floating Point variables. All data elements in a String Series are String or Text values.

Both series are also know as an array of elements that are accessed by their element position index value. Trading Blox Builder supports these two type of series: **Auto-Index** & **Manual-Index**. **Auto-Index**: is a process that sizes the number of elements in the series to be the same as number of records in the instrument file. To access an element in the series, the record number of the data's record is all that is needed to access that element in the series.

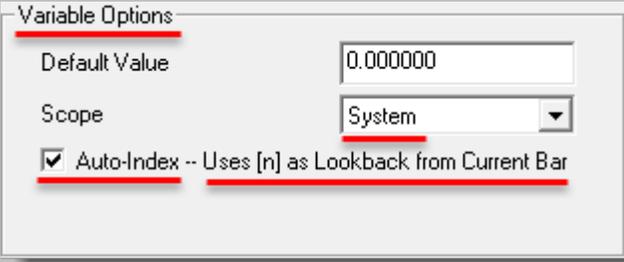
an indexed list of numeric value locations that can hold the same value, or a different a value in each of the series elements. Auto-Index Series match the instrument's date information and use the same index offset that the instrument's data use. i.e., an `instrument.close[0]`, represents the current close price. An `instrument.close[1]` represents the previous date's close price. That same offset number process is how the information can be accessed in an Auto-Index numeric series.

A numeric series can be Manually-Index. Manually-Indexed series require the user to determine the size of the series, this means how many variable places will be needed, and how to access the series requires the user to create the indexing and searching scripts to access information.

Numeric Series/Arrays, can only contain a list of numbers. These can be tied to the test day using the "Auto-Index" feature described below. They can also be a fixed manually sized series that can be expanded or reduce using scripting. Fixed series are access by an index that is managed by scripted statements.

<p>Numeric Series</p>	<p>All data elements in a Numeric series are a Floating Point variables. A series, also know as an array, are an indexed list of numeric value locations that can hold the same value, or a different a value in each of the series elements. Auto-Index Series match the instrument's date information and use the same index offset that the instrument's data use. i.e., an <code>instrument.close[0]</code>, represents the current close price. An <code>instrument.close[1]</code> represents the previous date's close price. That same offset number process is how the information can be accessed in an Auto-Index numeric series.</p> <p>A numeric series can be Manually-Index. Manually-Indexed series require the user to determine the size of the series, this means how many variable places will be needed, and how to access the series requires the user to create the indexing and searching scripts to access information.</p> <p>Numeric Series/Arrays, can only contain a list of numbers. These can be tied to the test day using the "Auto-Index" feature described below. They can also be a fixed manually sized series that can be expanded or reduce using scripting. Fixed series are access by an index that is managed by scripted statements.</p>
-----------------------	---

<p>Auto-Index Series</p>	<div data-bbox="483 237 1107 495" style="border: 1px solid gray; padding: 5px;"> <p><u>Variable Options</u></p> <p>Default Value <input type="text" value="0.000000"/></p> <p>Scope <input type="text" value="System"/></p> <p><input checked="" type="checkbox"/> <u>Auto-Index -- Uses [n] as Lookback from Current Bar</u></p> </div> <p style="text-align: center;">BPV Series Indexing Options</p> <p>A BPV Series with Auto Index checked, will track the test.currentDay index. When <u>Auto-Index is NOT Enabled</u>, the series will be a static series that is managed and accessed with scripting.</p>
<p>Plotting Controls</p>	<p>This option will not appear unless the BPV numeric-series type is selected and is using System-Scope setting.</p> <p>The numeric variable type that has a System scope setting will display this Plotting Control area where the user can create a chart graphic to display the numeric series information:</p> <div data-bbox="483 884 1260 1283" style="border: 1px solid gray; padding: 5px;"> <p><u>Plotting Controls</u></p> <p><input checked="" type="checkbox"/> Plots Enables BPV-Plotting</p> <p><input type="checkbox"/> Log Scale</p> <p>Plot Color <input style="background-color: blue; color: white; border: none;" type="text" value="Blue"/> Plotted Color Selector</p> <p>Graph Area <input type="text" value="Margin"/> Custom Graph Name</p> <p>Graph Style <input type="text" value="Thin Line"/> Type of Plot Display Selector</p> </div> <p style="text-align: center;">Appears when a BPV is System-Scope, or Test-Scope.</p> <ul style="list-style-type: none"> • The color of a graphed plot is assigned using the Plotted Color Selector. • The graph name where the plotted information is to appear is determined by the name entered into the Graph Area field. • The type of plotted display is based upon the selection appearing in the Graph Style selector. • When more than one series is being plotted in a custom graph, and one of those series has enabled "Log Scale" option, then all series plotting in the same Graph Area will plot as log scale. • When you plot more than one BPV variable in a custom graph, be sure the ranges of the all the series is within a reasonable range for all the series being plotted in that same graph area so the display is readable and consistent. • When a blox is used to create a custom graph, and that blox is inserted in multiple systems, there will be multiple plotted graphs displaying information from each of the systems. This is also true when you use multiple plotting

Auto-Index Series	 <p style="text-align: center;">BPV Series Indexing Options</p> <p>A BPV Series with Auto Index checked, will track the test.currentDay index. When <u>Auto-Index is NOT Enabled</u>, the series will be a static series that is managed and accessed with scripting.</p>
	statements in in your test, then multiple Summary Custom Charts will be created for each one.

SetSeriesSize

Data Series Indexing

Block Permanent Variables

Instrument Permanent Variables

SortSeries

SetSeriesValues

Numeric Series:

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.

String 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.

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.

Two Column Number Series:

Two-column series are single-column numeric series that are later modified with a simple scripting process to establish the second column. All two-column series must be manually indexed series. Manually indexed series require scripting to keep track of the series size and decide which index value is needed to access information.

Bar Indexing Description and Reference:

Instrument Permanent Variables

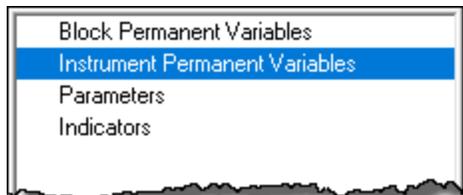
At the beginning of a test, Trading Blox Builder automatically loads the portfolio's [Instrument Data Properties \(IPV\)](#).

Each [IPV](#) data object is specific to one of the portfolio symbols loaded during the initialization of a test. During testing, instruments are processed one symbol at a time in the script sections that automatically provide instrument context (See [Accessing Instruments](#)).

This data is placed the Instrument Permanent Variables (IPV) that is made available in each of the [scripts sections that provide automatic context for instruments](#) that test instruments during a test.

Data loaded is the various properties at the start of a test can be different for each instrument. Data in a dictionary needed to support market information during a test can be the same. For instance, the point value for Corn and Soybeans can be the same. However, it won't be the same when an instrument is testing Gold.

Creating & Editing IPV Variables:

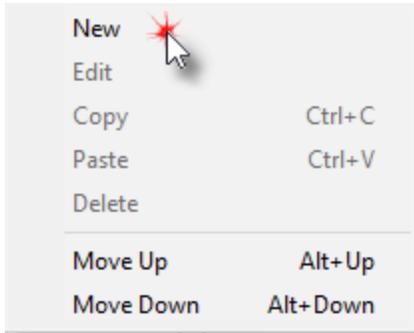


IPV Editor Menu

Double-Clicking on the Instrument Permanent Variable (IPV) menu above will open the [IPV Variable Editor Dialog](#) shown below.

A Right-Click on the same menu item, and then selecting the New or other options will display [IPV Editor](#).

IPV's Menu Options:



Block Basic Editor's BPV Menu Options

IPV Variable Editor Dialog:

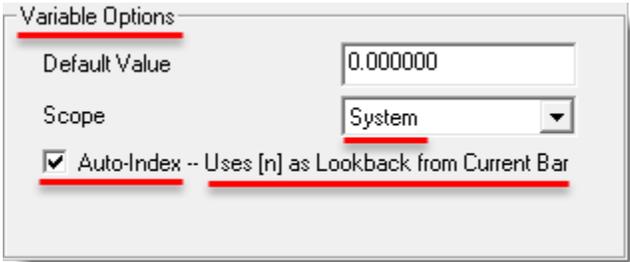
Instrument Permanent Variable (IPV) Editor

Variable Script Name & Display Name:	Descriptions:
Script Name	<p>This is the display name of the IPV Variable List section and in the scripted statements where it is used.</p> <p>Variable Name Rules: Variable names cannot have a space or any of the special characters in the name. However, an underscore character "_" are allowed.</p> <p>Use variables names that make sense to you when you create them or change them. When a variable name is changed, it will need to be change every place where it is used in script sections where it is used. Variable names that are descriptive will improve the understanding of why they are created and applied. Descriptive names also help to reduce the mystery of a statement's purpose. For example, if you want the variable to hold the current day-of-month value (dayInMonth), that name reflects the variables purpose. Better understanding of a variable's purpose is a key factor in preventing and reduce mistakes.</p> <p>Trading Blox Builder Common naming convention:</p> <ul style="list-style-type: none"> • Use a Lower-Case letter as the first character in a name for Object Property variables. • Uses an Upper-Case first letter for functions.
Display Name	<p>This field is optional, but when it is used, the text should provide a description of the variable. This field is displayed in the variable declaration section of the block, when it is Previewed or Printed using the option in the Editor's File menu.</p> <p>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.</p>
Defined Externally in Another Block	<p>This information explains: Defined Elsewhere this option.</p> <p>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.</p>

Variable Types:	Descriptions:
Integer	Any Integer numeric value (e.g. 1, -2,400, 5, etc.)
Floating Point	Any Decimal/Fractional numeric value (e.g. 2.5, 1.414, etc.)
Price	Any instrument fractional number in the symbol's range of price values

Variable Types:	Descriptions:
String	Any keyboard character. Value passed as test must be bounded by quote marks (e.g. "Hello", "Goodbye", etc.)
Numeric Series	<p>A series, also know as an array, are a list of numeric value locations that can hold the same, or a different a value in each of the series elements. Each element in a Numeric series are a Floating Point variables.</p> <p>Numeric Series/Arrays, can only contain a list of numbers. These can be aligned to an instrument's date using the "Auto-Index" feature described below. They can also be a fixed manually sized series that can be expanded or reduce using scripting. Fixed series are access by an index that is determined by scripted statements.</p>
String Series	<p>A String Series, also know as an array, are a list of String locations that can hold the same, or a different text value in each of the element locations. All data elements in a String series are a text character variables.</p> <p>String Series/Arrays, can contain a list of numbers if they assigned by another String variable, or by bounding the a numeric value with quote marks. (e.g. "54210", "100.25", etc.)</p> <p>A String series can be aligned to the instrument's date value using the "Auto-Index" feature described below. They can also be a fixed manually sized series that can be expanded or reduce using scripting. Manually size series are accessed by an index value determined by scripted statements.</p> <p>A Series of Strings can be used for many purposes such as labels or words that might be needed more than once. They can also be used as the x axis label in a custom charts.</p> <p>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.</p>
<p>Note: Variable Names and Types cannot be changed using scripts. To make a change to either, that change must be done using the BPV Variable Editor. There is more information In the About Data Variables topic and some variable use examples.</p>	

Variable Options:	Descriptions:
Default Value	The value entered into this field will be the initialized value this variable will have at the beginning of a test. Initialized values and be changed by script statements or functions.
Scope	The selected Scope determines where this variable can be accessed and changed.

Variable Options:	Descriptions:
	<p>BPV variables are allowed to use the following scope settings:</p> <ul style="list-style-type: none"> • Block -- This scope allows the variable to accessible only within the blox where it is declared. • System -- Allows this variable to accessible from within any of the blocks in the System where this block is listed. • Simulation -- This scope is the same as Test scope, except the value will not be reset when the Suite is performing a multi-step parameter test. This means the last value of this variable will be the value that appears at the beginning of the next parameter testing step. <p>The Scope determines where you can use this variable. Instrument Permanent Variables cannot be Test scope.</p>
Auto-Index Series	 <p style="text-align: center;">BPV Series Indexing Options</p> <p>An IPV Series with Auto Index checked, will track the <code>instrument.date</code> property of the current symbol be accessed. When <u>Auto-Index is NOT Enabled</u>, the series will be a static series that is managed and a accessed with scripting.</p> <p>For both Block Permanent and Instrument Permanent, if you do not select Auto Index, you must specify a size for the array. This example, <code>curentStopPrice</code>, is an Auto Index series that plots in the Price Chart graph area in Red as Small Dots.</p>
Reset Before Test	
Plotting Controls	

Adding IPV variables can extend the capabilities a system might needed during execution.

Instrument Permanent Variable [Block: EndOfWeek Test Idea | Group: _Help]

Script Name

Display Name

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
- Graph Area
- Graph Style
- Offset Plot Ahead One Bar

Variable Options

Default Value

Scope

Auto-Index -- Uses [n] as Lookback from Current Bar

IPV Variable Control Dialog

Plotting Controls

This option will not appear unless the IPV variable Type is a Numeric Series:

Plotting Controls

- Plots Display Value
- Log Scale
- Plot Color
- Graph Area
- Graph Style
- Offset Plot Ahead One Bar

IPV Series Plotting Options

Plotting Control Option:	Description:
Plots	To display the series data as plot on a symbol's chart, enable this option. When this option is enabled, it will display a plot on the chart section

Plotting Control Option:	Description:	
	determined by the Graph Area option below.	
Display Value	When data is plotted on a chart, it will show the plot information on the screen. The value of the data displayed can be seen by viewing chart section where various types of data are displayed on the right side of the chart area. For details about what is displayed, review the Trading Blox Builder Help User's Guide Help, Main Screen's topic Trade Chart .	
Plot Color	A wide range of colors for a plot display can be found in the Trading Blox Builder Help User's Guide Help, Editing Item Colors topic.	
Graph Area	The Price Chart area is the default area where most of the plotted indicators are displayed. However, when the values being plotted are below or above the displayed price range of the chart, the plot be on the chart, but it won't be visible. To make it visible, create a text-name that is entered into the Graph Area field that will represent the plotted information so that Trading Blox Builder can create a sub-chart area in the chart display that will display the plot information that needs its own plotted range.	
Graph Style		Click on the IPV Chart Plotting Styles Options image on the left to see the ten different styles of plots available for an IPV numeric series.
Offset Plot Ahead One Bar	This option will move the last plot information one test-record to the right. This option allows stops that are calculated on the day before the next trade record to be displayed in the location where the next trade record prices will be displayed when the next trade record is created.	
String Series Display Value	 <p data-bbox="578 1602 857 1625">IPV String Series Display Option</p> <p data-bbox="561 1671 1438 1738">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.</p>	

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.

Parameters

Trading Blox Builder Parameters are a very powerful features in many ways:

- Their interface is logical and their settings are easy to change.
- One or more parameters can be stepped during a test so the resulting variations can be viewed in reports and graphs.
- They are easy to create with words that identify what they control.
- They are displayed in the same order in which they are sequenced in the blocks where they exist.
- They can control how a test results are produced and which trading orders are generated.
- Their Parameters can be restricted from other blocks, or can be available to all the other blocks.

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 System.

When a selected Suite displays its selected system, Trading Blox automatically creates a user interface for your parameters. This interface gives 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.

NOTE:

- System Parameters are **READ ONLY**. They can be accessed by scripts, but scripts not change them.
- [Click to Enlarge an Image](#); [Click to Reduce an Image](#)

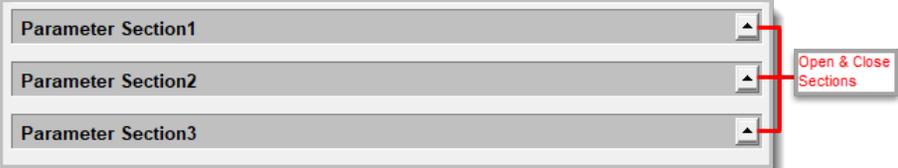
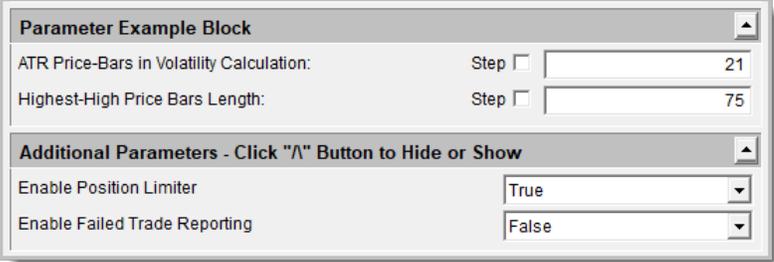
Parameter Creation & Settings Editor:

System Parameter Creation Dialog

The Parameter Type is an Integer in the case above. The supported types are:

Data Type Name:	Data Type Description:
Script Name	Script Name means the variable's name that will be used as the script parameter name.
Display Name	Display name is intended to explain the purpose of the parameter.
Integer	Whole number values e.g. 1, 400, 5, -10 - See Data-Type: Integer
Floating Point	Fractional numeric values e.g. 1.25, 2.5, 187.41415 - See Data-Type: Floating Point
Percent	Numeric percentage e.g. 1.5%, 10%. Enter these as a decimal (.50 for 50%) - See Data-Type: Percent
Boolean	Values that are either TRUE or FALSE - See Data-Type: Boolean
String	String value such as "hello" or "20081001" - See Data-Type: String
Selector	Trading Blox Builder Parameter selector enables the user to create a drop-down list of meaningful item-names that will influence how a system performs. When a list of items is created, the selector items are made automatically made available with drop-down display of the created name items. When the list is longer than the initial display, a scroll bar button is added to the list so the items not visible can be displayed.

Data Type Name:	Data Type Description:
	<p>As each user-defined name is created, Trading Blox Builder automatically creates a script-name that will be available. Each script-name created will act like a Constant, (fixed-value) script-word. Each item the user creates, will determine the number items the user will have available as testing options. Each user-name Constant will be assigned a numeric value that will be a fixed-value that begins at Zero for the first item in the list, the second item will be assigned a value of One, the third item will get a value of three, etc. until all the items in the list are numerated.</p> <p>Click on this link for detailed example of how to create and apply a Parameter Selector.</p>
<p>Section</p>	<p>Block parameters have always had the option to display their presence or to hide the parameters the user doesn't want to clutter the display. With the current release of Trading Blox Builder a block can have one or more sections that can display or hide the parameters within its section.</p> <p>Block's Parameters can now have multiple section where parameters are placed. This is useful for reducing the vertical parameter display space by hiding the parameters that aren't going to be changed during testing.</p> <div data-bbox="472 993 1122 1472" data-label="Image"> </div> <p style="text-align: center;">Creating a Parameter Section Example</p> <p>NOTE: Numeric parameter types will allow stepping within an added section. Boolean and Selector type parameters will step their values in a section, but the process is different.</p> <p>Numeric Parameters:</p> <ul style="list-style-type: none"> Parameters in a section that is open or closed are still active during a system test.

Data Type Name:	Data Type Description:
	<p>To reduce the vertical display of parameters, all the displayed parameters within a block can be hidden by clicking on the up pointing pyramid type arrow at the right most end of a section title information shaded area.</p> <p>be split into a section that shows more is available, without the space that the parameters in that section taking up the vertical space their display will consume.</p> <p>In this image, the section at the bottom of the displayed parameter block shows a section where the parameters in that section have been hidden. The process of hiding or displaying parameters is controlled by the small square button with .</p>  <p>Three Section Parameter Example with all Sections Closed.</p>  <p>Parameters With an Additional Section Group</p> <p>Clicking that button will display or hide the parameters in any its section.</p>  <p>Parameters With Displayed Section Controls</p>
<p>Use For Look-Back</p>	<p>When a Parameter with this option is enabled is assigned to an indicator calculation, the value in the parameter will determine the bar number when the indicator's calculations are available.</p> <p>When there are more than one parameters with its look back enabled and the indicator is not dependent on another indicator, and the value of this parameter's price-bar number will determine when this additional indicator's first result will be available.</p>

Data Type Name:	Data Type Description:
	<p>Check this box if the parameter is going to be used to reference past values of an indicator or past values of an instrument.</p> <p>For example, if you are using the following type of code in your script, the parameter "closeLookback" should be a lookback parameter:</p> <pre>IF instrument.close[closeLookback] > instrument.close THEN</pre> <p>If you are only using this parameter as input to an indicator, then you do not need to check the lookback box.</p> <p>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.</p> <p>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.</p>
Default Value	<p>Numeric Parameters (Integer, Floating Point & Percentage):</p> <p>The value enter in Parameter Editor is the Default Value that will appear when the blox is first connected to a system.</p> <p>Once the blox is in a system, the last value assigned to that parameter when it is displayed in the Main Screen System parameter display, will be the value that parameter will use during a test.</p> <p>value initially assigned to a parameter in the Before Test Script section. during test when it is first presented in the Parameter Editor for a system. It is also the intended value that is assigned to a numeric (Integer, Floating Point, and Percentage) parameter .</p> <p>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.</p>
Stepping Priority	<p>This option can be left at zero for most situations. Any parameter with a Priority value greater than zero will step the highest value ahead of the next higher value. enable the parameter to step before the global parameters. A value less than zero, will enable the parameter to step after of the global parameters.</p> <p>When this Parameter option has any value other than Zero, the order of when this parameter will be incremented will be changed. By default, the global</p>

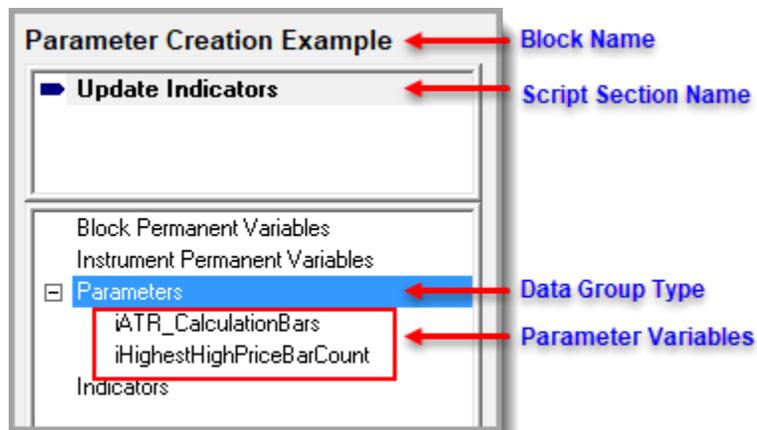
Data Type Name:	Data Type Description:
	<p>parameters have a step priority of zero. When a user created parameter has a value that isn't zero, its value won't change at the same time.</p> <p>This sets the priority of this parameter for stepping purposes. When stepping multiple parameters, the highest 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 Global parameters, use a negative step value.</p> <p>More information is available in the Parameter Stepping Priority topic.</p>
Stepping Enabled	<p>When this option is enabled, the selected parameter type will be given the ability to change the value of the parameter by using a value Parameter Stepping process.</p> <p>Review the various Parameter Types.</p>
Scope	<p>Set the Data Scope based on where a system will need a parameter's access in places other than in the block where they are created..</p> <p>If you leave the parameter's Scope setting to its default Block setting, access to that parameter's value will be limited to this block. If you set the parameter to System Scope, then all scripts in all block will have access to this parameters information. If you set the scope to Test, then all blocks in the test will have access to this parameters information.</p> <p>Click this Data Scope link for more information about Parameter Scope. On that same page is data scope information for all of Trading Blox Builder Data Types.</p>
Control Parameter	<p>This feature will hide or display a parameter section. To use its ability, a control parameter must be created so that you will have a control parameter script name to enter into the Control Parameter field name. See the Control Parameter topic for more details.</p>
Control Value	<p>In the Control Parameter field, the value must be a number. When a Binary Parameter is the Parameter that will control the display or one or more other parameters, the value you will enter into this field must a True = 1, or False = 0 to change the Child-Parameter's display.</p>

New Parameter Setting Steps:

- To create a Parameter, click the "New" button, or double-click the Parameter name heading.
- Enter the parameter setting property name in the "Name for Code" field.
- Enter the parameter setting description to display for this parameter.
- Select the Parameter data type.
- Enter a default value.
- Select the parameter value scope.
- Decide if this parameter needs to delay calculations because of look back referencing.
- If parameter requires stepping change priority, enter a step number.

Adding a New Parameter Setting:

Most of the indicators require Parameter values, but not all of them.



Parameter Creation Descriptions

This block has two parameters. Each Parameter will control the calculation length of the indicators they are assigned.

User Interface:

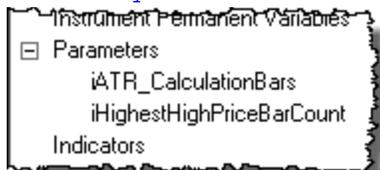
Creating Parameter variables are a simple process of filling out the fields that are important for the Parameter to use when it is in a system. The variable name of the parameter is created in the Script Name field. The display parameter description is entered in the Display Name field.

The 'Edit Parameter' dialog box contains the following fields and labels:

- Script Name:** iATR_CalculationBars (Label: Variable Script Name)
- Display Name:** ATR Price-Bars in Volatility Calculation: (Label: Parameter Displayed Description)
- Parameter Type:**
 - Integer - whole number values e.g. 1, 400, 5, etc. (Label: Variable Type Name)
 - Floating Point - fractional numeric values e.g. 1.25, 2.5, etc.
- Section - creates a section and tab
- Used for Lookback (Label: Indicator's Priming Required)
- Default Value:** 21 (Label: Calculation Default Value)
- Stepping Priority:** 0 (Label: Stepping Enabled)
- Scope:** Block (Label: Variable Scope)

Parameter Creation Details Example

This block has two parameters and both require Trading Blox Builder to automatically adjust how much of the data must be available so the results of the calculations can be provided to the system logic to perform correctly.



Parameter Variables Created for this Block.

This image shows how the parameter will appear when the system that uses this block will display the blocks information and enable the user to adjust or step the values during a test:

The 'Parameter Example Block' displays the following parameters:

- ATR Price-Bars in Volatility Calculation: Step [21]
- Highest-High Price Bars Length: Step [75]

Main Screen Parameter Display Example

Links:

[Data Names](#), [Data Scope](#), [Data Types](#), [Parameter Control](#), [Parameter Selector](#), [Parameter](#)

Links:[Stepping](#), [Parameter Stepping Priority](#) [Data Groups](#),**See Also:**[Global Parameters Properties](#), [Global Simulation Parameters](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 330

Parameter Control

Parameter Control provides us with the ability to hide and display previously hidden parameters. The parameter that controls one or more other parameters uses a value that can be as simple as True or False. It can be a Selector Parameter that changes values as different item selections are selected.

Control Parameter is a means of reducing the list of parameters that aren't necessary for your testing plans. When the Control-Parameter changes from the value that allows one or more child enabled parameters to be visible, the child parameters that are connected to the Parent Parameter will be hidden from the Main Screen's System Parameter display.

Notes:

- Text in this topic refers to the controlling parameter as a **Parent Parameter**.
- It also describes the controlled parameter as a **Child Parameter**.
- There can be **many Child Parameters** controlled by the **same Parent Parameter**.
- The **Parent Parameter** with the value that enables the **Child** to be display is only turning off the display of the parameter.
- When a **Child-Parameter is hidden**, its function or value in the system **will still be active**.

Creating Controlling Parent Parameter:

- Enter a Control Parameter Script Name the Child Parameter can use to select in its Control Parameter drop-down field.
- Enter a Control Parameter Display Information so its Control Parameter Parent Role will be identifiable.
- Select the type of Parameter to use to handle the On & Off display of a Child Parameter.
- Deselect The Stepping Enabled option so the Step All Values isn't added to the list of values.
- Set the Default Value to the Child-Parameter's Display value.
- Leave the Control Parameter Area blank.

Edit Parameter

Script Name: ← Control Parameter's Script Name

Display Name: ← System Display Name

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
- Section - creates a section and tab
- Used for Lookback

Selector Entries:

Entry	Basic Constant

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

Default Value:

Stepping Priority: Stepping Enabled

Scope:

Control Parameter:

Control Value:

Parameter Controlling Parent_1 Creation Details

Creating Controlled Child Parameter:

- o Enter a ScriptName that will be needed by a an indicator or some other process.
- o Enter a Descriptive Name that shows the reason for the parameter.
- o Select the type of Parameter that will be used by this indicator.
- o Leave the Stepping Enabled option checked if the parameter will need stepping.
- o Set the Default Value of this parameter.
- o Click on the Down-Arrow on Control Parameter Field Button on the Right and select the Script-Name of its control parameter.
- o Enter a the value of the control parameter that enables this parameter to be displayed. In this example, the value 1 for TRUE.

The screenshot shows the 'Edit Parameter' dialog box with the following fields and options:

- Script Name:** iParamChild_1 (Control Parameter's Script Name)
- Display Name:** Child-Param_1 is Displayed (System Display Name)
- Parameter Type:** Integer - whole number values e.g. 1, 400, 5, etc. (Selected)
- Default Value:** 10
- Stepping Priority:** 0
- Stepping Enabled:**
- Scope:** Block
- Control Parameter:** cParamControl_1 (Select Script Name From List Here)
- Control Value:** 1 (Enter Parameters Display Value)

Parameter Controlled Child_1 Creation Details

Notes:

- **Parameter Indent** now takes location from parent:
- Using no tab "**^n**" to override default indent.
- When using multiple tabs "**^t**" can be used for manual spacing indent adjustment.

The screenshot shows a parameter control interface with the following settings:

Parent Parameter Controls Child Parameter Example		
Param-Control_1 (True is Display – False is Hide):		True
Param-Control_2 (True is Display – False is Hide):		True
Child-Param_1 is Displayed	Step <input type="checkbox"/>	10
Child-Param_2 is Displayed	Step <input type="checkbox"/>	25
ParamChild_3 - Adjusted using Caret-n Option	Step <input type="checkbox"/>	30
ParamChild_4 - Adjusted using Caret-t Space Option	Step <input type="checkbox"/>	15

Parent Parameter Control of Child Parameters Example

Blox Parameters can be displayed in descriptive Tabs at the top of a system. The default size of a Tab is 6-characters.

Tabs are disabled by Default. To setting to enable tabs is in the "Application Section" or the TradingBlox.ini file. To enable tabs, review the "[Trading Blox Default Settings](#)" topic in the User's Guide Help File. The setting that controls the use of Tabs is:

System Edit Area Tabs=FALSE -- Tabs will be enabled by following the steps in the "[Trading Blox Default Settings](#)" topic and changing the **FALSE to TRUE**.

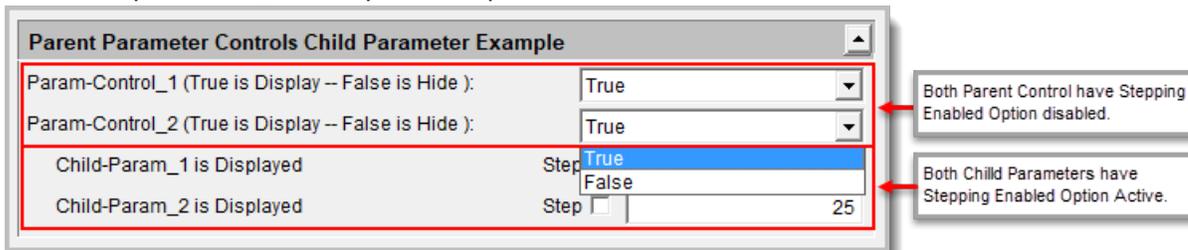
The tab size setting is in the "[Blox Basic](#)" group section where this setting is available:

Parameter Tab Size=6

Above Example Block Available Here:

[Parent-Child Parameter Control.tbx](#)

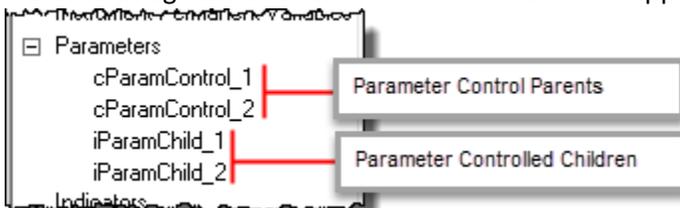
When the above two items are complete, the System Parameter display should show an Control Parameter and a Child Parameter. In this example, there are two Control-Parameters with different Script-Names so there would be more than one Control Parameter. Both control parameters are functionally the same, but they each only one Child-Parameter.



Parameter-Control Parent, Child-Parameter System Display View

In the dropped list of option, the Step All Value option is not included because the Stepped option of the Control Parameters was disabled. Both Child Parameters are displayed, but when either one of the Control-Parameters selects False, the Child-Parameter that is using the change Parent Parameter will be hidden. When it is changed back to True, the hidden Child-Parameter will be displayed again.

This next image shows how the Control Parameters appear in the Blox script display.



Parameter-Control Parent, & Child-Parameter View*

In this next image the second Control Parameter was changed to False, and the display of the second Child-Parameter is no longer visible.

Parent Parameter Controls Child Parameter Example

Param-Control_1 (True is Display -- False is Hide): True

Param-Control_2 (True is Display -- False is Hide): False → Child 2 Hidden

Child-Param_1 is Displayed Step 10

Parameter Controlling Parent_2 Hides Child_2

Note:

[Parameter Sections](#) can be added between Parameters that can be added where Control-Parameters can be placed along with seldom needed parameters can be placed. The Text Description of a Section can be hidden by clicking on the Down-Arrow button on the right-end of the text description to hide all the items in the section.

Links:
Parameters , Parameter Control , Parameter Stepping , Parameter Stepping Priority ,
See Also:

Parameter Value Updates

Directly changing a parameter's value during a test using a different value will not work. It is prevented because the built-in indicators are calculated prior to the start of testing. If the value of a parameter were changed, the values in the indicator would not be accurate because the value change is not the value used to calculate the indicator.

Custom indicators (IPV Series that are computed in the Update Indicators script) are dynamic and can use a BPV and be updated during the test. This is often used to modify channel width as an example depending on number of winning or losing trades. This method could certainly be used to update the parameter (BPV) values from one test to the next.

However, indicators are recomputed between test runs for a multi stepped test. This means that a parameter value change could be used in a multi-step test. For a multi-step test, consider changing the parameter value in the **After Test** script section so the indicators can be updated for the next step.

This idea has not been tested with test that have more than one thread. If you consider doing this change, test the idea first with the Thread setting set to 1. If a single thread worked as you expected, it might be possible to use `test.threadCount` property to select values for each new step in a multi-step test.

For example, typically the number of stepped tests is determined in advance by the number of stepping parameters. However, in this special case it might be possible to create stepping loop that could continue indefinitely until the parameters stop changing. Something to think about.

Example BPV Assignment Idea:

```

' In TB v3.x this top section shows a process that could change
' the user's rate of allocation so that it doesn't make an error.
' =====
' Parameter Value BPV Update
' BEFORE TEST SCRIPT - START
' =====
' ~~~~~
' This parameter value: AcctAdjustRate
' Is a Percentage Parameter field used in calculating
' the rate of equity allocation in a system.
' -----
' Limit Max Rate Adjustments to 100%
If AcctAdjustRate > 1 THEN AcctAdjustRate = 1

' Limit Min Rate Adjustments to 0%
If AcctAdjustRate < 0 THEN AcctAdjustRate = 0

' ~~~~~
' In TB-v5, this new section shows the same ability to control a
' user's error by substituting a BPV Float that could be changed.
' While a float variable will show the % as a decimal, it will do what we
' were doing previously. In this case, the parameter isn't used
' in scripting, but the BPV is used where the need to adjust the
' current allocation is needed.
' =====
' BEFORE TEST SCRIPT - END
' Parameter Value BPV Update
' =====

' This brief section was created to manage the same processes
' in a group of systems using the GSS process to update the
' same BPV_Variables to set the BPV-Variables in the systems.
' The Before Test Script in a multi-system using GSS Blox
' will update all the systems before testing starts.
'
' =====
' GSS Rank Adj Sizing Control
' BEFORE TEST SCRIPT - START
' =====
' ~~~~~
' GSS Module ID
iGSS_ID578 = 578
' System Index Numbers to Control
sGSS_ID578Systems = Trim(sSystemNumbers)
' -----
' Only Assign Values when System Numbers are Assigned
If Len(sGSS_ID578Systems) > 0 THEN
' GSS Rate Adjustment State
iGSS_iRankAdjSizingON = iRankAdjSizingON
' mm: H-Factor Percentile Adjust Rate:
GSS_H_Factor = H_Factor
' TSP Raw or Abs Application State
iGSS_RawScoreRankSort = iApplyABS2TSP
ENDIF ' Len(sSystemsControlled) > 0

```

Example BPV Assignment Idea:

```
' ~~~~~  
' =====  
' BEFORE TEST SCRIPT - END  
' GSS Rank Adj Sizing Control  
' =====
```

Links:

[Parameters](#), [Parameter Control](#), [Parameter Stepping](#), [Parameter Stepping Priority](#),

See Also:

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 699

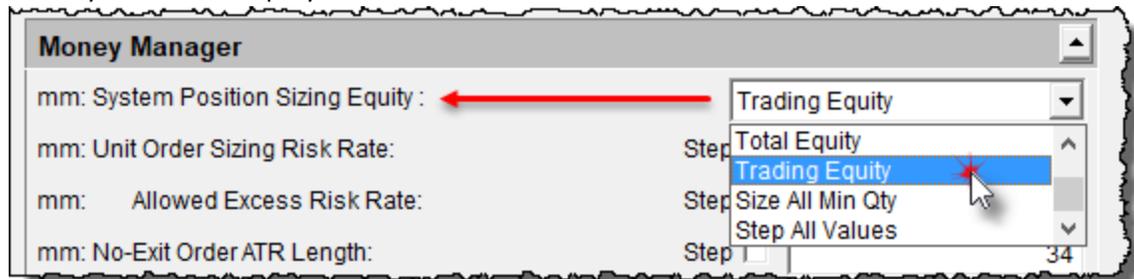
Parameter Selector

A Parameter with a Selectable list of options provides a range of options that can be easily selected and tested.

In this example, the ability to select from a range of account equity calculation methods will enable a trader to see how a system would perform with each of the Trading Blox Builder four equity calculation methods, and a single unit size of 1 Future's Contract, or Stock Share.

Parameter Selector List Display"

This next sectioned image shows how a Futures Money Manager provides the capability to test and use any of the four equity calculation methods.



Parameter Selector Items Display Example.

Parameter Selector Area Descriptions:

To create a Parameter Display of five different sizing options, the process is created by selecting the "Selector" Parameter type.

Edit Parameter

Script Name: OK

Display Name: Cancel

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**
- Section - creates a section and tab
- Used for Lookback

Default Value:

Stepping Priority: Stepping Enabled

Scope:

Control Parameter:

Control Value:

Selector Entries:

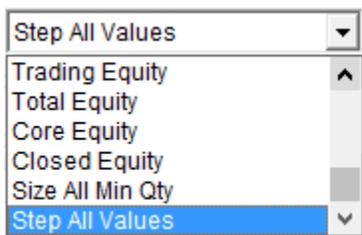
Entry	Basic Constant	Constant Value
Closed Equity	CLOSED_EQUITY	0
Core Equity	CORE_EQUITY	1
Total Equity	TOTAL_EQUITY	2
Trading Equity	TRADING_EQUITY	3
Size All Min Qty	SIZE_ALL_MIN_QTY	4

Parameter Selector List Example Details.

The Parameter Selector option selected will immediately display in the parameter setting display area.

In this example, there are five different options that all have different descriptive names. The item name order sequence doesn't matter when creating the list because there are two buttons below the list area that will allow you to change the vertical display of all the items.

As each list item text is entered, Trading Blox Builder will create a matching Constant name. List item text names can have a space-character between each word. However, constant script-names cannot have a space characters. To provide similar appearing word, Trading Blox Builder will remove the space character between the words and place an underscore character so the constant name will appear similar to the List-Item name.



User Created Selector Option Names + Stepping Option

The order of the items will determine the order in which the items are stepped when the Trading Blox Builder created selection "**Step All Values**" is the selected option. Stepping the list of items in a list required the Selector parameter shows the "[Stepping Enabled](#)" option is checked.

Creating a List of Selector Items:

Click on the **Add Entry** button and enter the an Item-Name.

Clicked Add-Item Entry Dialog

As each entry is added to the list, the other buttons display

Button Name:	Button Action:
Add Entry	Opens an entry form where a name can be entered.
Delete Entry	The high-lighted item will be removed when this button is clicked.
Move Entry Up	Any selected item that is not at the top of the list will be moved up one-row for each button click.
Move Entry Down	Any selected item that is not at the bottom of the list will be moved down on row for each button clicked.

Script Example:

Each item in a list is given a constant name.
 Each constant gets assigned a integer.
 The first item in the list has a value of Zero.
 The second item in the list has a value of one.
 The third item will have a value of three, etc.

```

23
24 ' -----
25 ' EQUITY DATA TYPES - Plots to Equity Graph
26 ' -----
27 ' Selects Equity Calculation Method
28 If Equity_Type = CLOSED_EQUITY THEN
29     aSystem_Equity = system.closedEquity
30 ELSE
31     If Equity_Type = CORE_EQUITY OR Equity_Type = SIZE_ALL_MIN_QTY THEN
32         aSystem_Equity = system.coreEquity
33     ELSE
34         If Equity_Type = TOTAL_EQUITY THEN
35             aSystem_Equity = system.totalEquity
36         ELSE
37             If Equity_Type = TRADING_EQUITY THEN
38                 aSystem_Equity = system.tradingEquity
39             ENDIF ' TRADING_EQUITY
40         ENDIF ' TOTAL_EQUITY
41     ENDIF ' CORE_EQUITY
42 ENDIF ' CLOSED_EQUITY
43
44 ' -----
45 ' SELECT RISK RATE BASED UPON ORDER DIRECTION
46 ' -----
47 ' Preserve Order Sizing Equity Rate Basis Amount

```

Parameter Selected Item Script Selection Example.

The above script is a simple example of how a nested sequence of "If THEN ELSE ENDIF" conditional statements can determine the user's selected constant-name item to influence how the system performs.

NOTE:

After the selection process above, and after the Stop and Account equity and Risk limits section completes, use the **SIZE_ALL_UNIT_QTY** value when the risk and equity will support that quantity.

Links:

[Parameters](#), [Parameter Control](#), [Parameter Stepping](#), [Parameter Stepping Priority](#),

See Also:

The values in each constant are assigned in the displayed order of the list. The constant item that is at the top of the list has a value of zero "0". The second item from the top of the list has a value of one "1". Each of the remaining items in the list will get a value of 2, 3, etc. until all the listed items in the list has a value.

Each names must be different from all the other names in the list.

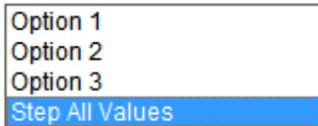
Names of the days in a week, Monday, Tuesday, etc., are Trading Blox defined Constant names. Weekday or any of the other Constant names are not allowed as an option name.

User-defined variables and indicator names are not allowed as an option name.

First option name created is assigned the value of zero. Each sequential name will be assigned the next available integer value until all options are assigned a value.

List of option name sequence is adjustable during creation and by using the Move Up or Move Down controls below the option listing area. Any change in the sequence of names will cause the values of each name to change.

In operation, the user selects one of the options from a drop-down list of option names. Selector Parameter Options can be stepped when the Step All option enabled when it is enabled.



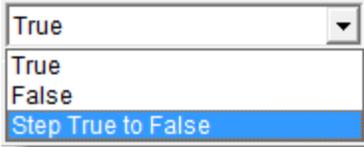
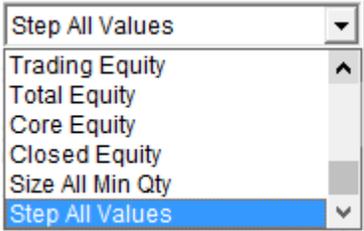
Parameter Stepping

 Stepping Enabled

Parameter Stepping Option

When a parameter shows the Stepping Enabled option is enabled, the parameter will be able to stepped during a test.

The table below shows how the different parameter types can be stepped when the Stepping Enabled Option in a parameter is enabled.

Parameter Types:		
Numeric		<p>All Numeric Parameters all use the same approach.</p>
Boolean		<p>Boolean Parameters will Step between True and False when the Stepping Enabled option shown above is enabled, and the user selects the "Step True to False" item before a test.</p>
Selector		<p>Select Parameter will Step through the entire list of items during a test when the "Step True to False" item before a test.</p>

Links:

[Links:](#)

[See Also:](#)

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 676

Parameter Stepping Priority

There are many reasons why you might want to use this option. The most common is the **Multi-Parameter Results Graph** that uses the first two parameters for the **X** and **Y** axis, and the average of the remaining stepped parameter results as the **Z** axis. By changing the priority index value from zero for one or more parameters, the testing process will step the first two parameters designated by Priority Value to be the **X & Y** axis values.

Notes:

- **Stepping Priority** values can be a positive or negative Integers
- [Click to Enlarge](#); [Click to Reduce an Image](#)
- **Global Parameters** have a **Stepping Priority** = 0

The **Donchian Entry-Exit System** is installed during the Trading Blox Builder software. It is the system source used to generate results for this Stepping Priority example.

When this system is installed, it displays each parameter default script value shown in this image.

Parameter	Step	Value
Entry Breakout (bars)	<input type="checkbox"/>	22
Entry Offset (atr)	<input type="checkbox"/>	-0.4
Stop (atr)	<input type="checkbox"/>	1.8
Exit Breakout (bars)	<input type="checkbox"/>	14
Exit Offset (atr)	<input type="checkbox"/>	0
ATR (bars)	<input type="checkbox"/>	39

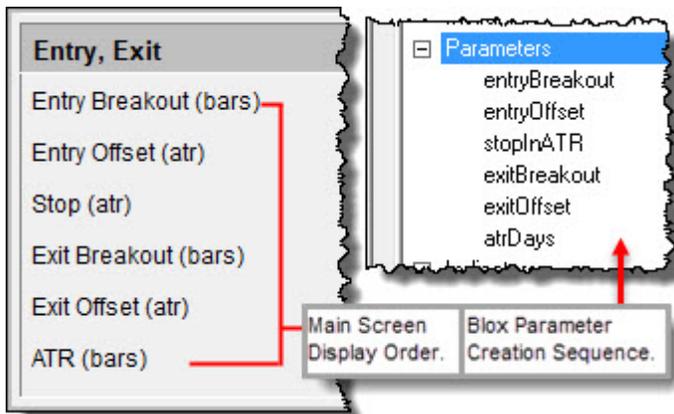
Donchian System Entry-Exit Parameters

When a parameter is created, the **Stepping Priority** option is assigned a value of zero. Changing the value of the **Stepping Priority** will only be effective during a [Parameter Stepping Test](#) is being performed.

<input checked="" type="checkbox"/> Used for Lookback
Default Value: 75
Stepping Priority: 0
<input checked="" type="checkbox"/> Stepping Enabled
Scope: Block

Parameter Stepping Priority Option Default Value

When all of the parameter Stepping Priority values are set to zero, the stepped test results will be in the order displayed in the image above.



System Parameter Display Order; Blox Creation Order

Trading Blox Builder parameter access screen follows the sequence of how the Parameters are arranged in their block. That value can be changed by the user when there is a need to create a **Multi-Parameter Results Graph** so the **X & Y** axis use the values from different parameters in the block.

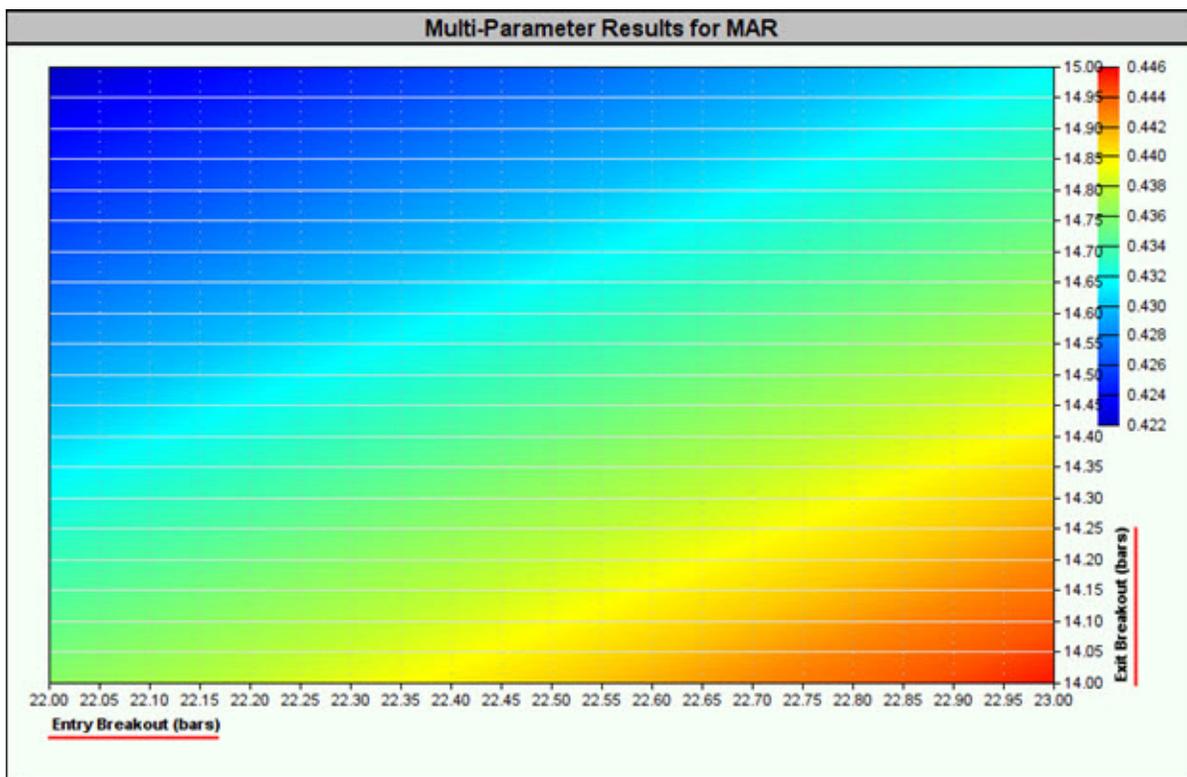
This first example of the Donchian System will step the first two parameters. Both have their Stepping Priority set to zero.

Image 1 - Click to Enlarge; Click to Reduce Image.

When the above test is run, the **Entry Breakout (days)** becomes the **X-axis** and the **Exit Breakout (days)** becomes the **Y-axis**. The **Entry Breakout (days)** is first, because it is the first parameter in the block, and its **Stepping Priority** is set to zero.

Suite: TestSuite		Results: TestSuite - 1		Results:
Summary Results		Trade Chart		
	Entry Breakout (days)	Exit Breakout (days)		
1	22	14		
2	22	15		
3	23	14		
4	23	15		

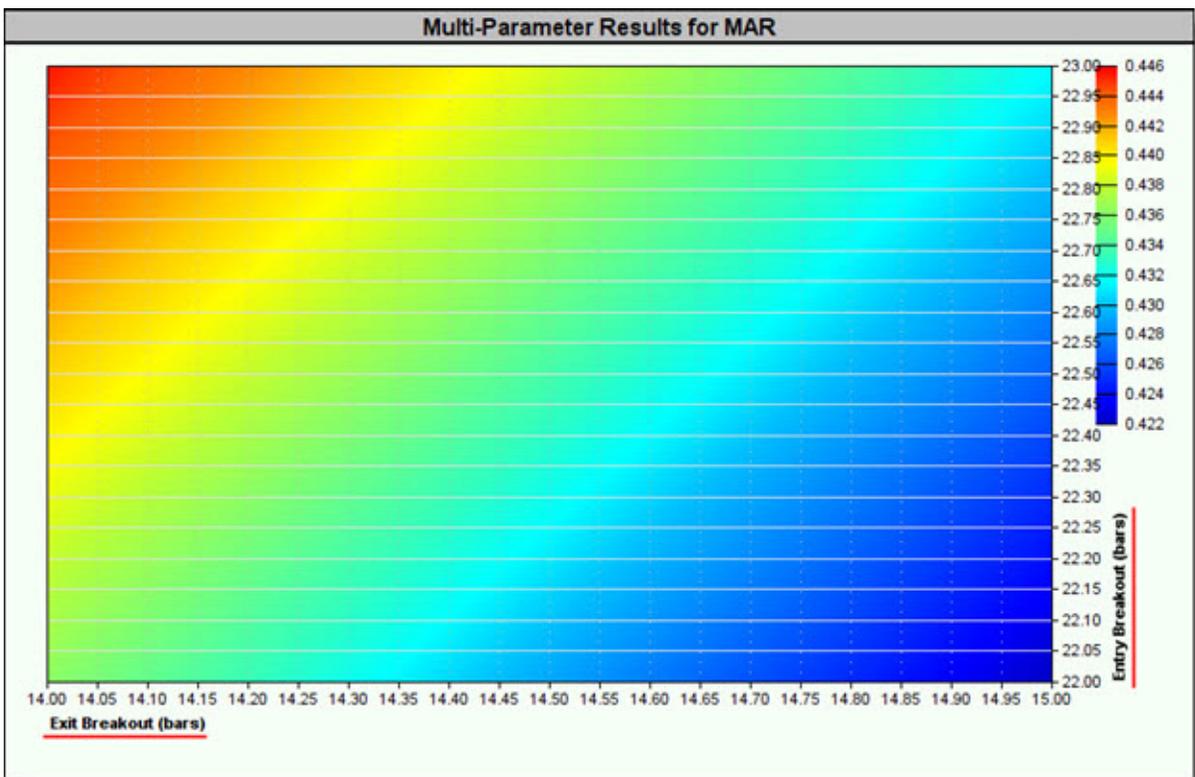
Image 2



However, when the **Exit Breakout** parameter to **1**, and the **Entry Breakout** parameter to **2**, this is the result:

Suite: TestSuite			
Results: TestSuite - 1		Results: TestSuite - 2	
Summary Results Trade Chart			
▲	Exit Breakout (days)	Entry Breakout (days)	End Ba
1	14	22	\$1,148,92
2	14	23	\$1,209,68
3	15	22	\$1,127,22
4	15	23	\$1,184,21

Image 3



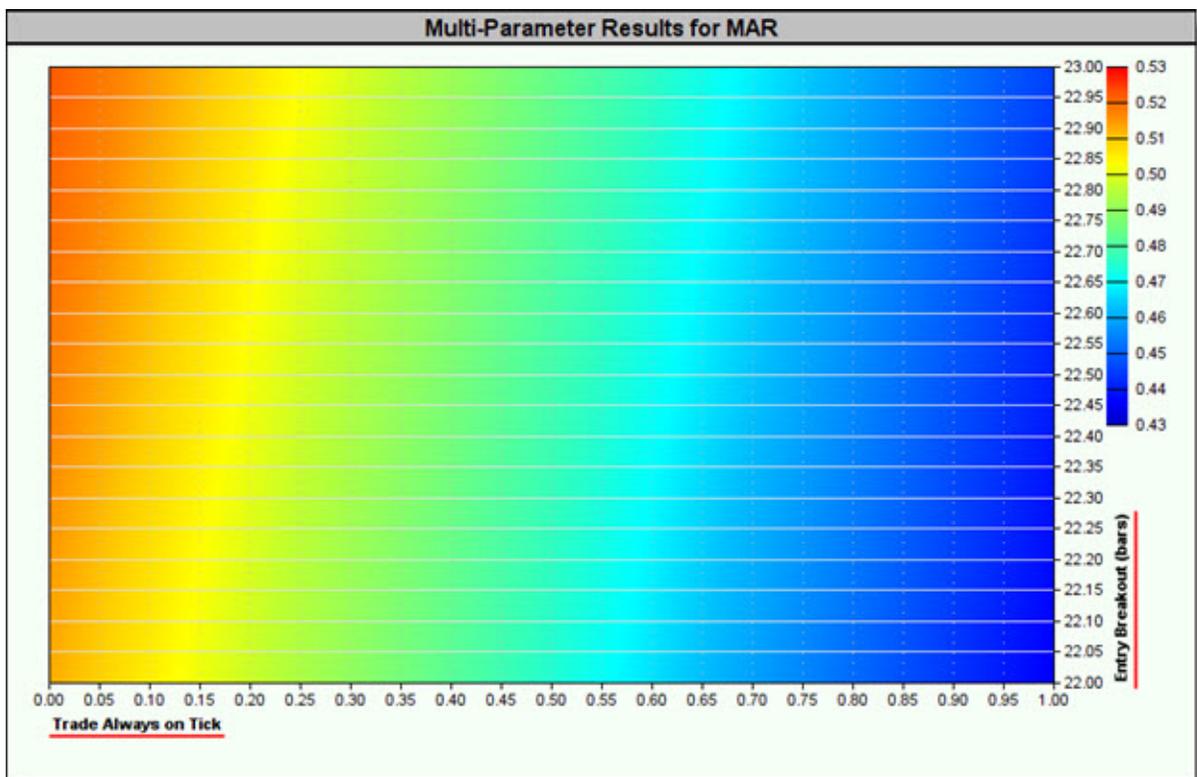
Stepping Priority = 1 for Exits & 0 for Entries

The **Exit Breakout** parameter becomes the **X-axis**, and the **Entry Breakout** **Y-axis**.

When a **Simulation Parameter** like the **Trade Always on Tick** uses its Stepping feature, and **Entry Breakout** uses a **Stepping Priority = 1**, the **Trade Always on Tick** parameter becomes the **X-axis** and the **Entry Breakout** becomes the **Y-axis**.

Suite: TestSuite			
Results: TestSuite - 1		Results: TestSuite - 2	
Summary Results Trade Chart			
#	Trade Always on Tick	Entry Breakout (days)	End Balance
1	TRUE	22	\$1,149,760.00
2	TRUE	23	\$1,204,870.00
3	FALSE	22	\$1,148,927.28
4	FALSE	23	\$1,209,680.61

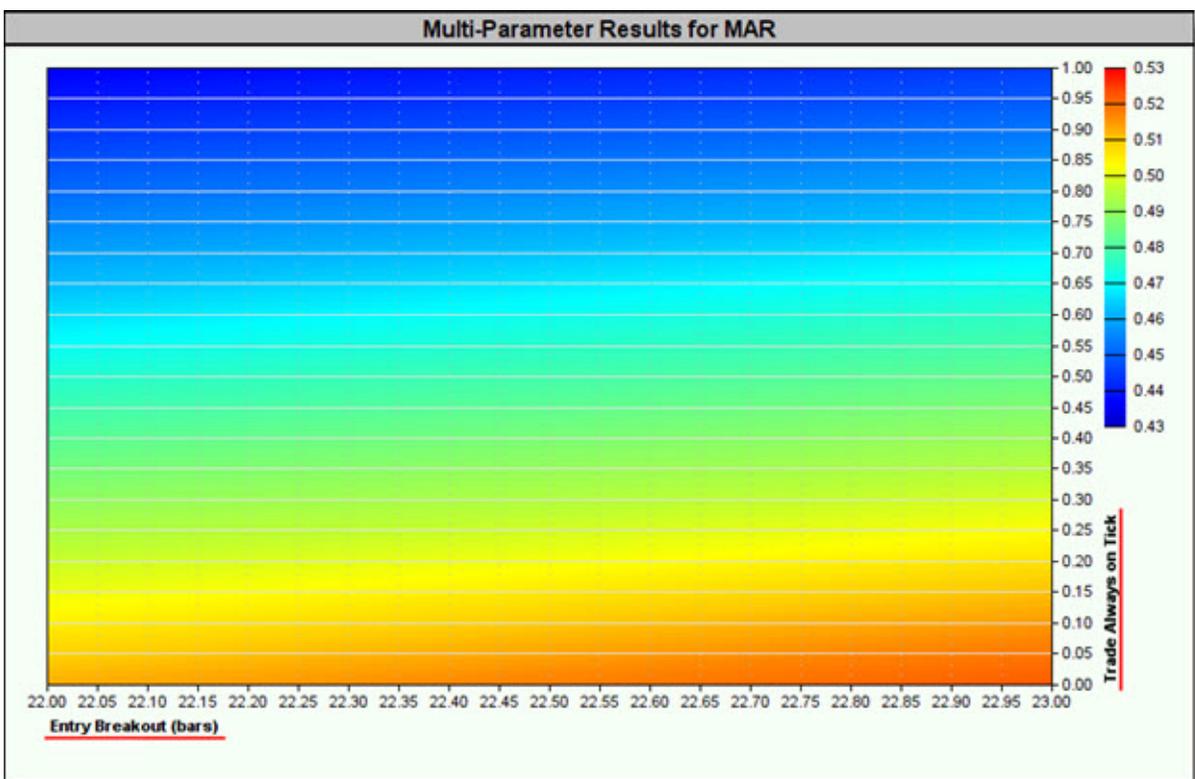
Image 4



However, when the the **Entry Breakout** parameter uses a **Stepping Priority = -1**, then it will be first because the priority of test selects the parameter with the **Lowest Stepping Priority** value.

Suite: TestSuite			
Results: TestSuite - 1			
Results: TestSuite			
Summary Results Trade Chart			
▲	Entry Breakout (days)	Trade Always on Tick	E
1	22	TRUE	\$1
2	22	FALSE	\$1
3	23	TRUE	\$1
4	23	FALSE	\$1

Image 5

**Links:**

[Parameters](#), [Parameter Control](#), [Parameter Selector](#), [Parameter Stepping](#),

See Also:

[Global Parameters Properties](#)

Indicators

Information about use and creation of indicators is available in these locations:

Types of indicators:	Description:
Basic Indicators	Use a Selected built-in formula from the Indicator Wizard Dialog.
Calculated Indicators	Take an expression. These can only use pre-test start 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	Indicator Packs are provided as an add-on list of built-in indicator methods. Indicator Pack_1 is included with Trading Blox automatically. Calculation methods contained with an indicator pack will be included in alphabetical order in the same drop down list all the previous built-in calculation methods. Indicator Pack_1 methods can be used in the same way as all other indicators.

Local Variables

Local variables are declared in a script section this way:

```
' -----  
'   Declare Local Variables in a manner similar  
'   to these three declaration, but you should  
'   consider using more meaningful names if it  
'   is important  
Variables: AnyNum1, AnyNum2, AnyNum3           TYPE: INTEGER  
Variables: AnyFloat1, AnyFloat2, AnyFloat3   TYPE: FLOATING  
Variables: AnyText1, AnyText2, AnyText3     TYPE: STRING  
Variables: pAnyPrice                          TYPE: PRICE
```

Local variables are not cleared to zero or any other value when they are declared. Clearing a local variable is the responsibility of the script section in which they are declared and used. Assigning a value to a local variable is no different than assigning a value to a Blox Permanent variable, a Instrument Permanent Variable.

For example, consider these value assignments for the first local variable in the three different local variable declaration types:

```
' -----  
'   Assign Some Value to these three variable types  
AnyNum1 = 100  
AnyFloat1 = 10.01563  
AnyText1 = "Order Placed"
```

Local variables are not accessible to other scripts unless they are contained within a Custom Function. When contained in a Custom Function, all the variables contained within the script section are accessible to script section from which the Custom Function was called.

```
' MKT is a BPV Instrument TYPE
' Local declared variables
VARIABLES: iAvgLen, iCount, iLoadOK, x, y Type: Integer
VARIABLES: iCount, instrumentCount Type: Integer
VARIABLES: fAvgClose Type: Price

' Initialize Variables
iAvgLen = 5
iCount = 0

' Get the large Stock Portfolio instrument count.
instrumentCount = system.totalInstruments

' Loop printing the symbol for each instrument.
For x = 1 TO instrumentCount STEP 1

    ' Load this portfolio instrument into context.
    iLoadOK = Mkt.LoadSymbol( x )

    ' Print out the file name.
    If iLoadOK = TRUE THEN
        ' Initialize Variables
        iCount = 0
        fAvgClose = 0

        ' Sum the Close Prices
        For y = 0 TO iAvgLen - 1
            ' Add Close Prices
            fAvgClose = fAvgClose + Mkt.close
        Next

        ' Calculate the Average Close Price
        fAvgClose = (fAvgClose / iAvgLen)

        ' Display the Symbol Information Results
        PRINT x, Mkt.symbol, fAvgClose
    ENDIF
Next ' x
```

2.6 Data 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. There are also 1485 Keywords that are in the Trading Blox Basic Language. That number usually increases a few times a year. If you find a word you entered and the parser complains, it is likely the word is reserved.

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

abs	long
Ab	loop
so	lowerCase
lut	ltrim
eV	max
al	mid
ue	middleCharacters
	min
acos	mod
and	monthNumber
ArcCosine	newPosition
ArcSine	next
ArcTangent	not
ArcTangentXY	or
asc	order
ascii	out
asciiToCharacters	pi
asFloating	print
asin	RadiansToDegrees
asInteger	random
asString	right
atan	rightCharacters

atan2	rtrim
beep	short
block	sin
broker	sine
CanAddUnit	sqr
chr	sqr
clearscreen	squareRoot
cos	step
cosine	stringLength
dayOfMonth	sub
DegreesToRadian	system
s	ta
degtorad	n
do	tangent
else	test
endfunction	then
endif	to
endsub	toJulian
endwhile	totalRisk
entryPrice	trim
exp	trimLeftSpaces
exponent	trimRightCharacters
false	trimRightSpaces
findString	trimSpaces
findString	true
for	type
function	ucase
goto	unitSize
hypot	until
hypot	upperCase
hypotenuse	variables
if	variables
index	while
instrument	xor
isFloating	
isInteger	
isString	
lcase	
left	
leftCharacters	
len	
log	
log10	

2.7 Data Scope

Trading Blox Builder variables, indicators and parameters can use various scope settings.

Being able to access data where it is needed, can help the other blocks in a system be more capable without them having to create the same processes needed to obtain the data. When a variable's Scope setting enables access outside of the block where it is created, the data in that variable can be used and changed as processing logic dictates.

Trading Blox provides data variables for different types of features. Each data type can have some of the same types of scope. Each Block variable created will have the same scope setting Block by default, unless the programmer selects a different scope access range.

Trading Blox scope settings determine where data can be restricted or available. Data can be limited to the Block where the scripting creates the information when the Scope Setting = Block. When the variable Scope Setting is Scope = System, data can be accessed by other blocks in the same system. A variable Scope Setting = Test, enables data to be accessed by any of the system Blocks in the same Testing Suite.

A Block that does not create data creates a Variable name defined in a different System Block. The "[Defined Elsewhere](#)" option will enable data from the variable name's primary Block to be accessed by the Block where the "[Defined Elsewhere](#)" option is enabled.

Accessing data using the Scope Setting process provides more Block capability during a test. The benefit is the ability to get data from other Blocks or systems in the Suite. This feature removes the burden of duplicate programming to obtain already available data in the system or testing suite.

BPV -- Block Permanent Variables:	
Block	You can only access this variable in the scripts that are in the block.
System	You can access this variable in any block in the System where the same name is created using the Defined Elsewhere option.
Test	You can access this variable in any block in the Test using the Defined Elsewhere option.
Simulation MT	This scope setting enables the value of the variable to be the same value during a multi-step test where threads will run their own test in each of the threads where a test is executing.

IPV -- Instrument Permanent Variables:	
Block	You can only access this variable in the scripts that are in the block.
System	You can access this variable in any block in the System by declaring the variable as External in the other blocks where you wish to have access to the variables information.

IPV -- Instrument Permanent Variables:**Note:**

The Simulation Scope setting is no longer available because the selection of System Scope, is the same as Simulation Scope. When another block needs to access a variable's value it must use [Defined Elsewhere](#) option in the other block's creation dialog.

Parameters:

Disabled	This setting will prevent the Parameter's value from being changed. When Disabled is selected, the displayed field will be locked. A parameter that is disabled will provide the displayed value to indicators and script sections where the parameter's value is used. This means, that all scripts and indicators will still have access to the value the parameter represents.
Block	You can only access this parameter with scripts that are contained in the same block.
System	You can access this parameter from in any block in the System
Test	You can access this parameter from any block in the Test.

Indicators:

Block	You can only access this indicator in the scripts that are in the same block.
System	You access this indicator in scripts that are in any block in the same System.

Note:

To access a System scoped Indicator in another block, define an external IPV Series of the same name in the other block.

Local Variables:

All Local-Variables are user defined variables that are limited in scope to the script section where they are defined.

This means, a **Local** variable defined using the **VARIABLES:** and **Type:** functions at the beginning and near the end of the statement and have script Item-Names, are a Local-Variable.

The variable names you use can be any name that is not a Trading Blox Builder Keyword.

```
' This Example shows how to Create Local Variables
' Each line creates a different type of variable
' Variable types must all be the same type.
VARIABLES: iValue1, iValue2      Type: Integer
VARIABLES: fChangeRate, fBaseAmt Type: Floating
VARIABLES: sSymbol, sRuleLabel   Type: String
```

Local Variables do not automatically initialize to a Zero or any other value. Before using a

Local Variables:

Local Variable, consider creating an initialization step so the process you use doesn't get unexpected results.

```
' Initialization Example:
iValue1 = 0
iValue2 = 0
```

Local-Variables are not as fast as **BPV** variables. If you have a process where there are many iterations of a local variable, a **BPV** will enable those iterations to be performed faster than a local variable.

Changing Scope:

Setting a variable to Block scope limits the information in that variable to the block were is it defined. When the scope is Not **Block** scope, changes where else the data in that variable will be available to other blox in the system. However, the other blox needs to have a variable with the same name, and if it is an **IPV** variable, it should be defined as external.

Block Permanent Variable [Block: Tutorial Entry Exit Lesson 1 | Group: Entry Exit]

Script Name: bloxBasicNameGoesHere (BPV Name (required))

Display Name: Human readable name goes here... (BPV Purpose (optional))

Defined Externally in Another Block (External Referenced Option)

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 (BPV Initialization Value)

Scope: Block (Data Scope Access Default)

Reset Before Test (Data Scope Access Options)

Stepped Simulation BPV Default Reset Option

Block Permanent Variable Creation Example (BPV)

Only **IPV** and **BPV** data types can be defined as an **External Reference** where the **Defined Elsewhere** option is used to create access to an **IPV** or a **BPV** data variable's value.

Instrument Permanent Variable Creation Example (IPV)

All variables dialog created parameters and indicators are reset between every test run during a multi-stepped parameter test, and it happens just before the [Before Test](#) script is executed. 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 is the same and it 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.

None of the script created Local-Variables are automatically initialized during a stepped test unless the script area where the Local-Variables are located have supporting script statement value assignments that will initialize the Local-Variables ahead of when they will be referenced.

Custom Function Scope:

When a script section calls a user created custom function, the [Local-Variables](#) in the calling script and in the custom function are accessible by both locations. This happens because the process of calling a user created custom function blends the calling scripts with the calling scripts environment so that both the calling script section Local Variables and the Local Variables in the custom user function code scan use information in both locations.

Multi-Threading Stepped Test:

Variables scoped **Test** can now enable or disable the resetting the **BPV** and **IPV** variables to default values before a test step in a multi-step test. Historically, the resetting to a default value of these variable was always forced to happen before the next stepped test was performed.

As you might know, in the past use of the **Simulation/Test** Scope setting would serve two purposes. It provided a scope definition that allowed broad access, and its second process was to suppress the reset before each test.

Trading Blox Builder has now separated these to concepts, so you can set the scope separately from this reset process by changing the status of the "Reset Before Test" option in the dialog where that variable is created. Keep in mind as we talk about variables and values and scopes, each thread gets it's own set of variables. This means a simulation variable is only visible in the thread were it is created for the step test.

We also now have "**Simulation MT**" scope, which is a simulation variable that is designed for multi threaded stepped testing. When this scope option is selected, it will follow the setting determined by the "**Reset Before Test**" option is enabled so same value is available to all thread-steps in the Multi-Step Test.

Reset Before Test:

By default all variables (other than **Simulation/Test**) are reset to their default before each stepped test in the Before Test script section. This allows each step of the test to start from the same value for all the data variables where the "**Reset Before Test**" option is enabled.

BPV's can be Block, System or Test scope. Each of those scope ranges can reset to default before each test or not.

The special type of Simulation MT is a test scoped variable that is accessible across all threads.

The IPV's can be Block or System scope, again resetting or not. There is no test or simulation scope for IPV's.

Each thread creates a complete copy of all the systems, blocks, variables, and indicators. So each thread runs totally independent of other threads. There is no way to share an IPV across multiple

threads.

For historical clarity, in the past the concept of “Simulation” IPV was really just a system scope that did not reset before each test. So we can still do that now, no change there.

The old and now deprecated “Simulation” BPV was defined as “Test” scope and no reset to default between tests.

The old and now deprecated “Simulation” IPV was defined as “System” scope and no reset to default between tests.

Simulation Scoped Variables:

In Trading Blox Builder versions before **5.4.1.x** used the scope option: **Simulation**, Trading Blox Builder will automatically disable the "**Reset Before Test**" option on the variables where their scope setting shows **Simulation**. All other variables that do not have the **Simulation** scoped option will have the "**Reset Before Test**" check-box checked when the block is used in versions **5.4.1.x** and later. This action is taken because the previous **Simulation** scope option is no longer available.

Previously, a Simulation Scoped variable could reset before each test, and a block scoped variable can suppress the reset before each test and keep the value from the prior test.

Defined Elsewhere

"Defined Elsewhere in Another Block" option :

Any variable "Defined Externally in Another Blox" Option

Almost all blox data variables are scoped to be entirely independent of other blocks. This variable independence is created when a variable is created using the Scope setting **Block**. Where this independence isn't true, is when a blox is created with a variable that has a Scope setting of System, Test, or Simulation. These alternate scope options enable the data to be accessible outside of the block where it is created, and a different block uses the same variable name and then enables the **"Defined Elsewhere in Another Block"** option (see above image).

The data in the variable using a Scope setting of System, Test or Simulation can be accessed by the block that has created a variable with the same name and enables **"Defined Elsewhere in Another Block"** option.

This data option item enables the **"Defined Elsewhere in Another Block"** definition option, only the type setting will be displayed in the blox where the data item is named. Blox that have this option enabled must also have another blox in the same system that provides the named data item's scope setting and any default initialization value required.

Data items that use the **"Defined Elsewhere in Another Block"** option, create a dependency that requires another blox to be in the system so the data item's information can be fully referenced for the test to operate successfully.

When the dependent blox isn't included, Trading Blox Builder will stop a test and display a message similar to this one at the bottom of the data window that appears:

"I can't run the '<Any Script Name>' script from the Trading Block '<AnyBloxName>'. The parser can't understand line 1 because variable '<AnyVariableName>' is undefined in this context."

This message is reporting the "**Defined Elsewhere in Another Block**" data item hasn't been fully declared in any of the blox listed in the system being tested.

For more information about the "**Defined Elsewhere in Another Block**" option, review the information in the [Data Scope](#) topic.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 672

2.8 Data 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</p>

Groups:	Use:
	<p>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> <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.</p>

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.

Legend:

B = BPV

I = IPV

P = Parameter

S = Script

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
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.

TYPE	Available:	Description:
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	<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.</p> <p>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.</p>
Series	B, I	<p>All series or arrays store a floating point number value or a text value in a series elements.</p> <p>Each element contains the default value it was assigned during creation, until it is cleared or updated by scripting.</p> <p>Each element in a series is accessible, by using its referenced location, or an index value when the series is not enabled for auto-index.</p> <p>In simple terms, a series is an array of elements that contain Floating numbers or String Alpha-Numeric characters.</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, the current value for the offset is zero, access to yesterday's series element is a 1 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> <p>When a series variable is created, it will have single column of elements.</p>

TYPE	Available:	Description:
		<p>A new series that disables the Auto-Index option can support 2-dimensions, or two aligned elements by using the series function:</p> <pre>' Add an additional column to this series so both ' series column elements contain 10-rows each. SetSeriesSize(multiArray, 10, 10)</pre> <p>' Data in a 2-dimension series can be changed by using the series function:</p> <pre>' Set the value located at row 5 ' and column 5, with the value 55.5 SetSeriesValue(multiArray, 55.5, 5, 5)</pre> <p>The <code>SetSeriesValue()</code> can used in a For-Next loop structure so all the elements in a column get updated. To update both columns, an embedded second loop that updates both series column elements before the next row is updated can change the values in all of the series elements.</p> <p>Access to values in a 2-dimension series can be obtained by using the</p> <pre>' Print this value value1 = GetSeriesValue(multiArray, 5, 5)</pre> <p>' To Print the contents of value1, use something like this:</p> <pre>Print "Data obtained is: ", value1</pre>
String	B, I, P	Characters, or combinations of characters like "Hello", "A", and "November Soybeans."
Local	Script	Local variables can be a String, a Floating-Point, or an Integer. Integers can be treated as binary by the script section scripts.
<p>Note:</p> <p>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 be stepped during a test simulation. Integer Parameters can be enabled 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 indicator)

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

Boolean

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.

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

Dialog Example:

Boolean Parameter Setup Example
Boolean Parameter Dialog Example

Parameter Menu Example:

Boolean 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](#)

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 178

Floating

Data-Type:	Description:
Floating	Decimal numbers like 1.24 or 3.14159 which are not whole numbers

Example:

```
' Floating TYPE example
VARIABLES: NumberIs  TYPE: Floating

' 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:

[PRINT](#)

See Also:

[Data Group and Types](#)

Instrument & BPV

Data-Types:	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.

Notes:

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.

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.

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 instrument:

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

When access to instrument information isn't automatic, a script section can load an instrument using the BPV Instrument Variable-Type with the [LoadSymbol](#) function ([see linked topic](#)).

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 canceled 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
```

Check the script section name topic for information that might not be presented here.

Links:

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

See Also:

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

Integer

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

Example:
<pre>' Integer TYPE example VARIABLES: NumberIs TYPE: Integer ' 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</pre>

Links:
PRINT
See Also:
Data Group and Types

Percent

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.

Data-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.

Dialog Example:

Edit Parameter

Name for Code: OK

Name for Humans: Cancel

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:

Add Entry Delete Entry
Move Entry Up Move Entry Down

Percent Parameter Selection Example

Parameter Menu Example:

Dialog Example:

Auxiliary

Percent Parameters Display: Step 10%

Default Value: 0.100000 → Entry Value Shows

Scope: Block

Used for Lookback Stepping Enabled

Percent Parameter Menu Example

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 = 100.000000000
```

Links:

[Constants Reference](#)

See Also:

[Data Group and Types](#)

Price

The Price data-type holds price information internally as a floating point variable.

Data-Type:	Description:
Price	<p>Price variables are printed according the current instrument's formatting and show in the debugger using that format.</p> <p>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..</p>

Example:

```
' Price TYPE example
VARIABLES: NumberIs TYPE: Price

' 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:

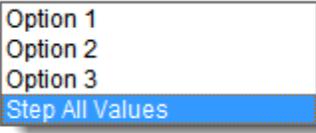
[PRINT](#)

See Also:

[Data Group and Types](#)

Selector

Selector is a variable option parameter setting. In use it provides a drop list of descriptive words that can be selected as a list for a parameter. Option selected determines the value of the parameter. When the list is created, the first option name at the top of the list is assigned the value of zero. Second option name is assigned the value of 2. Each remaining option name will be assigned the next available integer value by their display sequence until all options are assigned a value. Option name display order is adjustable during parameter creation or editing using the Move Up or Move Down button controls below the option listing area.

Data-Type:	Description:
Selector	<p>Selector is a variable option parameter setting.</p> <p>In use it provides a drop list of descriptive words that can be selected as a list for a parameter. Option selected determines the value of the parameter. When the list is created, the first option name at the top of the list is assigned the value of zero. Second option name is assigned the value of 2. Each remaining option name will be assigned the next available integer value by their display sequence until all options are assigned a value. Option name display order is adjustable during parameter creation or editing using the Move Up or Move Down button controls below the option listing area.</p> <p>In operation, the user selects one of the options from a drop-down list of option names. Selector Parameter Options can be stepped when the Step All option is enabled when it is enabled.</p>  <p>Each text option item is created by the user using the New Parameter dialog. The user created text-names for each option will be available at the parameter section location on the Main screen. Text can be any name that begins with a character, but each option name can only exist once in the Blox, but it can exist in other Blox as long as the scope is set to Blox.</p>

Selector Parameter Dialog Example:

New Parameter

Name for Code:

Name for Humans:

Parameter Type:

- Integer - whole numbers
- Floating Point - fractions
- Percent - fractions
- 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:

Scope:

Used for Lookback Stepping Enabled

Stepping Priority:

Selector Entries:

Entry	Basic Constant	
Option 1	OPTION_1	0
Option 2	OPTION_2	1
Option 3	OPTION_3	2

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

Parameter Selector Creation Setup Example

Add Selector Option Dialog:

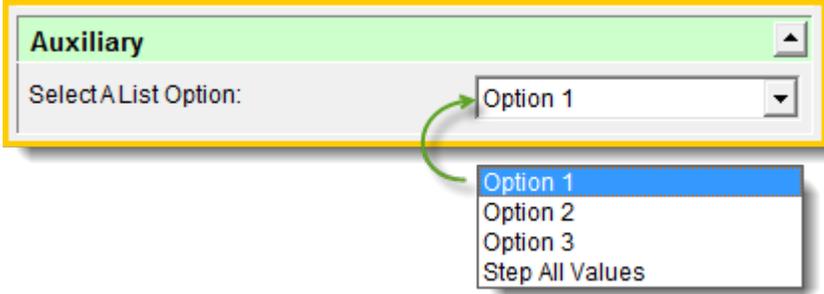
Selector Entry?

Please enter the new selector entry:

Buttons: OK, Cancel

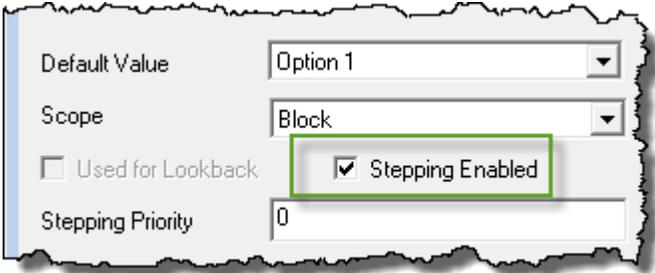
Adding a Selector Name Example

Parameter Menu Example:



Selector Parameter Selection Example

Step All Values Requirement:



Selector Parameter Stepping Option Example

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
EndIf

```

Links:

See Also:

[Data Group and Types](#)

Series

Block Permanent Variable (BPV) and **Instrument Permanent Variable (IPV)** variables support numeric and string 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	<p>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 test.currentDay property.</p> <p>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.</p>
	IPV	<p>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 instrument.bar property of each instrument.</p> <p>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.</p>
Manual-Index (see available to download modules)	BPV & IPV	<p>Manual indexing series size can be determined when the series is created, and they can be changed in the script with the SetSeriesSize series function.</p> <p>Minimum series size and index value is 1. Maximum series size and index value is 100,000. Index value is the value for the position of any element in the series including the first and largest size number of elements. Setting the index value to an element location will allow that element's value to be accessed..</p> <p>Current size of a series 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 100,000 elements.</p>

Index Method	Series Type	Description:
		<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> <p>Manual Array Sizing Demo There are a couple of modules available for download</p>
	BPV	<p>Numeric BPV series can also be declared in the script window: Variables: < series-name > TYPE: Series</p> <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	Manual indexed IPV series must be created using the IPV variable creation dialog.
Available Series Functions		GetSeriesSize SetSeriesSize SetSeriesValues SortSeries SortSeriesDual

Links:

[instrument.bar](#), [test.currentDay](#), [GetSeriesSize](#), [SetSeriesSize](#), [SetSeriesValues](#), [SortSeries](#), [SortSeriesDual](#)

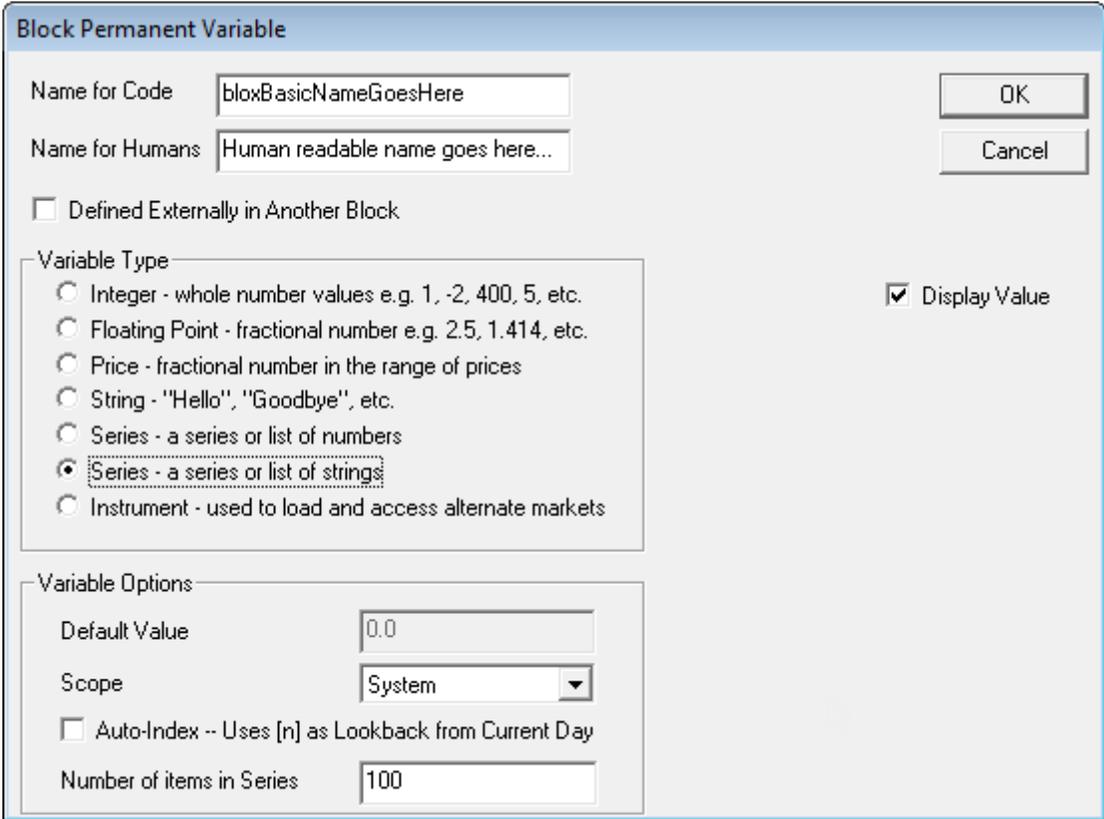
See Also:

[Data Group and Types](#), [Instrument Data Access Properties](#), [Test Object General Properties](#)

String Series

Data-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:



Block Permanent Variable

Name for Code:

Name for Humans:

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 strings
- Series - a series or list of numbers
- Instrument - used to load and access alternate markets

Display Value

Variable Options

Default Value:

Scope:

Auto-Index -- Uses [n] as Lookback from Current Day

Number of items in Series:

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](#)

Numeric Series

Data-Type:	Description:
Number 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:

Block Permanent Variable

Name for Code:

Name for Humans:

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:

Scope:

Auto-Index -- Uses [n] as Lookback from Current Day

Number of items in Series:

Plotting Controls

Plots Display Value

Log Scale

Plot Color:

Graph Area:

Graph Style:

OK Cancel

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](#)

Dual Series

Trading Blox default numeric and string series are single column series. However, Trading Blox version 4.5 and later will support a manually indexed two-column numeric series. A two-column series has two data columns that can be accessed independently.

By default, only a single column series is created when the BPV or IPV variable creation dialog is used to create a series. To add a second column the manually indexed series must be sized using scripting using two parameter size values.

Creating a series with two independent columns starts by first creating a manual single column series.

Then, changing the series is changed from is default single data column to a dual column series.

Parameter::	Description:
SortSeriesDual Usage Example	See this example in the Trading Blox Marketplace :

Setup Example:
<p>Here is how you define, set, and access multi dimensional numeric non auto indexed series:</p> <pre>' Change the myDual Series non auto indexed number array to a dual array SetSeriesSize(myDualSeries, 10, 10) ' Set the value of element 5, 1 to the number 99 SetSeriesValue(myDualSeries, 99, 5, 1) ' Print the value of element 5, 1 PRINT GetSeriesValue(myDualSeries, 5, 1)</pre>

Links:
See Also:

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.

Data-Type:	Description:
String	<p>Any keyboard character, or combinations of characters like "Hello", "A", and "November Soybeans."</p> <p>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:</p>

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](#), [String Functions](#), [EncloseInQuotes](#)

See Also:

[Data Group and Types](#)

Section 3 – Function Reference

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:
Custom Script Sections	Trading Blox uses the Script Object functions and properties to create custom calculations and custom report that are used from more than one location. Information about Custom Function is available in Script Object
Date & Time	Date & Time methods.
File & Disk	File access, copy, modification and access availability methods.
General	Various common methods.
Math	Many mathematical procedures.
Series	Table Listing of all function that work with variables declared as a series.
Strings	Many common string handling methods.
Type Conversion	Conversion methods that change a variable type from its current type to a new type are available here.

Each function section contains a table showing the names of the functions topics in that section.

3.1 Custom

Trading Blox uses the [Script](#) Object functions and properties that enable a user to create a custom script section where calculations and other processes can support a special need, like a custom reports, special calculation that can be accessed and used from more than one script section location.

Information about a Custom Function is available in [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 Trading Blox Builder 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 returned 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 do 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.SetReturnValue(
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 or String ) ' Use For a Integer, FLOAT and
String Returns
Script.SetStringReturnValue() No Longer Used See Above

' 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 `GetReference` function:

```
PRINT script.GetSeriesValue( 1, index )
```

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 245

3.2 Date & Time

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
```

Example:

```
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 be useful as a label.

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:**Description:****expression**

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:**Description:****d, m, y**

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:**Description:****expression**

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](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 260

DayOfWeek

Returns the day of the week for the date.

These are the built in [Constants References](#) to compare against:

```
' Day of the Week Number for each day of the a week
SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
  0         1         2         3         4         5         6
```

Syntax:

```
value = DayOfWeek( expression )
```

Parameter:**Description:**

expression

Any numeric integer 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
    ' When TRUE, execute the weekly update here.
ENDIF
```

Links:

[Constants References](#)

See Also:

[Date Time Functions](#)

DayOfWeekName

Returns the name of the day of the week. The name of the day returned is determined by the day-of-week number assigned to each day of the week.

These are the built in [Constants References](#) to compare against:

```
' Day of the Week Number for each day of the a week
SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
0         1         2         3         4         5         6
```

Syntax:

```
value = DayOfWeekName( expression )
```

Parameter:**Description:**

expression

Any numeric expression that resolves to an integer value between 0 and 6.

Returns:

Name of the day of the week for the given day of week number

Example:

```
' Generate a table of each property and function
' 19-bar test period set in Suite Parameters Tab
Print Block.name, _
      block.scriptName, _
      test.currentDay, _
      DayOfWeek(test.currentDate), _
      DayOfWeekName(DayOfWeek(test.currentDate))
```

Results:

```
Any_Idea_Test_04 Before Trading Day 1 1 Monday
Any_Idea_Test_04 Before Trading Day 2 2 Tuesday
Any_Idea_Test_04 Before Trading Day 3 3 Wednesday
Any_Idea_Test_04 Before Trading Day 4 4 Thursday
Any_Idea_Test_04 Before Trading Day 5 5 Friday
Any_Idea_Test_04 Before Trading Day 6 1 Monday
Any_Idea_Test_04 Before Trading Day 7 2 Tuesday
Any_Idea_Test_04 Before Trading Day 8 3 Wednesday
Any_Idea_Test_04 Before Trading Day 9 4 Thursday
Any_Idea_Test_04 Before Trading Day 10 5 Friday
Any_Idea_Test_04 Before Trading Day 11 1 Monday
Any_Idea_Test_04 Before Trading Day 12 2 Tuesday
Any_Idea_Test_04 Before Trading Day 13 3 Wednesday
Any_Idea_Test_04 Before Trading Day 14 4 Thursday
Any_Idea_Test_04 Before Trading Day 15 5 Friday
Any_Idea_Test_04 Before Trading Day 16 1 Monday
```

Example:

```
Any_Idea_Test_04 Before Trading Day 17 2 Tuesday
Any_Idea_Test_04 Before Trading Day 18 3 Wednesday
Any_Idea_Test_04 Before Trading Day 19 4 Thursday
```

Links:

[Constants References](#)

See Also:

[Date Time Functions](#)

DaysInMonth

Returns the number of calendar days in a month.

Syntax:
<code>numDays = DaysInMonth(m, y)</code>

Parameter:	Description:
<code>m, y</code>	The month number, and year number

Returns:
Number of days in the specified month.

Example:
<pre>PRINT DaysInMonth(5, 1990) ' PRINTS 31</pre>

Links:
See Also:
Date Time Functions

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 263

Hour

Returns the hour for the specified time.

Syntax:
<code>value = Hour(expression)</code>

Parameter:	Description:
<code>expression</code>	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](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 372

JulianToDate

Returns the date for a julian number. The reverse of the [DateToJulian](#) function.

Syntax:

```
date = JulianToDate( expression )
```

Parameter:

Parameter:	Description:
expression	any expression that resolves to a valid julian number

Example:

```
theDate = JulianToDate( DateToJulian( instrument.date ) + 5 )
```

Returns:

date in YYYYMMDD format

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:**Description:**

expression

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:	Description:
expression	Any expression that resolves to an integer between 1 and 12

Example:

```
PRINT MonthName( Month( test.currentDate ) )  
' Prints January for a date like 20100101
```

Returns:

Name of the month

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>	

Example:

```
SystemDate replaces CurrentDate
```

Returns:

```
SystemDate, 2013-01-25
```

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:

<none>

Example:

```
Print "SystemTime() ", SystemTime()
```

Returns:

SystemTime(), 1:28:03 PM

Links:

See Also:

[Date Time Functions](#)

TimeDiff

Returns the difference between two times, in time format.

Syntax:

```
value = TimeDiff( expression1, expression2 )
```

Parameter:	Description:
expression1, expression2	Any expression that resolves to a time HHMM format.

Example:

```
thedifference = TimeDiff( 1055, 945 )  
' Return 0110
```

Returns:

Difference between the two times, in HHMM format

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:**Description:**

expression

Any expression that resolves to a date in the format **YYYYMMDD**.

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

```

Example:

```
'   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
'   ~~~~~
```

Returns:

The week number for any date since 1900

Links:**See Also:**

[Date Time Functions](#)

Year

Returns the year for the specified date.

Syntax:

```
value = Year( expression )
```

Parameter:**Description:**

expression

Any expression that resolves to a date in the format **YYYYMMDD**

Example:

```
tradeYear = Year( 20021215 ) ' returns 2002
```

Returns:

Year for the given date

Links:**See Also:**

[Date Time Functions](#)

3.3 File & Disk

These Trading Blox Builder functions allow file manipulation at the disk level.

In the [File Manager](#) topic are additional functions that will allow you to create, access and write information to a file.

Functions:	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.
ClearLogWindow	Clear all text from main screen's Log Window area.
CloseAuxiliaryWindow	Function will close an open Auxiliary Window display.
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.
FileDate()	FileDate returns the last modified time of a file. This is different from how LoadSymbol loads from test start date, not earliest instrument data.
FileDateTime()	FileDateTime returns the last modified date and time of a file.
FileExists()	Confirms the existence of the referenced file or directory folder.
FileSize()	Returns the referenced file size.
FileTime()	FileTime returns the last modified time of the file.
GetPortfolioName()	Use this function to get the current portfolio name.
LoadBPVFromFile()	The Test Object's LoadBPVFromFile function will load a two-column file into an Auto-Index enable BPV Numeric Series variable.
LoadIPVFromFile()	Loads data from an external text files and attaches it to an IPV instrument's data.
MoveFile()	Moves the referenced file from its current location to the referenced destination location.
OpenAuxiliaryWindow	Function opens the Auxiliary Window in the main screen's area.
OpenFile()	Open the referenced file and displays it on the screen.

Functions:	Description:
OpenFileDialog	Displays the Windows Open File Dialog.
OpenLogWindow	Open the main screen's Log Window area.
RenameFile()	Function changes a current File name so that it has new file name.
SaveFileDialog()	Displays the Windows Save File Dialog.
SetAuxiliaryWindowText()	Send the Auxiliary Window Text Series to the Open Auxiliary Window.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 336

BuildDividendFiles

This Trading Blox function requires a CSI's Data Service stock subscription with dividends unencrypted to access Unfair Advantage dividend information. At the time of this text entry Dividend files were only available with a Gold Stock subscription service. They also must be unencrypted for Trading Blox to be able to access the dividend data in the CSI file so it can create local dividend files that Trading Blox can access.

Current CSI Data Subscriptions are available here: [CSI Data Service](#)

When this function is executed in the same Suite script where updated dividend files are needed, the test must be aborted after the dividend update and then run again with this new data. For some the process for updating dividend files is easier to use the [Trading Blox Preference -> Data Folders and Options](#) section "[Build Dividends](#)" option.

A batch file to execute a Suite file that updates the dividend files can also make it a two step process when the **Update Dividends** suite ends it then automatically executes the suite where the updated dividend files are needed.

Syntax:

```
BuildDividendFiles
```

Parameter:**Description:**

< none >

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
```

Returns:

Dividend Files will be updated.

Links:**See Also:**

Links:

--

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 180

ClearLogWindow

Keyword will clear all the contents of the main Trading Blox Low Window area.

Syntax:

`ClearLogWindow`

Parameter:	Description:
------------	--------------

<code><none></code>	
---------------------------	--

Example:

```
' Just enter the keyword to clear contents in the Log Window  
ClearLogWindow
```

Returns:

Clears the Log Window contents.

Links:

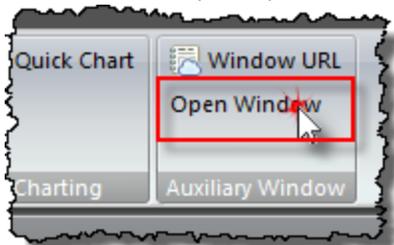
[CloseLogWindow](#), [OpenLogWindow](#)

See Also:

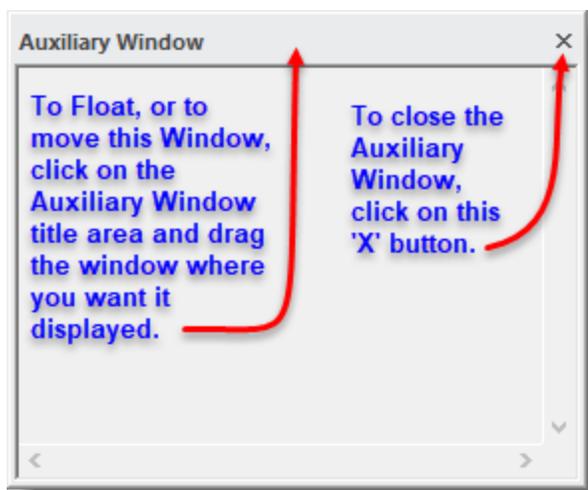
CloseAuxiliaryWindow

This function will close an open Auxiliary Window display.

An alternate way to Open, Close and Move an Auxiliary Window area is to use the mouse this way:



Auxiliary Window Open Button



Open Auxiliary Window Move and Close Options.

Syntax:

`CloseAuxiliaryWindow`

Parameter:

Description:

<none>

This command will close the **Auxiliary Window** when it is open. It will close the **Auxiliary Window** when it is **undocked**, or **docked**. When it is undocked and closed and is opened using the **Menu** option or the `OpenAuxiliaryWindow` function, the window will appear on the Main Screen in a docked condition.

Example:

```
' Auxiliary Window Will Open Now.
OpenAuxiliaryWindow
' Print Statements Will Appear in Log Window
Print "Auxiliary Window Now Open"
Print
```

Example:

```
' Auxiliary Window Will Close Now.  
CloseAuxiliaryWindow  
' Print Statements Will Appear in Log Window  
Print "Auxiliary Window Now Close"
```

Returns:

Auxiliary Window will close from the main screen's display area

Links:

[OpenAuxiliaryWindow](#), [SetAuxiliaryWindowText](#)

See Also:

[OpenLogWindow](#), [ClearLogWindow](#), [CloseLogWindow](#)

CloseLogWindow

Keyword will close an open Log Window.

Syntax:

`CloseLogWindow`

Parameter:**Description:**

<none>

Example:

```
' Just enter the keyword to close an open Log Window
CloseLogWindow
```

Returns:

Closes the open Log Window.

Links:

[ClearLogWindow](#), [CloseLogWindow](#), [OpenLogWindow](#)

See Also:

CopyFile

Function copies the Source-File name saving the duplicate file using the Copied-File-Name.

Syntax:

```
CopyFile( "C:\SourceFileName", "C:\CopiedFileName" )
```

Parameter:	Description:
"C:\SourceFileName"	File path and file name of source file.
"C:\CopiedFileName"	Destination path and copied file name.

Example:

```
' File path and name to copy and destination file path and file name.  
CopyFile( "C:\SourceFileName", "C:\CopiedFileName" )
```

Returns:

The 'Source-File-Name' is duplicated using the 'Copied-File-Name.'

Links:

[EditFile](#), [DeleteFile](#), [FileExists](#), [MoveFile](#), [OpenFile](#), [RenameFile](#)

See Also:

[File & Disk](#)

CreateDirectory

Function will create a file directory folder.

Use the [FileExists](#) function to determine if it folder name already is located where you want it created.

Syntax:
<code>iResult = CreateDirectory(directoryName)</code>

Parameter	Description:
:	
directory Name	Create a folder using the name specified.

Example:
<pre>' Create a new directory folder named 'MyOrders' If CreateDirectory(fileManager.DefaultFolder + "MyOrders\") THEN PRINT "Directory Created" ELSE PRINT "Directory already exists." ENDIF</pre>
Returns:
True if the directory was created; False if the directory creation failed.

Links:
FileExists , File Manager
See Also:

DeleteFile

Use this routine to delete a specific file.

Syntax:

```
DeleteFile( "FileToDelete" )
```

Parameter:

"FileToDelete"

Description:

Disk location and file name to delete. File information can be text, or the String variable containing the file's name and disk location.

Example:

```
' File location and name to be deleted  
DeleteFile( "c:\correlation.htm" )
```

Returns:

There is no return.

Links:

[CopyFile](#), [DeleteFile](#), [EditFile](#), [Extract](#), [FileExists](#), [FileSize](#), [MoveFile](#), [OpenFile](#), [RenameFile](#)

See Also:

[General](#), [File & Disk](#)

EditFile

Opens a file for editing.

When the file has a ".csv" suffix, it will open in Excel, or other spreadsheet program that is assigned to file with this suffix type.

When the file has a ".txt" suffix, it will open in a text-editor like Notepad, or any other text editing program that is assigned this suffix type.

Syntax:

```
EditFile( "FileToOpen" )
```

Parameter:**Description:**

"FileToOpen"

Disk location and file name to open. File information can be text, or the String variable containing the file's name and disk location.

Example:

```
' File will open in the default software assigned to files with a ".htm" suffix.  
EditFile( "c:\corr3.htm" )  
EditFile( "c:\Test.csv" ) ' Will open in Excel, or a spreadsheet as assigned in windows.  
EditFile( "c:\Test.txt" ) ' Will open in Notepad, as assigned in windows.
```

Returns:

No return.

Links:

[CopyFile](#), [DeleteFile](#), [Extract](#), [FileExists](#), [FileSize](#), [MoveFile](#), [OpenFile](#), [RenameFile](#)

See Also:

[General](#), [File & Disk](#)

Extract

Extracts all records of indicators and [IPV](#) Series variables by date time and by instrument to a file.

Note:

The Extract function will not work with a disabled series.

Syntax:
<code>Extract(file_Name, startDate, endDate)</code>

Parameter:	Description:
<code>file_Name</code>	The name and location where the list of records will be created.
<code>startDate</code>	The Start-Date value.
<code>endDate</code>	The End-Date value.

Example:
Returns:

Links:
CopyFile , DeleteFile , EditFile , Extract , FileExists , FileSize , MoveFile , OpenFile , RenameFile
See Also:
General , File & Disk

FileDate

This [FileDate](#) function returns the last modified time of a file.

This is different from how [LoadSymbol](#) loads data from test start date, not earliest instrument date.

Syntax:

```
FileDate( fullFilePathPath )
```

Parameter:	Description:
fullFilePathPath	The full drive path and file name of the file is used to file the file.

Example:

```
' Assign the Drive, Path and Full-File name.  
fullFilePathPath = "C:\Data\Stocks\AAPL.CSV"  
  
FileDate( fullFilePathPath )
```

Returns:

```
3/13/2020
```

Links:

[FileDateTime](#), [FileTime](#)

See Also:

[File & Disk](#), [Date & Time](#)

FileDateTime

This **FileDateTime** function returns the last modified date and time of a file.

This is different from how [LoadSymbol](#) loads data from test start date, not earliest instrument date.

Syntax:

```
FileDateTime( fullPathPath )
```

Parameter:

Description:

fullFilePat
hPath

The full drive path and file name of the file is used to file the file.

Example:

```
' Assign the Drive, Path and Full-File name.  
fullFilePathPath = "C:\Data\Stocks\AAPL.CSV"
```

```
FileDateTime( fullPathPath )
```

Returns:

3/13/2020 5:38 PM

Links:

[FileTime](#), [FileDate](#)

See Also:

[File & Disk](#), [Date & Time](#)

FileExists

This FileExists function will return a **TRUE** = (1) value when the file, or folder does exist, and a **FALSE** = (0) value when the file or folder does not exists.

Syntax:

```
exists = FileExists( path )
```

Parameter:**Description:**

Parameter:	Description:
path	The full path of the file or folder.

Example:

```
' Does the "test.txt" exists in the fileManager's Default Folder?
IF FileExists( fileManager.DefaultFolder + "test.txt" ) THEN
  PRINT "File Exists"
ELSE
  PRINT "File does not exist."
ENDIF
```

Returns:

Return is TRUE when the File Exists is reported, and False when "Does Not Exists" is reported.

Links:

[CopyFile](#), [DeleteFile](#), [EditFile](#), [Extract](#), [FileExists](#), [FileSize](#), [MoveFile](#), [OpenFile](#), [RenameFile](#)

See Also:

[General](#), [File & Disk](#)

FileSize

This method Returns the file's size.

Syntax:

```
iFileSize = FileSize( FullPathFileName )
```

Parameter:

Description:

FullPathFileName

String value parameter that contains the file path and file name.

Example:

```
iFileSize = FileSize( FullPathFileName )
```

Returns:

Size of File

Links:

[CopyFile](#), [DeleteFile](#), [EditFile](#), [Extract](#), [FileExists](#), [MoveFile](#), [OpenFile](#), [RenameFile](#)

See Also:

[General](#), [File & Disk](#)

FileTime

The **FileTime** function returns the last modified time of the file.

This is different from how [LoadSymbol](#) loads data from test start date, not earliest instrument date.

Syntax:

```
FileTime( fullFilePathPath )
```

Parameter:

Description:

fullFilePathPath

The full drive path and file name of the file is used to file the file.

Example:

```
' Assign the Drive, Path and Full-File name.
fullFilePathPath = "C:\Data\Stocks\AAPL.CSV"

FileTime( fullFilePathPath )
```

Returns:

5:38 PM

Links:

[FileDateTime](#), [FileDate](#)

See Also:

[File & Disk](#), [Date & Time](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 665

LoadBPVFromFile

LoadBPVFromFile loads a two column comma delimited file where the first column is a date and the second column contains a numeric value. Most often the user will create a **BPV Auto-Index enable Numeric Series**. However, this function will support a **BPV Auto-Index String Series**.

Think of the process this way:

1. All **BPV** series element values are initialized at the time of creation with the default value the user allows in the series creation.
2. The BPV will contain the values located in the second column for any `test.currentDate` locations that match a date record from the loaded file.
3. The series' default value at a `test.currentDate` location will be found when a matching date wasn't found in the loading file.

Syntax:
<code>LoadBPVFromFile(fullPathFileString, numericSeriesBPVname)</code>

Parameter:	Description:
<code>fullPathFileString</code>	<p>The drive's letter and full path to the folder that contains the named file.</p> <p>The file information needed to load a file must be in the String that contains the <code>fullPathFileString</code> parameter text. That parameter needs the text information to find and then load the two-column file. In that string variable, the full-Path-name plus the File-name must match an actual directory's path location where the file name including its suffix is located (see the below example's "Setup the path for the file to Load into a BPV" section for how it might be obtained.)</p>
<code>numericSeriesBPVname</code>	<p>BPV Auto-Index enabled numeric series.</p> <p>During the file loading process, all <code>test.currentDate</code> locations that match a loaded file record date will contain the loaded file's second column value at the <code>test.currentDate</code> date location that matched a record in the loaded file. When the loaded file does not have a date record for a <code>test.currentDate</code> value, the initialize BPV Auto-Index enable Numeric Series value won't be changed from its previous initialized value.</p>

Example:

```

=====
' LoadBPVFromFile is an Import File Example available in the Blox
MarketPlace
' https://tinyurl.com/y7slhqrl
'
' In the example space below, a second BPV manually-indexed numeric
series was created so the
' value in the second column would have a series lookup location to
display the holiday date name
' that appears when the scripting display the second column value for the
matching dates.

```

Example:

BLOCK: Load BPVFile into a Numeric Series

Block Permanent **Variables**

```
tBPV_SeriesName [ BPV Variable Series Name ]; Series String; 0.000000;
Block
aHoliday [ Holiday as a YYYYMMDD Float ]; Series; 0.000000; Block
tPathName [ Any File Path Location String ]; String; ; Block
tFileName [ Any file name String ]; String; ; Block
tFullPathFileName [ BPV Load String Variable Name ]; String; ; Block
tHolidayNames [ Human readable name goes here... ]; Series String;
0.000000; Block
iFileFound [ True File Found ]; Integer; 0; Block
```

Parameters

```
VarToShowBlox [ Param to Displays System Blox ]; String; Block
```

SCRIPT: Before **Test**

```
'
=====
' LoadBPVFromFile-Example
' BEFORE TEST - START
'
' =====
' ~~~~~
' Setup the path for the file to Load into a BPV
'
' -----
' Assign a file path location (example uses
' Example uses TB's default installation path.
' Example information file is placed in the default
' path's data folder.
tPathName = FileManager.DefaultFolder + "Data\"

' Give the file name the text name of the file.
tFileName = "US-Holiday-Series_20201225.csv"

' Append the Path, the "\" and the Filename
' to create a full File & Path name text value
tFullPathFileName = tPathName + tFileName

' Validate File's Name and Location.
iFileFound = FileExists(tFullPathFileName)
'
' -----
' Test the presence of the file. If it is found, it will load
' the file into the BPV variable 'holidays'
If iFileFound AND
    test.LoadBPVFromFile(tFullPathFileName, "aHoliday" ) THEN
ELSE
    ' When the conditional test fails, this
    ' error message will appear.
    MessageBox("Unable to Find or Load data!")
ENDIF ' test.LoadBPVFromFile...
```

Example:

```

' ~~~~~
' Holiday Name Lookup List
' Holiday File Example uses the Holiday sequence number in the
' the series name braces to select the holiday name to display
' -----
tHolidayNames[ 1] = "New Year Day"
tHolidayNames[ 2] = "Martin Luther King Jr. Day"
tHolidayNames[ 3] = "Presidents Day (Washingtons Birthday)"
tHolidayNames[ 4] = "Memorial Day"
tHolidayNames[ 5] = "Independence Day"
tHolidayNames[ 6] = "Labor Day"
tHolidayNames[ 7] = "Columbus Day"
tHolidayNames[ 8] = "Veterans Day"
tHolidayNames[ 9] = "Thanksgiving Day"
tHolidayNames[10] = "Christmas Day"
' ~~~~~
' =====
' BEFORE TEST - END
' LoadBPVFromFile-Example
' =====
-----
SCRIPT: Before Trading Day
-----
' =====
' LoadBPVFromFile-Example
' BEFORE TRADING DAY - START
' =====
' ~~~~~
' Check Current Test Date to the Loaded Holiday dates.
' When a matching Holiday date is found, the name of
' the US Holiday will appear
' -----
' Each new date it will show a holiday record
' That aligns with a holiday date
If aHoliday != 0 THEN
    PRINT test.currentDate, _
        " is Holiday#: ", _
        AsInteger(aHoliday), _
        tHolidayNames[aHoliday]
ENDIF
' ~~~~~
' =====
' BEFORE TRADING DAY - END
' LoadBPVFromFile-Example
' =====

```

Returns:

Example above returns the annual sequence count of each of the ten US-Federal Holidays show next:

```

...[SNIP]
20141111 is Holiday#: 8 Veterans Day
20141127 is Holiday#: 9 Thanksgiving Day
20141225 is Holiday#: 10 Christmas Day
20150101 is Holiday#: 1 New Year Day

```

Example:

```
20150119 is Holiday#: 2 Martin Luther King Jr. Day
20150216 is Holiday#: 3 Presidents Day (Washingtons Birthday)
20150525 is Holiday#: 4 Memorial Day
20150703 is Holiday#: 5 Independence Day
20150907 is Holiday#: 6 Labor Day
20151012 is Holiday#: 7 Columbus Day
20151111 is Holiday#: 8 Veterans Day
20151126 is Holiday#: 9 Thanksgiving Day
20151225 is Holiday#: 10 Christmas Day
20160101 is Holiday#: 1 New Year Day
20160118 is Holiday#: 2 Martin Luther King Jr. Day
20160215 is Holiday#: 3 Presidents Day (Washingtons Birthday)
20160530 is Holiday#: 4 Memorial Day
20160704 is Holiday#: 5 Independence Day
20160905 is Holiday#: 6 Labor Day
20161010 is Holiday#: 7 Columbus Day
20161111 is Holiday#: 8 Veterans Day
20161124 is Holiday#: 9 Thanksgiving Day
[SNIP]...
```

Links:[FileExists](#)**See Also:**[General](#), [File & Disk](#), [Test Object Functions](#)

LoadIPVFromFile

This instrument function loads data from an external comma separated text file into one or more user created instrument property names. The data column names that match the user's previously added IPV properties names must match for the transfer to succeed.

The IPV property types must be an **Auto-Index Numeric** or a **String** series. When the **IPV** properties are **Auto-Index Numeric** series Trading Blox Builder will match the dates in the imported text file to align the property values to those same dates in the instrument where they are being added.

Each date available in the text file is less than the instrument's record dates, the instrument record dates that didn't have a matching date in the loaded text file, will leave those instrument records with the default value that Trading Blox Builder assigns when it initializes the user created **Auto-Index Numeric** or a **String** series name.

The dates where the default value in the instrument file, the default value will be used when scripting accesses a instrument record date that wasn't updated with data from the imported text file data. A solution to that will provide the last know update to a matching date record is available in an example [here](#):

This function most often is used in the [Before Simulation](#) or [Before Test](#) script. To make this transfer work, it requires a [BPV Instrument Variable](#) to act as an instrument container. This container is needed because the instrument will need to be loaded with an instrument using the [LoadSymbol](#) function. This loading function is needed when the script sections where an instrument needs to be loaded does not support automatic [Instrument Context](#).

Once the instrument is loaded, this [LoadIPVFromFile](#) function can then load the Instrument Data file information and update the instrument properties that match the data column names.

The [LoadIPVFromFile](#) function can work in script sections where the [Instrument Context](#) is automatically provided.

Syntax:

```
' No "Date" parameter is used with this function.
' isLoaded = Returns TRUE when the function's operation
' is successful, and FALSE when it fails.
isLoaded = LoadIPVFromFile( filePathName, columnsToSkip[,myIPVSeries1]
[,myIPVSeries2][,etc.] )
```

Parameter:	Description:
filePathName	The name of the file to open. If no path name is given, it defaults to the location of the instrument data file.
columnsToSkip	Add the number of comma separated columns in the file that has the values to be added to the new IPV series properties created to store addition property data. If data is available right after the date column, enter a Zero to specify no columns should be skipped.

Parameter:	Description:
[,myIPVSeries1]	Adding More Columns is Optional: The name for the first column of data after the date and optional time.
[,myIPVSeries2]	Adding More Columns is Optional: The name for the second column of data after the date and optional time.
[,etc.]	Adding More Columns is Optional: The name of an additional column of data.

LoadIPVFromFile Information:

This function will load **date** and **time** data into an **IPV** as well. To update the date and time, the file format would then be "Date, Time, column1, column2, etc.". This different format will work correctly only when the instrument data is an intraday data series. All date time combinations in the file must also be present in the instrument data file.

For this function example, the process requires the added text files and a **IPV Numeric Series Auto-Index Properties** to be created before this function can succeed. The column heading names in the text file and in the added IPV Property names must be the same. This example's uses a **beta** and **eps** for the column property names.

This image shows how one of the **IPV Numeric Series Auto-Index Property** used the **eps** name. The other property created used the **beta** name for the text column and for the added **IPV Property**:

Instrument Permanent Variable [Block: Help-LoadIPVFromFile | Group]

Script Name:

Display Name:

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

Variable Options:

Default Value:

Scope:

Reset Before Test

Auto Index

Enabled

IPV LoadIPVFromFile IPV Requirements

Block Permanent Variables

- Mkt BPV Instrument Type container
- Instrument Permanent Variables
 - Beta Added Numeric Auto-Index
 - Eps System Scoped Series
- Parameters
- Indicators

IPV LoadIPVFromFile IPV Requirements

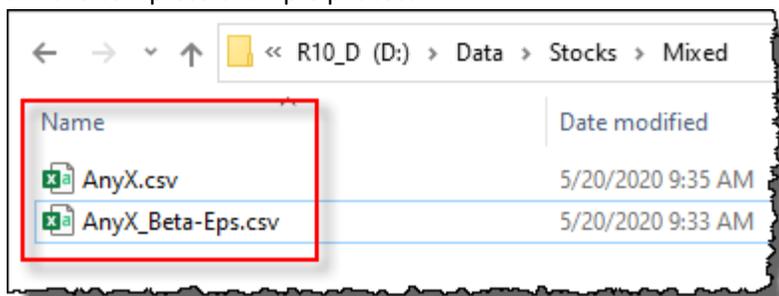
This function example used the Before Simulation script section. That script section will only load the added data information once during the entire simulation. Before using the **LoadIPVFromFile** function in the Before Simulation script section, set the new **IPV Numeric Auto-Index Series Property** variables to **System** scope. This scope setting will avoid overwriting any new property data with the IPV's default Block scope setting.

The example text file below will only provide new information every new quarter in the year. This means, dates that are not in the text file columns will be set to the default value for the series. When creating a new numeric IPV property, set the default value to "-1.000000" to indicate the property update is **False**.

If it needs to be used with a Multi-Step simulation, the Before Test script section can be used so the

added data will be refreshed before each parameter test-step execution.

To simplify the use of the example scripts below, we placed the instrument and sample data into the same disk folder location. The instrument symbol is also the first part of the added data sample. The instrument's symbol should be the lead characters in both of the files. In this example, both the symbol and added data information are fictitious symbol files created to provide a more complete example process.



IPV LoadIPVFromFile IPV Data Location Example

Once the function loads the external property data into the symbol's added numeric series, the names in the sample data that match the names of the added **IPV** properties will be updated. In this case, the names of the added properties and the column names must be the same: **beta & eps**.

To access these new IPV properties, they can be accessed in other locations:

```
IF instrument.beta > 1.2 THEN
```

OR

```
IF instrument.eps > instrument.eps[90] THEN
```

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:

```
value = instrument.beta      ' returns 1.112
value = instrument.beta[1]   ' returns default value
value = instrument.beta[2]   ' returns default value
```

Sample Property Data Example:

```
' Data is Random information created so the
' transfer process can use the following
' scripts to move the data from the sample
' into the symbol's Property data.
' -----
```

Sample Property Data Example:

```

Date,      beta,      eps
20050118,  1.201,    5.8
20050415,  1.345,    6.2
20050715,  1.112,    5.3
20051017,  1.535,    6.9
20060117,  1.231,    8.4
20060417,  1.159,    8.3
20060717,  1.188,    8.8
20061016,  1.208,    7.7
20070116,  1.434,    4.6
20070416,  1.459,    5.6
20071015,  1.484,    6.1
20080115,  1.509,    8.3

```

```

' -----
' The dates of the Sample data must match a
' date in the instrument data series for the
' data updating from the external data file
' to be successful.
'
' Added data files must be comma separated values.
' All dates in the added file must also be in the
' instrument file in the YYYYMMDD format. When a
' date in the sample file is not found in the
' instrument file, the added data will not be
' added to the instrument information.

```

Example - Step 1:**Notes:**

- The "Mkt" variable is a **BPV Instrument Type Variable** that is the container that holds the loaded instrument. The Sample data in a separate file will move its column data into the instrument's same name properties.
- [LoadSymbol](#) function loads the text file named column information into instrument symbol information using the `Mkt.LoadIPVFromFile` function.
- In this example, the default location for the Sample data and its symbol-matching instrument are both in the same folder location. This means the script path and file names will be in the same folder in the disk.
- The same full path is used to simplify the access and loading of an instrument and its new property.
- Place the scripted example statements below into the **Before Simulation** script section.
- Edit the disk and folder information so that it matches the names and path information you are going to use. In this case, there are 12 new IPV Property values to add to an instrument, and the instrument output results show there are 12 update symbol data records in the Results section of this table.

```

' =====
' LoadIPVFromFile - Help File Example - Step 1
' BEFORE SIMULATION - START
' =====

```

Example - Step 1:

```

' ~~~~~
VARIABLES: instrumentCount, _
              iSymbolExist, _
              iDataFileExist, _
              index, _
              iLoadOK, _
              instrumentCount Type: Integer

VARIABLES: DataFullPathName, _
              SymbolFullPath, _
              DataPathName, _
              SymbolName, _
              DataFileName Type: String

' ~~~~~
' Enter the Full Disk & Folder Path Sequence destination
DataPathName = "D:\Data\Stocks\Mixed"
' Enter the Symbol Series Data File to load
SymbolName = "AnyX.CSV"
' Enter the IPV Series Data File to load
DataFileName = "AnyX_Beta-Eps.csv"
' Assemble Path & File Strings for Access
SymbolFullPath = DataPathName + "\" + SymbolName
' Is Symbol File Found?
iSymbolExist = FileExists( SymbolFullPath )

' -----
' Create a Full Path & File Name String
DataFullPathName = DataPathName + "\" + DataFileName
' Is Data File Found?
iDataFileExist = FileExists( SymbolFullPath )

' -----
PRINT "iSymbolExist", iSymbolExist
PRINT "iDataFileExist", iDataFileExist

' Get the instrument count.
instrumentCount = system.totalInstruments
' Print out the file name.
PRINT "Loading External File: ", iDataFileExist

' ~~~~~
' Loop initializing each instrument.
For index = 1 TO instrumentCount STEP +1
  ' Set the portfolio instrument.
  ' "Mkt" is defined as a BPV Instrument variable.
  iLoadOK = Mkt.LoadSymbol( index )

  ' Load the external data.
  If iLoadOK = TRUE THEN
    PRINT index, Mkt.fileName, Mkt.symbol, DataFileName
    ' Load IPV Data File
    Mkt.LoadIPVFromFile(DataFullPathName,0,"beta","eps")
  ELSE
    ' Report External Instrument Load Failure

```

Example - Step 1:

```
    PRINT "Could not Load Property Data: ", DataFileName
ENDIF
Next ' index value
' ~~~~~
' =====
' BEFORE SIMULATION - END
' LoadIPVFromFile - Help File Example
' =====
```

Step 1 Returns:

```
iSymbolExist 1
iDataFileExist 1
Loading External File: 1
1 ANYX.CSV ANYX AnyX_Beta-Eps.csv
```

Example - Step 2:**Notes:**

- This scripting is placed into the **BEFORE INSTRUMENT DAY** script section.
- Its only purpose is to verify and audit the transfer from the new property data file to the instrument's numeric series properties of the same name.
- It is also provides a day-of-week name for the data date aligned with the new property data.
- Output only shows the dates where the data from the new property values file to the instrument data records is a match.
- If any data didn't get transferred, it is likely there is a date that isn't a market day in one of the files. Fix the missing data problem by validating the instrument date and the added property data dates.

Example - Step 2:

```

' =====
' LoadIPVFromFile - Help File Example
' BEFORE INSTRUMENT DAY - START
' =====
' ~~~~~
' Local-Variables Definitions
VARIABLES: iNdx, iHeader Type: Integer
' -----
' If the Output Column Header has not been displayed,...
If iHeader = FALSE THEN
'   Generate a Header to the columns
PRINT "Symbol,Date, Close, Beta, EPS, DayName"
'   Update iHeader State
iHeader = TRUE
ENDIF ' iHeader = FALSE
' -----
' When an Updated Property is found,...
If instrument.beta <> 0 OR instrument.eps <> 0 THEN
'   Report the update records updated
PRINT instrument.symbol, _
        instrument.date, _
        instrument.close, _
        instrument.beta, _
        instrument.eps, _
        DayOfWeekName( DayOfWeek(instrument.date) )
ENDIF ' i.beta != 0 OR i.eps != 0
' ~~~~~
' =====
' LoadIPVFromFile - Help File Example
' BEFORE INSTRUMENT DAY - END
' =====

```

Step 2 Returns:

```

Symbol,Date, Close, Beta, EPS, DayName
ANYX 20050118 39.951000000 1.201000000 5.800000000 Tuesday
ANYX 20050415 39.734333330 1.345000000 6.200000000 Friday
ANYX 20050715 40.971000000 1.112000000 5.300000000 Friday
ANYX 20051017 44.011000000 1.535000000 6.900000000 Monday
ANYX 20060117 50.321000000 1.231000000 8.400000000 Tuesday
ANYX 20060417 54.421000000 1.159000000 8.300000000 Monday
ANYX 20060717 50.721000000 1.188000000 8.800000000 Monday
ANYX 20061016 58.231000000 1.208000000 7.700000000 Monday
ANYX 20070116 63.464000000 1.434000000 4.600000000 Tuesday
ANYX 20070416 69.944000000 1.459000000 5.600000000 Monday
ANYX 20071015 74.714000000 1.484000000 6.100000000 Monday
ANYX 20080115 66.334000000 1.509000000 8.300000000 Tuesday

```

Missing Property Data:

In the past, Trading Blox Builder provided the option to get the last known value when there

Missing Property Data:

hadn't been an update to the property for all instrument dates. This isn't an option any longer, but it can be obtained by a simple conditional statement that executes in the [UPDATE INDICATORS](#) Script Section.

```

' =====
' LoadIPVFromFile - Help File Example
' UPDATE INDICATORS - START
' =====
' ~~~~~
' When the IPV auto indexed series loaded from a text file that
' didn't update each instrument record, and the need is to know
' most recent property value, this script will carry that value
' in the new user created IPV static values.
'
' NOTE:
'   -1 Is recommended Default value To use when LoadIPVFromFile
'   Data might skip LoadIPVFromFile Data might leave the default
'   instrument record value, use this process to carry the
'   value forward.
' -----
'   If the value Is Not -1,...
' If instrument.eps <> -1 Or instrument.beta <> -1 Then
'   Update static IPV with an Updated value
'   currentEPS = instrument.eps
'   currentBETA = instrument.beta
' ENDIF
' ~~~~~
' =====
' LoadIPVFromFile - Help File Example
' BEFORE INSTRUMENT DAY - END
' =====

```

Links:

[LoadExternalData](#), [LoadBPVFromFile](#)

See Also:

[Block Permanent Variables](#), [Instrument Permanent Variables](#)

MoveFile

This function moves a file from its current location to the location specified in the second parameter.

Syntax:

```
MoveFile( CurrentFilePathFileName, NewFilePathFileName )
```

Parameter:	Description:
CurrentFilePathFileName	The file's current address.
NewFilePathFileName	Destination address for the file.

Example:

```
CopyFile( "c:\correlation.htm", "c:\corr2.htm" )  
DeleteFile( "c:\correlation.htm" )  
MoveFile( "c:\corr2.htm", "c:\corr3.htm" )  
OpenFile( "c:\corr3.htm" )
```

Returns:

File relocated to its new address.

Links:

[CopyFile](#), [DeleteFile](#), [EditFile](#), [Extract](#), [FileExists](#), [FileSize](#), [OpenFile](#)

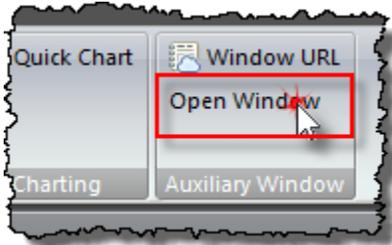
See Also:

[General](#), [File & Disk](#)

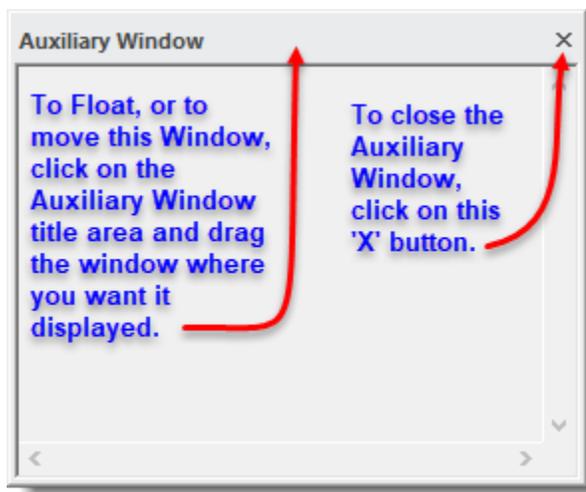
OpenAuxiliaryWindow

Function opens the Auxiliary Window in the main screen's area.

An alternate way to Open, Close and Move an Auxiliary Window area is to use the mouse this way:



Auxiliary Window Open Button



Open Auxiliary Window Move and Close Options.

Syntax:

`OpenAuxiliaryWindow`

Parameter:	Description:
<code><none></code>	<p>An Auxiliary Window is an window that will appear Docked to the main screen when it is closed. When it is undocked and visible, it will stay undocked and the information sent to it will updated. If the Auxiliary Window is closed, and the <code>OpenAuxiliaryWindow</code> function is not executed prior to the sending text to the Auxiliary Window, the <code>SetAuxiliaryWindowText</code> function execution to send data to the Auxiliary Window will not appear.</p> <p>This command must be executed prior to the sending of text by <code>SetAuxiliaryWindowText</code> function.</p>

Example:

```
' Auxiliary Window Will Open Now.  
OpenAuxiliaryWindow  
' Print Statements Will Appear in Log Window  
Print "Auxiliary Window Now Open"  
Print  
  
' Auxiliary Window Will Close Now.  
CloseAuxiliaryWindow  
' Print Statements Will Appear in Log Window  
Print "Auxiliary Window Now Close"
```

Returns:

Auxiliary Window Appears on the main screen area.

Links:

[CloseAuxiliaryWindow](#) [SetAuxiliaryWindowText](#)

See Also:

[OpenLogWindow](#), [ClearLogWindow](#), [CloseLogWindow](#)

OpenFile

Opens a file. Function requires the drive and folder names that provide the location of the file.

Syntax:

```
OpenFile( fileName )
```

Parameter:	Description:
fileName	File to open, copy, delete, or move

Example:

```
OpenFile( "c:\corr3.htm" )
```

Returns:

Displays the specified file using the Window's assigned software.

Links:

[CopyFile](#), [DeleteFile](#), [EditFile](#), [Extract](#), [FileExists](#), [FileSize](#), [MoveFile](#)

See Also:

[General](#), [File & Disk](#)

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 ] )
```

Parameter:	Description:
[filter extension]	Optional: This is for filter extensions, like .exe, .txt or .csv
[file Name]	Optional: Use this field for the file name to find.
[start path]	Optional: Use the path field to specify where the dialog will open a disk location.

Example:

```
' 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"
PRINT OpenFileDialog( "Text Files|*.txt;*.csv", "orders.txt",
fileManager.DefaultFolder )
PRINT SaveFileDialog( "Text Files|*.txt;*.csv", "orders.txt",
fileManager.DefaultFolder )
```

Returns:

When this function is working, a Windows file dialog will appear.

Links:

[SaveFileDialog](#)

See Also:

OpenLogWindow

Function opens the main Trading Blox Log Window area near the bottom edge of the program.

Syntax:

`OpenLogWindow`

Parameter:**Description:**

`<none>`

Example:

```
' Just enter the keyword to open a closed Log Window  
OpenLogWindow
```

Returns:

Log Window will open then the keyword is executed.

Links:

[ClearLogWindow](#), [CloseLogWindow](#)

See Also:

[OpenAuxiliaryWindow](#), [CloseAuxiliaryWindow](#)

RenameFile

Function Renames a current File name so that it has new file name.

Syntax:

```
RenameFile(oldname, newname)
```

Parameter:	Description:
oldname	Current File name of existing file.
newname	New File name.

Example:

```
' Current File name to copy and destination file path and file name.  
RenameFile( "SourceFileName", "NewFileName" )
```

Returns:

The saved File-Name' is the same file with a new name.

Links:

[EditFile](#), [CopyFile](#), [DeleteFile](#), [FileExists](#), [MoveFile](#), [OpenFile](#)

See Also:

[File & Disk](#)

SaveFileDialog

`SaveFileDialog` returns the string of the path the user has selected using the save file dialog.

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"
```

Syntax:

```
completeFilePath = SaveFileDialog( , , )
```

Parameter:	Description:
<code>[filterExtension]</code>	Optional: This is for filter extensions, like <code>.exe</code> , <code>.txt</code> or <code>.csv</code>
<code>[fileName]</code>	Optional: Use this field for the file name to find.
<code>[startPath]</code>	Optional: Use the path field to specify where the dialog will open a disk location.

Example:

```
PRINT OpenFileDialog( "Text Files|*.txt;*.csv", _
    "orders.txt", fileManager.DefaultFolder )
PRINT SaveFileDialog( "Text Files|*.txt;*.csv", _
    "orders.txt", fileManager.DefaultFolder )
```

Returns:

A string containing the complete path the user selected

Links:

[OpenFileDialog](#)

See Also:

3.4 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	List of User's Preference settings keyword properties.
ProductVersion	Returns the product version such as 3.4
ProductVersionNumerical	Returns the numerical such as 03040000
SetRegistryKey	Sets a value into the registry
Type	Variables can be restricted to a specific type. In Trading Blox Builder, there are three main groups of variable types. Within each group of types the some of the variables that are available can be the same, or different. Each of the group of variable types change how they can be accessed, or restricted. Click on this link to get an understanding how details are different.
Variables	In Blox Basic you can create variables which have names and can hold values just like a spreadsheet cell can. For example, a spreadsheet displays rows and columns of cells. A cell on a spreadsheet can hold the total sales for the month. That value is likely to be one of many other values. An example would be a list of months where the current value of each month is displayed. When all the monthly sale values are added together, they will represent the current status of all the known "monthlySales".

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 349

ColorRGB

Function returns a numerical color value.

Use this function in place a variable to the left so the function can assign the values it creates using the equal sign. Function requires values in all three of its parameter fields. Any and all values can be zero. All values must be in the range of zero to 255.

More Color Information is available here: [Color Selection Dialog](#)

Syntax:

```
' Create a color based upon the values of the three parameters
ColorValue = ColorRGB( BlueValue, GreenValue, RedValue
```

Parameter:**Description:**

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.

Example:

```
' Assign ColorRGB value to another variable
' Generate Red Color Number
ColorValue1 = ColorRGB( 255, 0, 0 )
' Generate Green Color Number
ColorValue2 = ColorRGB( 0, 255, 0 )
' Generate Blue Color Number
ColorValue3 = ColorRGB( 0, 0, 255 )

OR

' Assign ColorRGB values to chart object function
' Use Red Color value for data series Line1
chart.AddLineSeries( AsSeries( Line1 ), 100, "Line1", ColorRGB( 255, 0,
0 ) )
' Use Green Color value for data series Line1
chart.AddLineSeries( AsSeries( Line2 ), 100, "Line2", ColorRGB( 0, 255,
0 ) )
' Use Blue Color value for data series Line1
chart.AddLineSeries( AsSeries( Line3 ), 100, "Line3", ColorRGB( 0, 0,
255 ) )

OR
```

Example:

```
' Assign ColorRGB values to SerSeriesColorStyle function
' Use Red 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 Blue Color value for PlotLine3 series Line1
SetSeriesColorStyle( PlotLine3, ColorRGB( 0, 0, 255 ) )

OR

' Consider a Random number color assignment
'
'           Red           GREEN           BLUE
plotColor = ColorRGB( Random(255), Random(255), Random(255) )
```

Returns:

Value returned is the **RGB** value that results from the combination of the three color values assigned.

Links:

[AddLineSeries](#), [Colors](#), [Color Selection Dialog](#), [SetSeriesColorStyle](#)

See Also:

Edit Time: 3/21/2024 10:41:38 AM

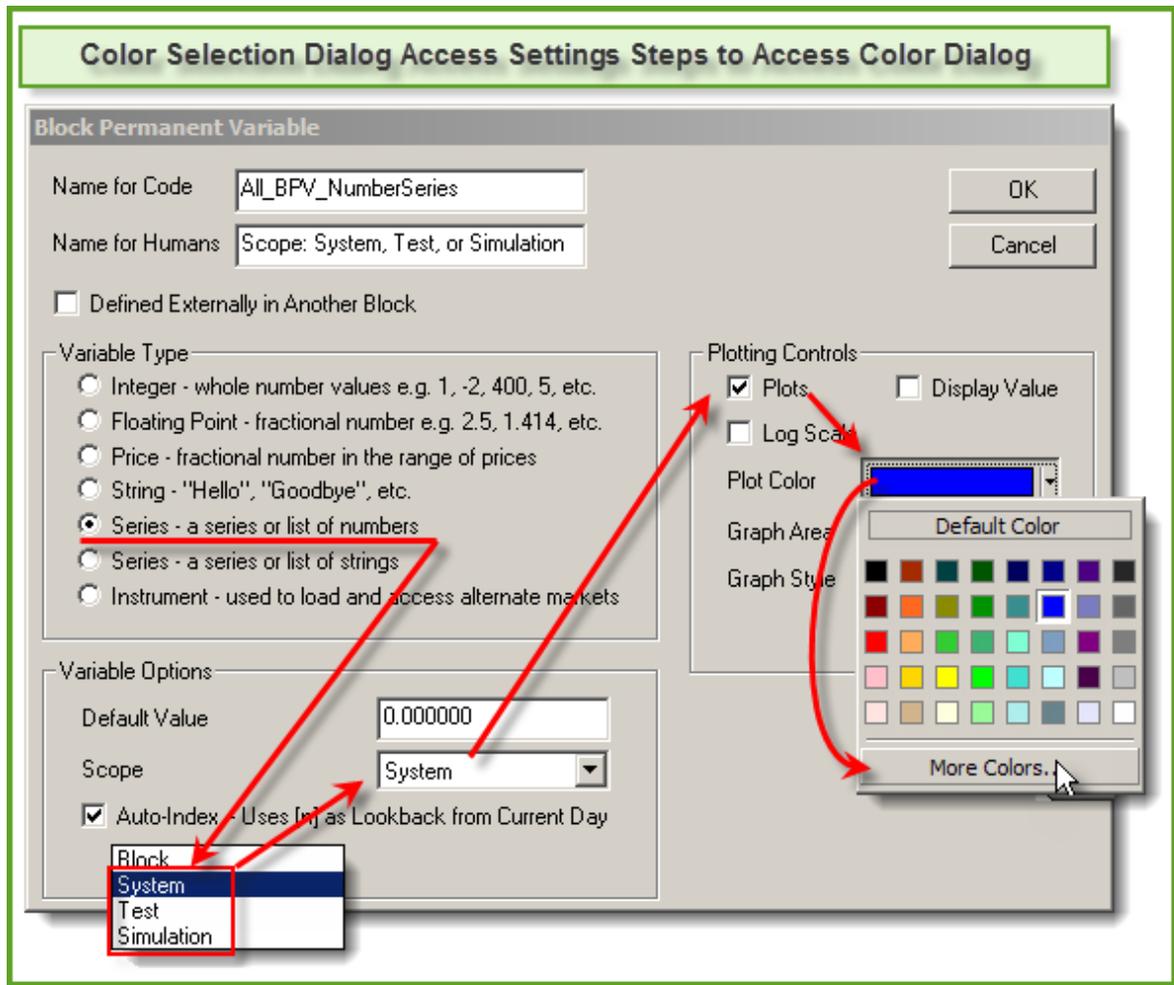
Topic ID#: 202

Color Selection Dialog

Trading Blox Color Selection Dialog:

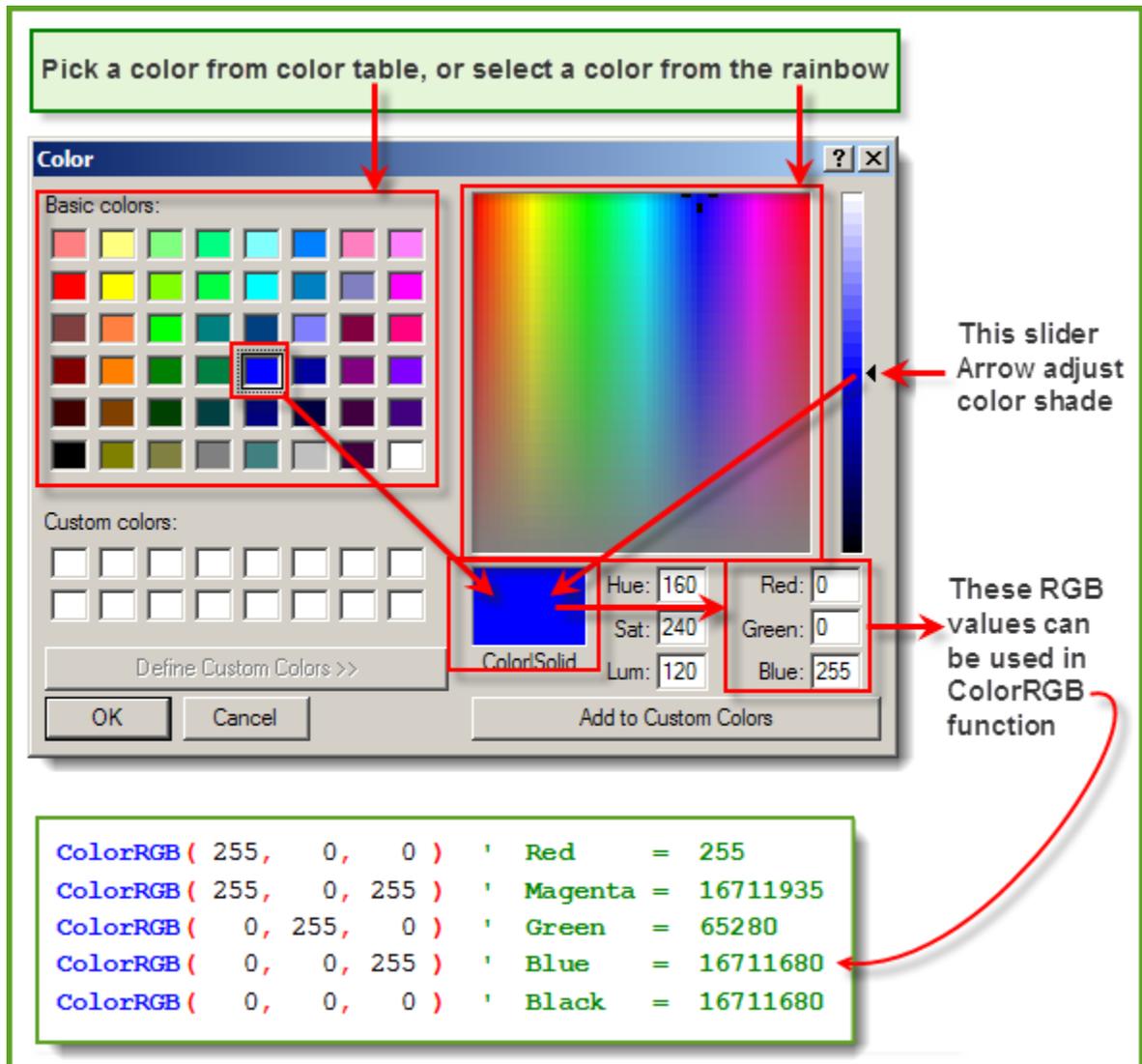
All BPV Series will provide access to the Trading Blox Color Selection Dialog when the BPV series uses a System, Test or Simulation Scope setting with a BPV numeric series.

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



Color Selection

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



Color Selection Options

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 the Trading Blox [ColorRGB](#) function requires, place the color numbers using Blue, Green and Red as the first, second and third parameter locations

Script Color Assignment Examples:

```

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

FileVersion

This function shows the current Trading Blox Builder version information as a string value.

Syntax:
<code>FileVersion</code>

Parameter:	Description:
N/A	Does not use parameters.

Example:
<pre>' Display Trading Blox Current Version information as a String value: PRINT "TB File-Version ", FileVersion</pre>
Returns:
TB File-Version 5.1.2

Links:
FileVersionNumerical
See Also:
General

FileVersionNumerical

This function will show the current Trading Blox Builder numerical version information as Long Integer value.

Syntax:

`FileVersionNumerical`

Parameter:

N/A

Description:

Does not use parameters.

Example:

```
' Display Trading Blox Current Numerical Version Number:
PRINT "FileVersionNumerical ", FileVersionNumerical
```

Returns:

FileVersionNumerical 5010200

Links:

[FileVersion](#)

See Also:

[General](#)

GetRegistryKey

Use this function to get a registry key value, that was set with the SetRegistryKey function

Syntax:
<code>keyValue = GetRegistryKey(keyName, [subKeyName])</code>

Parameter:	Description:
keyName	The name of the key, used by SetRegistryKey .
[subKeyName]	The name of the subKey .

Example:
<pre> SetRegistryKey("HelloWorld", "What a wonderful day.") PRINT GetRegistryKey("HelloWorld") SetRegistryKey("Count", 0) count = GetRegistryKey("Count") count = count + 1 ' Update the RegistryKey count SetRegistryKey("Count", count) ' Display GetRegistryKey Returned Value PRINT GetRegistryKey("Count") </pre>
Returns:
Returns the Count of the Registry Key

Links:
SetRegistryKey
See Also:
General

LicenseName

Function returns the Trading Blox Builder the registered user license name.

Syntax:

[LicenseName](#)

Parameter:**Description:**

N/A

Does not use parameters.

Example:

```
' Display the current registered License Name as a String value.  
PRINT "Registered User License Name: ", LicenseName
```

Returns:

Registered User License Name: **Registered-User-Name**

Links:**See Also:**

[General](#)

LineNumber

Use this for debugging purposes along with the **PRINT** function to know line number in script section where the information is located.

NOTE:

Always append an open and closed parentheses () symbol after function name (See examples).

Syntax:
<code>Print LineNumber()</code>

Parameter:	Description:
N/A	Does not use a parameter.

Example 1:
<pre>' Display the editor line number where the LineNumber function is located. PRINT "LineNumber ", LineNumber()</pre>

Return 1:
Returns the current line number of the script where this function is placed.

Example 2:
<pre>' Use LineNumber, and block.scriptName to know where ' Print Output information is located. to identify PRINT LineNumber(), "Enter any other values needed"</pre>

Return 2:
872,Enter any other values needed

Example 3:
<pre>' Print Script Information with a Line Number Reference PRINT "iLineNum", "ScriptName", "Why", "t.CurDate" PRINT iLineNum(), block.scriptName, sWhyMsg, test.currentDate</pre>

Return 3:								
<table> <thead> <tr> <th>iLineNum</th> <th>CallScript</th> <th>Why</th> <th>t.CurDate</th> </tr> </thead> <tbody> <tr> <td>24</td> <td>Before Trading Day</td> <td>Position Changes</td> <td>1/13/2015</td> </tr> </tbody> </table>	iLineNum	CallScript	Why	t.CurDate	24	Before Trading Day	Position Changes	1/13/2015
iLineNum	CallScript	Why	t.CurDate					
24	Before Trading Day	Position Changes	1/13/2015					

Links:
Block Object

Links:**See Also:**[General](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 402

Message Box

MessageBox function presents the user with a message box and stops processing of the test.

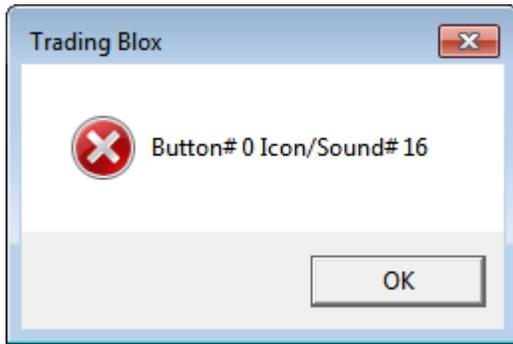
Syntax:	
<code>returnValue = MessageBox(message, [Button Options], [Icon and Sound])</code>	

Parameter:	Description:
<code>message</code>	The string message to display.
<code>[Button Options]</code>	Optional: The decimal number from the type1 list.
<code>[Icon and Sound]</code>	Optional: Then decimal number from the type2 list.
<code>returnValue</code>	Decimal number from the return value list.

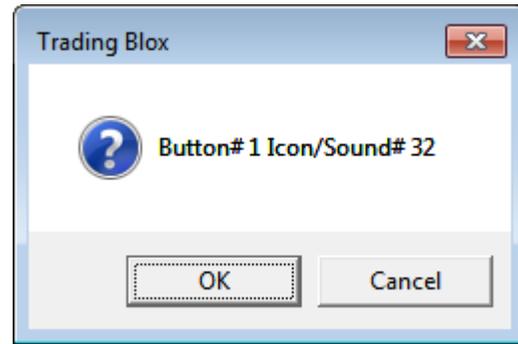
Button Options:	Icon and Sound Options:	Button Return Values:
0 -- OK	16 -- Icon X, Critical Stop	1 -- OK
1 -- OK/Cancel	Sound	2 -- Cancel
2 --	32 -- Icon Question, Ding	3 -- Abort
Abort/Retry/Ignore	Sound	4 -- Retry
3 -- Yes/No/Cancel	48 -- Icon Exclamation,	5 -- Ignore
4 -- Yes/No	Exclamation Sound	6 -- Yes
5 -- Retry/Cancel	64 -- Icon Information,	7 -- No
6 -- Cancel/Try	Error Sound	10 -- Try Again
Again/Continue	80 -- No Icon, Ding Sound	11 -- Continue
	128 -- No Icon, No Sound	

Example:
<pre>' 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)</pre>
Returns Any of these Dialog Examples:

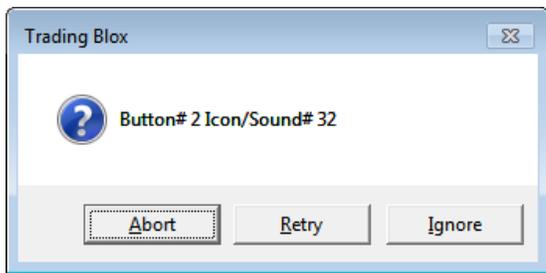
Example:



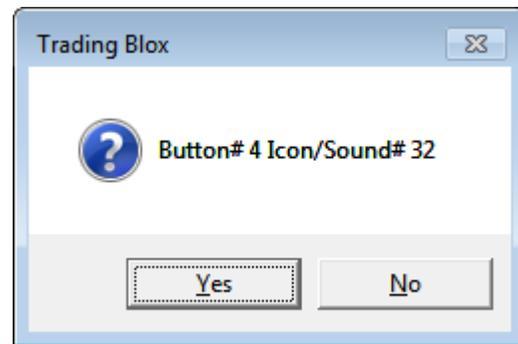
Button - OK & Icon X, Critical Stop Sound



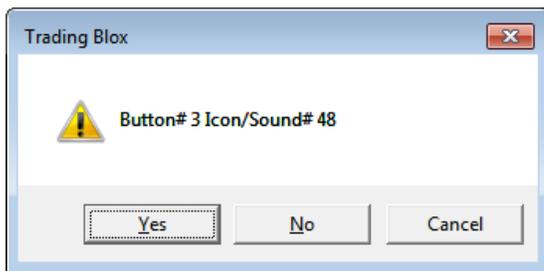
Button - OK/Cancel & Icon Question, Ding Sound



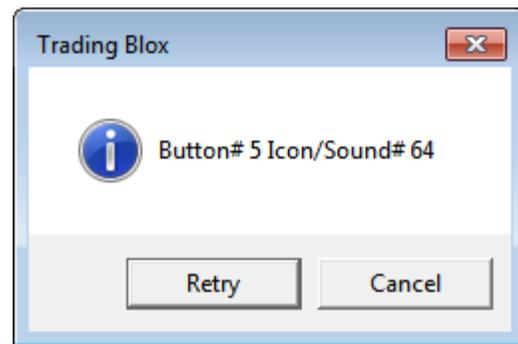
Button -Abort/Retry/Ignore & Icon Question, Ding Sound



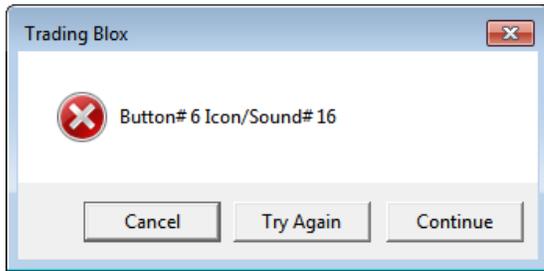
Button -Yes/No & Icon Question, Ding Sound



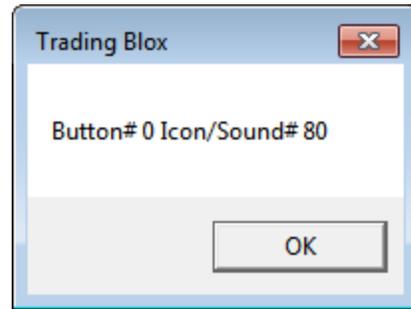
Button -Yes/No/Cancel & Icon Exclamation, Exclamation Sound



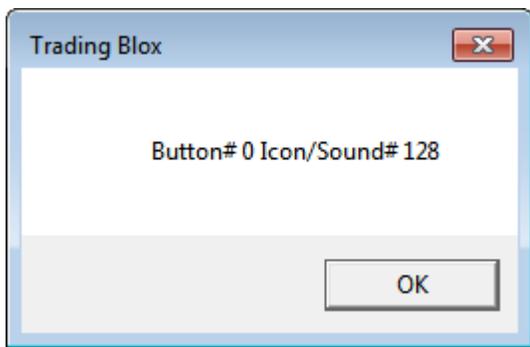
Button -Retry/Cancel & Icon Information, ERROR Sound

Example:

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 2:

```
' Ask User if it is OK to continue Processing Data
result = MessageBox( "Is it OK to process data for " + instrument.date, 3,
32 )

' 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 3:

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"

```

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 )  
  
' 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](#)

See Also:

[General](#)

PlaySound

Plays a sound file from the Sounds folder.

Syntax:

```
PlaySound( SoundFileName )
```

Parameter

:

Description:SoundFile
Name

The name of the sound file.

Example:

```
' Play the sound of the 'Test Done' sound file.  
PlaySound( "Test Done.wav" )
```

Returns:

Sound of sound file is played.

Links:**See Also:**[General](#)

Preference Items

Keyword property names.

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

Function returns the current Trading Blox Builder version as a String value.

Syntax:

`ProductVersion`

Parameter:**Description:**

N/A

Does not use parameters.

Example:

```
' Display current Trading Blox product version as a String value
PRINT "ProductVersion ", ProductVersion
```

Returns:

ProductVersion 5.1

Links:

[ProductVersionNumerical](#)

See Also:

[General](#)

ProductVersionNumerical

This function returns the current Trading Blox Builder numerical version information as a Long-Integer value.

Syntax:

```
ProductVersionNumerical
```

Parameter:**Description:**

N/A

Does not use parameters.

Example:

```
' Display Trading Blox Current Numerical Version Number as a Long-Integer:
PRINT "ProductVersionNumerical ", ProductVersionNumerical
```

Returns:

```
ProductVersionNumerical 5010000
```

Links:[FileVersion](#)**See Also:**[General](#)

SetRegistryKey

Use this function to set a registry key.

Syntax:
<code>SetRegistryKey(keyName, keyValue, [subKeyName])</code>

Parameter:	Description:
keyName	The name of the key, used by <code>SetRegistryKey</code> .
keyValue	The string value of the key numbers will be converted to strings
[subKeyName]	The string value of the key. numbers will be converted to strings.

Example:
<pre>' Example Ideas SetRegistryKey("HelloWorld", "What a wonderful day.") PRINT GetRegistryKey("HelloWorld") SetRegistryKey("Count", 0) count = GetRegistryKey("Count") count = count + 1 ' Update the RegistryKey count SetRegistryKey("Count", count) ' Display GetRegistryKey Returned Value PRINT GetRegistryKey("Count")</pre>
Returns:
Returns the Count of the Registry Key

Links:
GetRegistryKey
See Also:
General

Type

This keyword used as a function defines the type of [Local-Variables](#) created.

[Declaring Local-Variables](#) can be any of the following types shown in this table:

Variable Types:	Descriptions:
Integer	Are whole number values e.g. 1, -2, 400, 5, etc.
Floating-Point	These are fractional or decimal numbers e.g. 2.5, 1.414, etc.
Price	These are fractional or decimal numbers in the range of prices and are intended to be used when data formatting improves the understanding.
String	Text variables are used to labeling, describing and explaining.

Examples:

```
' -----
' Create Temp Variables Processing in this script section
VARIABLES: iDate, iInPortfolio Type: Integer
VARIABLES: sSymbol           Type: String
VARIABLES: fRateChange       Type: Floating
' -----
```

Links:

[About Data Variables](#), [Data Names](#), [Data Scope](#), [Data Types](#), [Type](#), [Variables](#), Variables Definition

See Also:

[General](#),

Variables

This keyword function is used to create Local Variables in a script section.

Local-Variables are created in a script section and are usually only available by script in that section. The exception to this limitation happens when another script section calls a Custom Script Section Name. When the calling script section has called a Custom Script Section the variables of the calling script and the Local Variables in the called script are available the calling script section. The variables in the calling script section and the Local Variables in the called script section can be accessed by all the active scripts.

[Declaring Local-Variables](#) can be any of the following types shown in this table:

Variable Types:	Descriptions:
Integer	Are whole number values e.g. 1, -2, 400, 5, etc.
Floating-Point	These are fractional or decimal numbers e.g. 2.5, 1.414, etc.
Price	These are fractional or decimal numbers in the range of prices and are intended to be used when data formatting improves the understanding.
String	Text variables are used to labeling, describing and explaining.

Examples:

```
' -----
' Create Temp Variables Processing in this script section
VARIABLES: iDate, iInPortfolio Type: Integer
VARIABLES: sSymbol           Type: String
VARIABLES: fRateChange       Type: Floating
' -----
```

Examples:

```
' -----
' Create Temp Variables Processing in this script section
VARIABLES: iDate, iInPortfolio Type: Integer
VARIABLES: sSymbol           Type: String
VARIABLES: fRateChange       Type: Floating
' -----
```

Links:

[About Data Variables, Data Names, Data Scope, Data Types, Type, Variables, Variables Definition](#)

See Also:

[General](#),

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 694

3.5 Math

Table descriptions will be added soon. Until then, click on function link to see description in the function's topic page.

Function:	Description:
AbsoluteValue	Returns the absolute value of a number.
ArcCosine	Returns the arc cosine of an angle specified in radians. The range of the result is 0 to PI radians.
ArcSine	Returns the arc sine of an angle specified in radians.
ArcTangent	Returns the arc tangent of an angle specified in radians.
ArcTangentXY	Returns the arc tangent of an angle specified in radians by the number x / y .
Average	Simple moving average that returns the sum of the values divided by the number of values.
CAGR	The CAGR function will return the Compounded Annual Growth Rate as computed internally by Trading Blox.
Ceiling	Ceiling returns 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.
Correlation	Correlation uses the Pearson's correlation coefficient method to return the statistical coefficient for the range of bars between series.
CorrelationLog	
Cosine	Returns the arc sine of an angle specified in radians.
DegreesToRadians	Returns the angle in radians corresponding with an angle specified in degrees.
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.
Exponent	Returns e (2.71828182845904, Base of natural logarithms) raised to a power.
Floor	Floor returns 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.
Hypotenuse	Calculates the length of the hypotenuse of a right triangle, given the length of the two sides sideOne and sideTwo.
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.
IsUndefined	Returns True if the variable is undefined.

Function:	Description:
Log	Returns the logarithm of a number.
Max	Returns the highest value in a list of values.
Min	Returns the lowest value in a list of values.
RadiansToDegrees	Returns the angle in degrees corresponding with an angle specified in radians.
Random	Returns a random integer given a range of integers.
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.
Round	Returns the rounded value.
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.
Sine	Returns the sine of an angle specified in radians.
Square Root	Returns the square root of the specified number.
StandardDeviation	By default, Trading Blox Builder Standard Deviation function uses Population formula instead of Sample formula to be more consistent with industry standards. The Standard Deviation formula used can be changed.
StandardDeviationLog	
SumValues	Returns the sum of a list of values.
Tangent	Returns the tangent of an angle specified in radians.

AbsoluteValue

Returns the absolute value of a number.

Short Form name: "[Abs](#)"

Syntax:

```
value = AbsoluteValue( expression )
```

Parameter:**Description:**

expression	Any expression or numeric value.
------------	----------------------------------

Example:

```
value = AbsoluteValue( 5.6 )      ' Returns 5.6
value = AbsoluteValue( -5.6 )     ' Returns 5.6
value = AbsoluteValue( "Hello" )  ' Returns 0.
value = Abs( -1 * 45 )            ' Returns 45.
```

Returns:

Absolute value of expression

Links:**See Also:**

[Mathematical Functions](#)

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 name: "Acos"

Syntax:

```
value = ArcCosine( expression )
```

Parameter:	Description:
expression	Any value or expression that resolves to a valid cosine range -1 to 1

Example:

```
value = ArcCosine( 0 )           ' Returns 1.570795327
value = ArcCosine( 0.5 )        ' Returns 1.047197551
value = RadiansToDegrees( ArcCosine( 0.5 ) ) ' Returns 60
```

Returns:

Arc cosine of expression.

Links:

[RadiansToDegrees](#)

See Also:

[Mathematical Functions](#)

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 name: "[Asin](#)"

Syntax:

```
value = ArcSine( expression )
```

Parameter:	Description:
expression	Any value or expression that resolves to an angle in radians.

Example:

```
value = ArcSine( PI )      ' Returns 5.6.  
value = ArcSine( PI / 2 ) ' Returns 5.6.
```

Returns:

Arc sine of expression.

Links:

[RadiansToDegrees](#)

See Also:

[Mathematical Functions](#)

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 name: "[Atan](#)"

Syntax:

```
value = ArcTangent( expression )
```

Parameter:	Description:
expression	Any expression that resolves to an angle in radians.

Example:

```
value = ArcTangent( PI )      ' Returns 5.6  
value = ArcTangent( PI / 2 ) ' Returns 5.6
```

Returns:

[RadiansToDegrees](#)

Links:

See Also:

[Mathematical Functions](#)

ArcTangentXY

Returns the arc tangent of an angle specified in radians by the number x / y .

Returns result in radians within the range of $-\pi$ to π 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 name: "[Atan2](#)"

Syntax:

```
value = ArcTangentXY( X, Y )
```

Parameter:	Description:
X	Any value or expression.
Y	Any value or expression.

Example:

```
value = ArcTangentXY( PI, 2 ) ' Returns 5.6
```

Returns:

Arc tangent of (x / y)

Links:

See Also:

[Mathematical Functions](#)

Average

Finds the average value of the series.

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:

```
value = Average( series, bars, [offset] )
```

Parameter:	Description:
series	Name of the series.
bars	Number of bars over which to find the value.
[offset]	Number of bars to offset before finding the value.

Example:

```
VARIABLES: highestClose, highestHigh, lowestLow TYPE: Price
```

```
VARIABLES: 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 )
```

Returns:

Average value found by taking the sum all values in the series and then divided by the bar value.

Links:

[Median](#)

See Also:

[Series Functions](#)

Edit Time: 9/11/2020 4:48:24 PM

Topic ID#: 158

CAGR

The CAGR function will return the Compounded Annual Growth Rate as computed internally by Trading Blox.

Syntax:

```
CAGR( days , starting_value , ending_value )
```

Parameter:	Description:
days	Number daily results to use.
starting_value	Beginning value to use.
ending_value	Ending value to use.

Example:

```
' Trading Blox uses the following formula to compute CAGR  
CAGR = ( ending value / starting value ) ^ ( 1 / ( days / 365.25 ) ) - 1
```

Returns:

Compounded Annual Growth Rate for the selected period and range of values.

Links:

See Also:

[Mathematical Functions](#)

Ceiling

Ceiling returns 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:**

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
' ~~~~~
```

Returns:**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:

[AsInteger](#), [AsString](#), [Floor](#), [FOR](#), [Round](#)

See Also:

[Mathematical Functions](#)

Correlation

Correlation is a series function that uses the Pearson's correlation coefficient method to return the statistical coefficient for the range of bars between two markets entered into the first two parameters of this function.

Syntax:

```
Correlation( series1, series2, barsToMeasure, [offset1], [offset2] )
```

Parameter:	Description:
series1	Series element/property value of the first series.
series2	Series element/property value of the second series.
barsToMeasure	Number of bars over which to measure the correlation coefficient.
[offset1]	Optional: Offset value for the first series.
[offset2]	Optional: Offset value for the second series.

Example:

```
' Load first parameter's instrument series
soybeans.LoadSymbol( "S" )
' Load second parameter's instrument series
gold.LoadSymbol( "GC" )
' Calculate the coefficient value between the two series
coefficient = Correlation( soybeans.close, gold.close, 500 )
```

Returns:

Above example will returns the correlation coefficient between GC and S over the last 500 days.

Return Range Values:	Correlation Meaning:
1.0	Value 1 indicates there is a perfect positive correlation between the two series. <ul style="list-style-type: none"> ○ Values less than 0.7 is generally considered Uncorrelated. ○ Values between 0.7 to 0.9 is generally considered Loosely correlated. ○ Values greater than 0.9 is considered Closely correlated.
0	Value of 0 indicates there is no correlation between the two series
-1.0	Value of -1 indicates the two series have a perfect negative correlation. <ul style="list-style-type: none"> ○ Values greater than -0.7 is generally considered Uncorrelated. ○ Values between -0.9 to -0.7 is generally considered Loosely correlated. ○ Values less than -0.9 is considered Closely correlated.

Links:

[Correlation Properties](#), [CorrelationSynch](#), [CorrelationLogSynch](#), [MaxSynchBars](#)

See Also:

[Mathematical Functions](#), [Series Functions](#),

CorrelationLog

The CorrelationLog function returns the statistical correlation between two series, for the specified number of bars. These series can be any list of numbers, and does not have to be market data. Can be auto indexed or non auto indexed, and can be IPV or BPV.

The Correlation function uses the actual values such as: Value1 and Value 2.

The CorrelationLog function uses the change in the log of the values in the series such as: $\text{Log}(\text{Value1}) - \text{Log}(\text{Value2})$.

Syntax:

```
CorrelationLog( series1, series2, barsToMeasure, [offset1], [offset2] )
```

Parameter :	Description:
barsToMeasure	Number of bars over which to measure the correlation.
series1	First series name.
series2	Second series name.
offset1	Offset of the first series.
offset2	Offset of the second series.

Returns:

Statistical correlation for last period length in the barsToMeasure parameter.

Example:

```
' Load Soybeans in the BPV Instrument "soybeans"
soybeans.LoadSymbol( "S" )

' Load Gold in the BPV instrument "gold"
gold.LoadSymbol( "GC" )

' Return the Correlation between GC (Gold)
' and S (Soybeans) over the last 500 days.
Correlation = CorrelationLog( soybeans.close, gold.close, 500 )
```

Results:

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.

Example:

Returns **1** If the two series are perfectly positively correlated.

Generally **0.7** OR greater is considered loosely correlated and **0.9** OR greater is considered closely correlated.

Links:

[Correlation](#), [CorrelationLogSynch](#), [CorrelationSynch](#), [MaxSynchBars](#)

See Also:

[Series Functions](#)

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 229

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 name: "Cos"

Syntax:

```
value = Cosine( expression )
```

Parameter:	Description:
expression	Any expression that resolves to an angle in radians.

Example:

```
value = Cosine( PI )      ' Returns 1.0.  
value = Cosine( PI / 2 ) ' Returns 0.0 approximately.
```

Returns:

Cosine of expression.

Links:

[RadiansToDegrees](#)

See Also:

[Mathematical Functions](#)

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 name: "DegToRad"

Syntax:

```
value = DegreesToRadians( expression )
```

Parameter:	Description:
expression	Any expression that resolves to an angle in degrees.

Example:

```
value = DegreesToRadians( 180 )    ' Returns 5.6.  
value = DegreesToRadians( PI / 2 ) ' Returns 5.6.
```

Returns:

Radians corresponding with expression.

Links:

[RadiansToDegrees](#)

See Also:

[Mathematical Functions](#)

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.

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". Or more simply, the 10 day exponential moving average of the close would be defined as follows, where closeEMA is a series variable.

Syntax:

```
value = EMA( lastValueOfSeries, movingAverageDays, newValue )
```

Parameter:	Description:
lastValueOfSeries	Previous value in the series
movingAverageDays	Number of days in the moving average
newValue	New data value to include in EMA series.

Example:

```
' Example Name References
' EMA( closeChangeMovingAverage[1], _
'     daysInMovingAverage, _
'     instrument.close - instrument.close[1] )
closeEMA = EMA( closeEMA[1], 10, instrument.close )
```

Returns:

Current EMA value after the data value included.

Links:

[Average](#)

See Also:

[Mathematical Functions](#)

Exponent

Returns e (2.71828182845904, Base of natural logarithms) raised to a power.

See also the [power operator ^](#) if you want to raise 10 to some number.

Short form: "[Exp](#)".

Syntax:

```
value = Exp( expression )
```

Parameter:	Description:
expression	Any expression or value that defines the power.

Example:

```
value = Exp( 0 ) ' Returns 1  
value = Exp( 1 ) ' Returns 2.718282  
value = Exp( 2 ) ' Returns 7.389056
```

Returns:

Value raised to specified power.

Links:

[Log](#)

See Also:

[Mathematical Functions](#)

Floor

Floor returns 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:

```
value = 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
' ~~~~~
```

Returns:

```
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:

[AsInteger](#), [AsString](#), [Ceiling](#), [FOR](#), [PRINT](#)

See Also:

[Mathematical Functions](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 345

Hypotenuse

Calculates the length of the hypotenuse of a right triangle, given the length of the two sides sideOne and sideTwo.

Short form name: "[Hypot](#)"

Syntax:

```
value = Hypotenuse( sideOne, sideTwo )
```

Parameter:	Description:
sideOne	Any value, or expression used to define the length of the first side.
sideTwo	Any value, or expression used to define the length of the second side.

Example:

```
value = Hypotenuse( 3, 4 ) ' Returns 5  
value = Hypotenuse( 9, 16 ) ' Returns 25
```

Returns:

Length of the hypotenuse.

Links:

See Also:

[Mathematical Functions](#)

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 )
```

Parameter:	Description:
condition	Condition, or statement that evaluates to TRUE or FALSE
trueValue	Value returned if condition is TRUE
falseValue	Value returned if condition is FALSE

Example:

```
' 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 )

value = IfThenElse( 1 = 2, 3, 4 ) ' Returns 4
value = IfThenElse( 2 = 2, 3, 4 ) ' Returns 3
```

Returns:

True-Value, if condition is TRUE, otherwise False-Value.

Links:**See Also:**

[Mathematical Functions](#)

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 )
```

Parameter:**Description:**

Parameter:	Description:
variable	Variable to evaluate.

Example:

```
notDefined = IsUndefined( instrument.defaultAverageTrueRange )
```

Returns:

True if the variable is undefined.

Links:

[defaultAverageTrueRange](#)

See Also:

[Mathematical Functions](#)

Log

Returns the logarithm of a number.

Syntax:

```
value = Log( number [, base] )
```

Parameter:

Description:

number

Any expression, or statement.

[base]

The base of the logarithm, if omitted function will return the natural logarithm (assumes base **e**)

Example:

```
value = Log( 1 )           ' Returns 0
```

```
value = Log( 16, 2 )      ' Returns 4
```

```
value = Log( 100, 10 )   ' Returns 2
```

Returns:

Number's log value

Links:

[Exponent](#)

See Also:

[Mathematical Functions](#)

Max

Returns the highest value in a list of values. List of values can be any number of arguments that create a value, or a list of values. All values must be separated by a comma.

Max is not compatible for with numeric series. To find the highest or largest value in an entire or partial segment of numbers in a series see the function [Highest](#).

Syntax:

```
value = Max( expression1, expression2, expression3, ... )
```

Parameter:	Description:
expression1	Any numeric expression.
expression2	Any numeric expression.
expression3	Any numeric expression.
...	Additional numeric expressions, if they are needed.

Example:

```
' Find the Highest value of the expression
' values included within the parentheses
value = Max( 5, 6, 8 )           ' Returns 8
value = Max( 2 + 1, 5 )        ' Returns 5
```

Returns:

Max returns the highest value of the values compared.

Links:

[Highest](#), [Lowest](#), [Min](#)

See Also:

[Mathematical Functions](#)

Min

Returns the lowest value in a list of values. List of values can be any number of arguments that create a value, or a list of values. All values must be separated by a comma.

Min is not compatible for with numeric series. To find the lowest or smallest value in an entire or partial segment of numbers in a series see the function [Lowest](#).

Syntax:

```
value = Min( expression1, expression2, expression3, ... )
```

Parameter:	Description:
expression1	Any numeric expression
expression2	Any numeric expression
expression3	Any numeric expression
...	Additional numeric expressions, if they are needed.

Example:

```
' Find the Highest value of the expression
' values included within the parentheses
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)
```

Returns:

Lowest returns the lowest value of the values compared.

Links:

[Highest](#), [Lowest](#), [Max](#)

See Also:

[Mathematical Functions](#)

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 420

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 name: "RadToDeg"

Syntax:

```
value = RadiansToDegrees( expression )
```

Parameter:	Description:
expression	Any expression.

Example:

```
value = RadiansToDegrees( PI )      ' Returns 360  
value = RadiansToDegrees( PI / 2 ) ' Returns 180
```

Returns:

Angle in degrees corresponding with the angle specified in radians.

Links:

See Also:

[Mathematical Functions](#)

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:
<code>value = Random(range [, or lowerValue], [upperValue])</code>

Parameter:	Description:
range	A range value, or the Lower value of range.
[, or lowerValue]	Optional: Lower value of range.
[upperValue]	Optional: Upper value of range.

Example:
<pre>' Returns a random number from 1 to 10 PRINT Random(10) ' Returns a random number from 10 to 20 PRINT Random(10, 20)</pre>
Returns:
A random value that is within the range values specified.

Links:
RandomDouble , RandomSeed
See Also:
Mathematical Functions

RandomDouble

Returns a random double between 0 and 1.

Syntax:

```
value = RandomDouble()
```

Parameter:	Description:
N/A	Function doesn't use parameters.

Example:

```
' Returns a random double between 0 and 1  
PRINT RandomDouble()
```

Returns:

Returns any random double value between 0 and 1.

Links:

[Random](#), [RandomSeed](#)

See Also:

[Mathematical Functions](#)

RandomSeed

Seeds the random number generator with the optional seed value. A time value is used when the Seed-Value is excluded so that the sequence of random numbers is different every time a test is run.

If you don't use the **RandomSeed** function, the Random function will return the same sequence of random numbers for every simulation run.

When you need to create a repetitive series of values while debugging a problem, do not use the RandomSeed function. Otherwise, it is usually a good practice to generate a random different sequence of numbers each time.

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 ] )
```

Parameter:**Description:**

```
[ seedValue ]
```

Optional Seed-Value.

Example:

```
' 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 )
```

Returns:**Links:****See Also:**

[Mathematical Functions](#)

Round

Returns the rounded value.

Syntax:

```
value = Round( expression1, decimals )
```

Parameter:**Description:**

expression1	Any numeric expression
decimals	Number of decimals to round too.

Example:

```
value = Round( 5.123456, 2 )      ' Returns 5.12
value = Round( 1.25 + 1.5, 1 )    ' Returns 2.8
value = Round( 12345, -2 )        ' Returns 12300
```

Returns:

Lowest value of the list of expressions.

Links:

[AsInteger](#), [Ceiling](#), [Floor](#)

See Also:

[Mathematical Functions](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 508

RSquared**Syntax:**

```
RSquared
```

Parameter:**Description:**

--	--

Returns:

Example:

Links:

See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 729

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:**Description:**

Parameter:	Description:
AnyValue	Any numeric value, Or numeric expression that results in a numeric value.

Example:

```
' ~~~~~
' 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
```

Example:

```
' ~~~~~  
' Show Sign of Last Calculation  
' ~~~~~  
If Sign( AnyNumber ) = TRUE THEN  
    PRINT "Sign( AnyNumber ) is Positive"  
ELSE  
    PRINT "Sign( AnyNumber ) is Negative"  
ENDIF
```

Returns:

Returns a 1 when the value is positive, and a -1 when the value is negative.

Links:**See Also:**

[Mathematical Functions](#)

Sine

Returns the sine of an angle specified in radians.

The range of the result is 0 to π radians.

To convert angle from radians to degrees use [RadiansToDegrees](#) function.

Short form: "sin"

Syntax:

```
value = Sine( expression )
```

Parameter:

Description:

expression

Any expression that resolves to an angle in radians.

Example:

```
value = Sin( PI )           ' Returns 0.0 approximately  
value = Sine( PI / 2 )     ' Returns 1.0  
value = Sine( DegreesToRadians( 30 ) ) ' Returns 0.5
```

Returns:

Sine of the angle specified in radians.

Links:

[DegreesToRadians](#), [RadiansToDegrees](#)

See Also:

[Mathematical Functions](#)

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 )
```

Parameter:

Parameter:	Description:
expression	Any numeric value or expression.

Example:

```
value = SquareRoot( 4 ) ' Returns 2
value = SquareRoot( -4 ) ' Returns 2
value = SquareRoot( 2 ) ' Returns 1.414.
Value = Sqr( 9 ) ' Returns 3
```

Returns:

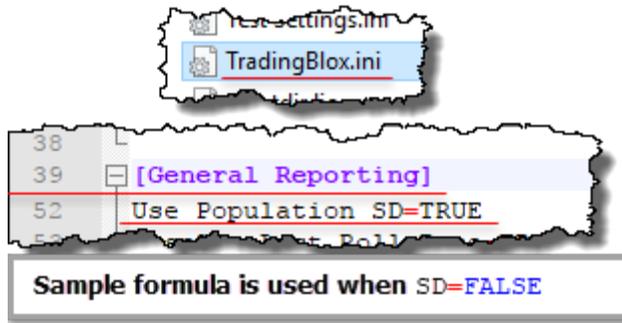
Square root of the specified number.

Links:**See Also:**

[Math Functions](#)

StandardDeviation

By default, Trading Blox Builder Standard Deviation function uses Population formula instead of Sample formula to be more consistent with industry standards. To change the default to the Sample formula, access the Trading Blox Builder.ini file:



Trading Blox installation folder default setting

In terms of the Sample vs. Population Standard Deviation, TRUE enables Population formula, FALSE enables Sample formula.

This For information on using functions with non auto indexed series review [Series Functions](#).

Syntax:

```
x = StandardDeviation( series, bars, [offset] )
```

Parameter:	Description:
series	The name of the series.
bars	The number of bars over which to find the value.
[offset]	The optional number of bars to offset before finding the value.

Example:

```
' Example shows the use of the common auto indexed series.
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 )
```

Returns:

Standard Deviation value for each location in a series.

Links:

[Series Functions](#)

See Also:

[Math Functions](#)

StandardDeviationLog

Finds the standard deviation of the series. Uses the change in the log of the values.

Syntax:	
<code>StandardDeviationLog(series, bars, [offset])</code>	

Parameter:	Description:
<code>series</code>	The name of the series
<code>bars</code>	The number of bars over which to find the value
<code>[offset]</code>	The number of bars to offset before finding the 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 = StandardDeviationLog( instrument.close, 100 )
```

Returns:

The standard deviation of the series used over the period specified.

This example shows the use of a common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

Links:

Links:**See Also:**[Math Functions](#), [Series Functions](#)

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, ... )
```

Parameter:	Description:
expression1	Any numeric expression.
expression2	Any numeric expression.
expression3	Any numeric expression.
...	Any numeric expression, if more are needed.

Example:

```
value = SumValues( 5, 6, 8 )           ' Returns 19  
value = SumValues( 2 + 1, 5 )         ' Returns 8
```

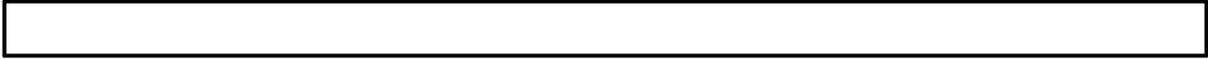
Returns:

Returns the sum of the values.

Links:

See Also:

[Mathematical Functions](#)



Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 594

Tangent

Returns the tangent of an angle specified in radians.

Short Form name: "Tan"

Syntax:

```
value = Tangent( expression )
```

Parameter:	Description:
------------	--------------

expression	Any expression that resolves to an angle in radians.
------------	--

Example:

```
value = Tangent( PI )      ' Returns 0.0 approximately  
value = Tangent( PI / 4 ) ' Returns 1.0
```

Returns:

Tangent of the specified angle

Links:

See Also:

[Mathematical Functions](#)

3.6 Series

The following functions can be used with series variables.

[Data series Indexing](#) information.

Function Name:	Description:
AsDate	This function that converts an integer date YYYYMMDD to string "YYYY-MM-DD" for PRINTing or other string output.
AsSeries	Function Passes the address of the series.
Average	Finds the average value of the series.
CopySeries	This Series functions will Copy the data in a from one series to another, and also transpose a row or column from a dual array to single.
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.
DownloadWebFile()	FTPDownload and DownloadWebFile now take optional local folder parameter
FTPDownloadFile()	
FTPUploadFile()	
GetReference	Deprecated. Use AsSeries() to pass a series element 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.

Function Name:	Description:
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.
SetSeriesAutoIndex	SetSeries Enable for IPV and BPV and numeric and string. (Test)
SetAuxiliaryWindowText	
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. Manual-Indexed series have a minimum size of 1, and a maximum size of 1,000,000 elements.
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.

Last Edit: 9/11/2020

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 527

AsDate

This function that converts Integer date **YYYYMMDD** into string "**YYYY-MM-DD**" that can be assigned to a String-name variable. As a date formatted String, the text date will display in its formatted date when it appears after a **PRINT** `textDate` statement.

Syntax:

```
AsDate(dateValue)
```

Parameter:

dateValue

Description:

Any date value as an Integer value.

Returns:

Converts an integer valid date value: `20200315` into a string/text that will appear as: `"2020-03-15"`

Example:

```
Variables: textDate TYPE: STRING
```

```
textDate = AsDate(20200315)
```

```
Print textDate
```

```
Output "2020-03-15"
```

Links:

[PRINT](#), [Variables](#), [Series](#)

See Also:

[String](#)

GCD

Used to determine minimum tick of price series.

A return where the Precision is 2, assumes 2 decimals (.01) prices, A return of 4, assumes 4 decimals (.0001).

Syntax:

```
GCD(series, elementCount, precision)
```

Parameter:

series

elementCount

precision

Description:**Returns:****Example:****Links:****See Also:**

AsSeries

Changes how a **TYPE SERIES** array is passed to a [Chart Object](#) or [Scrip Object](#) Custom function.

Most functions can accept values from variables by the value in the variable that is passed to the function. Series data cannot be passed by value, unless a single indexed element is being passed. To pass all the elements in a series array it must be passed by reference. **AsSeries()** provides the reference information required to access the elements in the series.

Script Execute functions and Custom Chart Director functions require the first elements memory location of data's for the first element the series. Charting functions also require data passed to their functions also include an element count.

This function is not required for static variables or properties that are not an Auto-Index or Manually Indexed series.

Syntax:

```
AsSeries( AnySeries )
```

Parameter:

Description:

AnySeries

Any series assigned to any Custom Chart or Script Object parameter is required to use this conversion function.

Example:

```
' Add 5 element values to represent "bar1"
chart.AddBarSeries( AsSeries( bar1 ), 5 )

' Passing Series to a Custom Function
script.Execute("ExampleFunction", AsSeries(anySeries))
```

Links:

[Chart Object](#), [AsFloating](#), [AsInteger](#), [AsString](#), [IsFloating](#), [IsInteger](#), [IsString](#), [Scrip Object](#)

See Also:

[Data Groups and Types](#)

Average

Finds the average value of the series.

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:

```
value = Average( series, bars, [offset] )
```

Parameter:	Description:
series	Name of the series.
bars	Number of bars over which to find the value.
[offset]	Number of bars to offset before finding the value.

Example:

```
VARIABLES: highestClose, highestHigh, lowestLow TYPE: Price
```

```
VARIABLES: 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 )
```

Returns:

Average value found by taking the sum all values in the series and then divided by the bar value.

Links:

[Median](#)

See Also:

[Series Functions](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 156

CopySeries

Syntax:
CopySeries(Series1, Series2)

Parameter:	Description:
CopySeries	

Returns:

Example:

Links:

See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 661

Correlation

Correlation is a series function that uses the Pearson's correlation coefficient method to return the statistical coefficient for the range of bars between two markets entered into the first two parameters of this function.

Syntax:

```
Correlation( series1, series2, barsToMeasure, [offset1], [offset2] )
```

Parameter:	Description:
series1	Series element/property value of the first series.
series2	Series element/property value of the second series.
barsToMeasure	Number of bars over which to measure the correlation coefficient.
[offset1]	Optional: Offset value for the first series.
[offset2]	Optional: Offset value for the second series.

Example:

```
' Load first parameter's instrument series
soybeans.LoadSymbol( "S" )
' Load second parameter's instrument series
gold.LoadSymbol( "GC" )
' Calculate the coefficient value between the two series
coefficient = Correlation( soybeans.close, gold.close, 500 )
```

Returns:

Above example will returns the correlation coefficient between GC and S over the last 500 days.

Return Range Values:	Correlation Meaning:
1.0	Value 1 indicates there is a perfect positive correlation between the two series. <ul style="list-style-type: none"> ○ Values less than 0.7 is generally considered Uncorrelated. ○ Values between 0.7 to 0.9 is generally considered Loosely correlated. ○ Values greater than 0.9 is considered Closely correlated.
0	Value of 0 indicates there is no correlation between the two series
-1.0	Value of -1 indicates the two series have a perfect negative correlation. <ul style="list-style-type: none"> ○ Values greater than -0.7 is generally considered Uncorrelated. ○ Values between -0.9 to -0.7 is generally considered Loosely correlated. ○ Values less than -0.9 is considered Closely correlated.

Links:

[Correlation Properties](#), [CorrelationSynch](#), [CorrelationLogSynch](#), [MaxSynchBars](#)

See Also:

[Mathematical Functions](#), [Series Functions](#),

CorrelationLog

The CorrelationLog function returns the statistical correlation between two series, for the specified number of bars. These series can be any list of numbers, and does not have to be market data. Can be auto indexed or non auto indexed, and can be IPV or BPV.

The Correlation function uses the actual values such as: Value1 and Value 2.

The CorrelationLog function uses the change in the log of the values in the series such as: $\text{Log}(\text{Value1}) - \text{Log}(\text{Value2})$.

Syntax:

```
CorrelationLog( series1, series2, barsToMeasure, [offset1], [offset2] )
```

Parameter	Description:
barsToMeasure	Number of bars over which to measure the correlation.
series1	First series name.
series2	Second series name.
offset1	Offset of the first series.
offset2	Offset of the second series.

Returns:

Statistical correlation for last period length in the barsToMeasure parameter.

Example:

```
' Load Soybeans in the BPV Instrument "soybeans"
soybeans.LoadSymbol( "S" )

' Load Gold in the BPV instrument "gold"
gold.LoadSymbol( "GC" )

' Return the Correlation between GC (Gold)
' and S (Soybeans) over the last 500 days.
Correlation = CorrelationLog( soybeans.close, gold.close, 500 )
```

Results:

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.

Example:

Returns **1** If the two series are perfectly positively correlated.

Generally **0.7** OR greater is considered loosely correlated and **0.9** OR greater is considered closely correlated.

Links:

[Correlation](#), [CorrelationLogSynch](#), [CorrelationSynch](#), [MaxSynchBars](#)

See Also:

[Series Functions](#)

CorrelationLogSynch

This function returns the correlation value of two IPV series. The series are date synched, to remove holidays or other missing data, so each record compared between the series have the same date. The [CorrelationLogSynch](#) function uses the log of the (rate of) change between the elements in the series, whereas the [CorrelationLog](#) function uses just the rate of change between the elements in the series. To avoid computing the Log of a negative number, us the "uc" parameter if the back adjusted futures data might go negative.

Because these are date synched, these Synch functions only work with auto indexed IPV series.

Syntax:

```
CorrelationLogSynch( series1, series2, sampleSize, offset1, offset2,
"uc" )
```

Parameter:	Description:
series1	The Series 1 name.
series2	The Series 2 name.
sampleSize	Number of bars over which to measure the correlation
offset1	Offset value for the first series.
offset2	Offset value for the second series.
"uc"	Use the unadjusted close as the rate of change divisor, in case the adjusted price goes negative.

Example:

```

' Run the correlation process once a month
If Month( test.currentDate ) <> lastMonth THEN

  ' Reset/Clear previous correlations information
  instrument.ResetCloselyCorrelated
  instrument.ResetLooselyCorrelated

  ' Loop through all symbols in the portfolio so
  ' their correlation can be evaluated.
  For index = 1 TO system.totalInstruments STEP 1

    ' Load the next instrument
    inst.LoadSymbol( index )

    ' Get the correlation between the two instruments
    ' over the last correlationPeriod days
    If instrument.bar > correlationPeriod * SynchCorrelationBuffer AND
       inst.bar > correlationPeriod * SynchCorrelationBuffer THEN

      ' Calculate Log value of Synch'd comparison record
      correlationValue = CorrelationLogSynch( instrument.close, _
                                             inst.close, _
                                             correlationPeriod )

    ELSE
      ' Clear the working variable when conditions don't match
      correlationValue = 0
    ENDIF

    ' Determine if the value is closely or loosely
    ' correlated before adding it to correlation matrix
    If correlationValue > closeThreshold THEN
      ' Add Loaded Symbol to Closely Correlated record series
      instrument.AddCloselyCorrelated( inst.symbol )
    ELSE
      ' Add Loaded Symbol to Loosely Correlated record series
      If correlationValue > looseThreshold THEN
        instrument.AddLooselyCorrelated( inst.symbol )
      ENDIF
    ENDIF
  Next ' index

  ' Save the month so we know when it changes
  lastMonth = Month( test.currentDate )
ENDIF ' Month( test.currentDate ) <> lastMonth

```

Returns:**Links:**

[MaxSynchBarsCorrelation Properties](#), [CorrelationLog](#), [CorrelationSynch](#), [CorrelationLogSynch](#), [MaxSynchBars](#)

Links:

See Also:

[Mathematical Functions](#), [Series Functions](#),

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 230

CorrelationSynch

This function returns the correlation value of two IPV series. The series are date synched, to remove holidays or other missing data, so each record compared between the series have the same date. The [CorrelationLogSynch](#) function uses the log of the (rate of) change between the elements in the series, whereas the [CorrelationLog](#) function uses just the rate of change between the elements in the series. To avoid computing the Log of a negative number, us the "uc" parameter if the back adjusted futures data might go negative.

Because these are date synched, these Synch functions only work with auto indexed IPV series.

Syntax:

```
CorrelationSynch( series1, series2, sampleSize, offset1, offset2, "uc" )
```

Parameter:	Description:
series1	Series element/property value of the first series.
series2	Series element/property value of the second series.
sampleSize	Number of bars over which to measure the correlation coefficient.
offset1	Offset value for the first series.
offset2	Offset value for the second series.
"uc"	Use the unadjusted close as the rate of change divisor, in case the adjusted price goes negative.

Example:

```
' Run the correlation process once a month
```

Example:

```

' Can run every day or bar if desired.
If Month( test.currentDate ) <> lastMonth THEN

  ' Reset/Clear previous correlations information
  instrument.ResetCloselyCorrelated
  instrument.ResetLooselyCorrelated

  ' Loop through all symbols in the portfolio so
  ' their correlation can be evaluated.
  For index = 1 TO system.totalInstruments STEP 1

    ' Load the next instrument
    inst.LoadSymbol( index )

    ' Get the correlation between the two instruments
    ' over the last correlationPeriod days
    If instrument.bar > correlationPeriod * SynchCorrelationBuffer AND
      inst.bar > correlationPeriod * SynchCorrelationBuffer THEN

      ' Calculate Log value of Synch'd comparison record
      correlationValue = CorrelationSynch( instrument.close,
inst.close, correlationPeriod )
    ELSE
      ' Clear the working variable when conditions don't match
      correlationValue = 0
    ENDIF

    ' Determine if the value is closely or loosely
    ' correlated before adding it to correlation matrix
    If correlationValue > closeThreshold THEN
      ' Add Loaded Symbol to Closely Correlated record series
      instrument.AddCloselyCorrelated( inst.symbol )
    ELSE
      ' Add Loaded Symbol to Loosely Correlated record series
      If correlationValue > looseThreshold THEN
        instrument.AddLooselyCorrelated( inst.symbol )
      ENDIF
    ENDIF
  Next ' index

  ' Save the month so we know when it changes
  lastMonth = Month( test.currentDate )
ENDIF ' Month( test.currentDate ) <> lastMonth

```

Returns:**Links:**

[Correlation Properties](#), [CorrelationLog](#), [CorrelationSynch](#), [CorrelationLogSynch](#), [MaxSynchBars](#)

See Also:

Links:[Mathematical Functions](#), [Series Functions](#),

CrossOver

Finds the Cross-Over timing and relationship of two data series.

Syntax:

```
CrossOver( series1, series2, [direction], [crossOverSize], [offset] )
```

Parameter:	Description:
series1	The first data series to compare to the second data series.
series2	The second data series that is used to compare to the first data series.
[direction]	<p>Optional: The direction of the cross over.</p> <ul style="list-style-type: none"> Use a positive [direction] number to look for series1 goes from below series2 to above series2. Use a negative [direction] number to look for when series1 goes from above series2 to below series2. <p>By default, function use a 1 positive value.</p>
[crossOverSize]	<p>The number of bars to consider when looking for the cross.</p> <p>The default value is 1 bar, which will check if the cross occurred between the last bar and the current bar.</p>
[offset]	<p>The number of bars to offset back before looking for the cross over.</p> <p>The default is to look at the current bar.</p>

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
```

Returns:

TRUE, if there was a cross over, **FALSE**, if not.

Links:**See Also:**

[Series Functions](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 241

Data Series Indexing

Series can be Auto Indexed or indexed manually.

An **Auto-Indexed** series is sized and indexed to the test or instrument. A manually indexed series must have its size declared in the series creation dialog. It can then be managed and maintained when more or less series elements are required. All series elements are initialized to the value given in the series creation dialog. Manually-Indexed series can be cleared and re-sized using script to adjust and clear the series by calling a series function.

Accessing a **Manually-Index** series element value requires the script to determine the index value placed in the "[]" braces. Value placed in those brackets is the absolute element location reference.

Auto-Indexed series automatically have their size set according to the number of bars contain in the instrument data file or number of test dates in the test.

Block Permanent Series (BPV) Auto-Index aligned to `test.currentDay`.

Instrument Permanent Series (IPV) Auto-Index aligned to `instrument.bar`.

This difference is because some markets/instruments don't trade on certain days because of exchange-specific holidays.

Example of an IPV Auto-Indexed series:

Day 1:

```
myIPVSeries = 10           ' Sets the value of 10 for the
series on day 1
```

Day 2:

```
myIPVSeries = 20           ' Sets the value of 20 for the
series on day 2
```

Day 3:

```
Print myIPVSeries           ' Prints the default value as set
in the variable editor, likely 0.
Print myIPVSeries[1]        ' Prints 20.
Print myIPVSeries[2]        ' Prints 10.
```

Series Indexing Notes:

Dialog example for creating a Manually-Index series.

Manually-Index series are **not Auto-Indexed** using either the `test.currentDay` (BPV) or `instrument.bar` (IPV) properties:

Series Indexing Notes:

Instrument Permanent Variable Series Indexing

When you do not enable the **Auto-Index** option each element is accessed by a script calculated index value.

Manually-Index series are adjustable in size. This is different than **Auto-Index** series. **Auto-Index** series are sized to `test.currentDay` (BPV) number of test bars, and (IPV) series are sized to the `instrument.bar` record count of the instrument.

This means you need to set each value using the `[]` braces. Retrieve a value will require you to use the same manually set index.

You also need to manage the series size yourself since Trading Blox won't know how many elements the system are required in the series.

Errors will be generated when the non auto indexed array is used:

- Without an index `[]`
- When the index is less than 1 or greater than the number of Manually-Index series defined elements.

Example of an Manually-Indexed series:

Series 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
```

Links:

[GetSeriesSize](#), [SetSeriesColorStyle](#), [SetSeriesSize](#), [SetSeriesValues](#)

See Also:

[Series Functions](#)

DownloadWebFile

DownloadWebFile can use a local folder parameter path or a file parameter path

Syntax:
<code>DownloadWebFile(fullPath)</code>

Parameter:	Description:
<code>fullFilePath</code>	

Returns:

Example:

Links:

See Also:
<code>FTPDownloadFile()</code>

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:	Description:
seriesName	Name of manually sized IPV & BPV series arrays. Manually means the Auto-Index option is disabled.
returns	Assign the number of elements in the named 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
```

Returns:

```
seriesSize = , 34
```

Links:

[SetSeriesSize](#), [SetSeriesValues](#)

See Also:

[Series Functions](#)

Highest

Finds the highest value of the series.

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:

```
Highest( series, bars, [offset] )
```

Parameter:	Description:
series	The name of the series.
bars	The number of bars over which to find the value.
[offset]	The number of bars to offset back before finding the 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 )
```

Returns:

Highest value in the series over the number of bars specified

Links:

[HighestBar](#), [Lowest](#), [LowestBar](#)

See Also:

[Series Functions](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 367

HighestBar

Finds the highest value of the series, then return the bars back.

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](#).

Syntax:

```
HighestBar( series, bars, [offset] )
```

Parameter:	Description:
series	The name of the series.
bars	The number of bars over which to find the highest bar value.
[offset]	The number of bars to offset back before finding the value.

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 ]
```

Returns:

Bar number of the Highest value in the series over the number of bars specified

Links:

[Highest](#), [Lowest](#), [LowestBar](#)

See Also:

[Series Functions](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 368

Lowest

Finds the lowest value of the series.

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:

```
Lowest( series, bars, [offset] )
```

Parameter:	Description:
series	The name of the series.
bars	The number of bars over which to find the lowest value.
[offset]	The number of bars to offset back before finding 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 )
```

Returns:

Lowest value in the range of bars in the series.

Links:

[Highest](#), [HighestBar](#), [LowestBar](#)

See Also:

Links:

[Series Functions](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 410

LowestBar

Finds the lowest value of the series, then it returns the bars back number.

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](#).

Syntax:

```
LowestBar( series, bars, [offset] )
```

Parameter:	Description:
series	The name of the series.
bars	The number of bars over which to find the value.
[offset]	Optional -- The number of bars to offset before finding the value.

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 ]
```

Returns:

Bar number of the bar with the lowest value in the series.

Links:

[Highest HighestBar](#), [Lowest](#)

Links:

See Also:

[Series Functions](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 411

LowestBar2

Series function **LowestBar2** function returns the bar index of the lowest value of a particular column in a dual array.

Syntax:

```
LowestBar2(series, bars, [offset])
```

Parameter:	Description:
series	
bars	
offset	

Example:**Returns:****Links:****See Also:**

MaxSynchBars

This function returns the maximum available date synched bars in the two series. It was designed to be used with the [CorrelationSynch](#) and [CorrelationLogSynch](#) functions to determine a minimum threshold value, return 0 or some other action.

This example shows the use of the common auto indexed series.

Syntax:

```
MaxSynchBars( series1, series2 )
```

Parameter:	Description:
series1	The name of the first series.
series2	The name of the second series

Example:

```
' Possible Use Example
maxAvailableSynchBars = MaxSynchBars( instrument.close,
instrument2.close )
' When Correlation comparison lenght is less than the max available
records,...
IF maxAvailableSynchBars > correlationPeriod THEN
  ' Use the correlation period parameter length
  correlationValue = CorrelationSynch( instrument.close,
instrument2.close, correlationPeriod )
ELSE
  ' Use the maxAvailableSynchBars period length
  correlationValue = CorrelationSynch( instrument.close,
instrument2.close, maxAvailableSynchBars )
ENDIF
```

Returns:

Returns the maximum number of bars from the current series location.

Links:

[CorrelationSynch](#), [CorrelationLogSynch](#)

See Also:

[Series Functions](#)

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.

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:

```
Median( series, [bars], [offset] )
```

Parameter:	Description:
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.

Example:

```
' Find the median value of the last 100 bars of the series.
medianValue = Median( instrument.close, 100 )
```

Returns:

The median value of series calculated at the current record location.

Links:

[Average](#)

See Also:

[Series Functions](#)

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 )
```

Parameter:

series

Description:

Then name of the series.

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.

' This statement retrieves the Slope of the Close
slope = RegressionSlope( instrument.close, 10 )

' This Statment returns the Regression End value.
endPoint = RegressionEnd( instrument.close )

' Plot the linear regression on the bars that
' have been used to calculate it. This is a
' postdictive calculation.

' The variable plotRegression is declared as
' an Instrument Permanent Auto Index Series
' Variable with plotting.

' Plot the Regression for the last 10 values
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.
```

Returns:

Example:

This example shows the End point value of the Regression Slope of an example auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#)..

Links:

[RegressionSlope](#)

See Also:

[Series Functions](#)

RegressionSlope

Finds the linear regression slope of the series.

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:
<code>RegressionSlope(series, bars, [offset])</code>

Parameter:	Description:
<code>series</code>	Then name of the series.
<code>bars</code>	The number of bars over which to find the value.
<code>[offset]</code>	Then number of bars to offset before finding the value.

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.

' This statement retrieves the Slope of the Close
slope = RegressionSlope( instrument.close, 10 )

' This Statment returns the Regression End value.
endPoint = RegressionEnd( instrument.close )

' Plot the linear regression on the bars that
' have been used to calculate it. This is a
' postdictive calculation.

' The variable plotRegression is declared as
' an Instrument Permanent Auto Index Series
' Variable with plotting.

' Plot the Regression for the last 10 values
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.
```

Returns:

This example shows the Regression Slope value of an example auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

Links:

[RegressionEnd](#)

See Also:

[Series Functions](#)

RegressionValue

Finds the value of a linear regression of the series at any point using an offset.

Note:

This function must be used in conjunction with [RegressionSlope](#) because it merely returns the slope and endpoint already calculated by [RegressionSlope](#). This means 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:
<code>RegressionValue(series, offset)</code>

Parameter:	Description:
series	The name of the series.
offset	The number of bars back.

Example:
<pre> ' 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 it. This ' is postdictive calculation result. ' ' The variable plotRegression is declared ' as an Instrument Permanent Auto Index ' Series Variable with plotting. For i = 0 TO 9 plotRegression[i] = RegressionValue(instrument.close, i) Next </pre>
Returns:
Regression Slope value at the offset bar location.

Links:
RegressionSlope

Links:**See Also:**[Series Functions](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 494

RSI

Computes the Relative Strength Index (RSI) of a series

Syntax:

```
RSI( series, rsiBars, [elementCount], [offset] )
```

Parameter:	Description:
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.

Example:

```
' Calculate the RSI of this instrument's  
' Average Closing price.  
RSI_Result = RSI(AvgCloseSeries, CalcLength, 0)
```

Returns:

RSI Calculation Result is the calculated Relative Strength Index (RSI) value associated with the data record where the RSI calculation is performed.

Links:**See Also:**

[Basic Indicators](#), [Series Functions](#)

SetAllowParameterWrite

Syntax:	
Parameter:	Description:
Example:	
Returns:	
Links:	
See Also:	

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 335

SetAllowPostDictive

Syntax:	
Parameter:	Description:
Example:	
Returns:	

Links:

See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 355

SetAuxiliaryWindowText

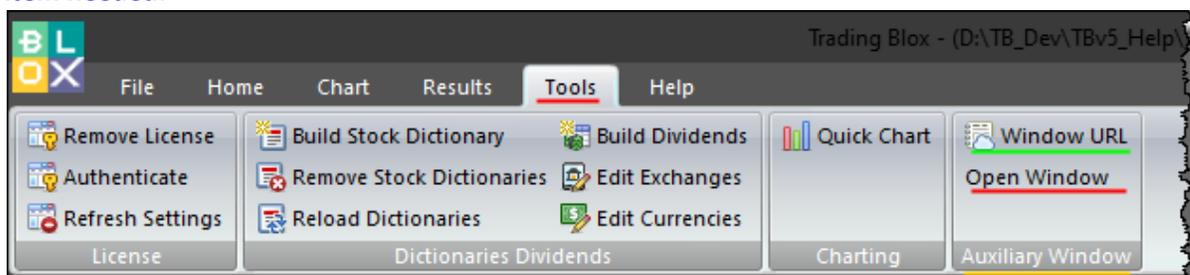
Trading Blox Builder has added a new dock-able **Auxiliary Window** that can be displayed or hidden. This new independent text display window can be fixed to the main screen, or it can float over Trading Blox Builder or outside of the Trading Blox Builder display area.

The Trading Blox Builder Log Window has the same features that allow it to float above or outside of Trading Blox Builder and both windows can be displayed vertically along the right side, or one over the other at the bottom of the main screen area.

Information sent to the Auxiliary Window can be sent to a "Cloud" destination for display. For now, when you use the **Auxiliary Window Window URL** option, **only send information to same IP for your personal security reasons.**

To Open the **Auxiliary Window** area, click image:

Main Menu's Tools Group => Click the Auxiliary Window Group => Click the Open Window menu item needed:



Tools Open Auxiliary Window Menu

When the **Open Window** button is used, the information sent to the **Auxiliary Window** can also be sent to a "Cloud" location when the "**Window URL**" button is selected.

A dockable or floating information display can provide flexibility because it can easily be moved around, change shape, and be used in tandem other information displays.

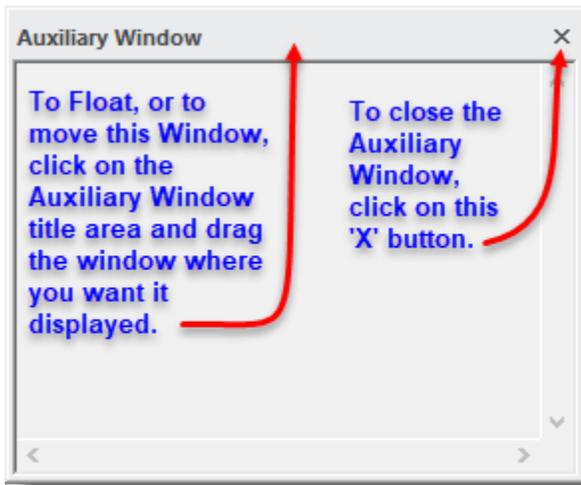
This optional Auxiliary Window text display can appear above or below the [Log Window](#).

The contents of this window can also be simulcast to the cloud using the:

Tools => Auxiliary Window => URL Window selection.

An **Auxiliary Window** can be displayed on the screen by clicking on the **Open Window** option shown above, or it can be opened and closed by using either one of these two function within a script section: [OpenAuxiliaryWindow](#), [CloseAuxiliaryWindow](#).

When an Auxiliary Window is open, it can also be moved away from the main Trading Blox Builder screen area by grabbing its title bar area with a right-side mouse click and then dragging it to where you want it displayed:



Open Auxiliary Window Move and Close Options.

A video in the Trading Blox Builder Forum: [Blox MarketPlace](#), shows how these options work : [Docked-Undocked-Floating Windows](#)

Syntax:

```
SetAuxiliaryWindowText( "windowText" )
```

Parameter:

"windowText"

Description:

"windowText" is a Non_Auto indexed BPV String array.

The contents of this Auxiliary Window are sent using the `SetAuxiliaryWindowText(windowText)` function, where "windowText" is a non-auto-indexed BPV String array.

How to Use:

There are many ways to use the display window. . This example send a new text information at the end of each year. In that test statement the process is kept simple so the process can be the focus. As each year of the test ends, the test will enter the After Test script section where the entire text series is sent to the Auxiliary Window text area.

Create the BPV:

This dialog creates the series "windowTextString" that will get a new text statement as each year of testing ends.

Block Permanent Variable [Block: windowTextStringSeries | Group: _Help]

Script Name

Display Name

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 [Create a Text String Series](#)
- Instrument - used to load and access alternate markets

Variable Options

Default Value

Scope

Reset Before Test [Block Scope should be enough.](#)

Auto Index [Series Must have a Manual Index](#)

Number of items in Series [Rows of Text Should be OK.](#)

BPV Dialog String-Series Settings

Once you've named your string series, the example script used the Block scope range. When everything is contained within the same blox, there is no need for a broader scope range. Because the script captures and sends text to the text string, which is contained within the same blox, it outputs information about the test, or for other uses.

Execute this function: [SetAuxiliaryWindowText](#) statement in the After Test script to emulate the example's process.

Example - Notes:

The example script section is created as an Auxiliary Blox. The output from that block is created when this Auxiliary blox is included in a system. In this example's output, the Dual Moving Average system installed when Trading Blox Builder is installed is used to generate trades on a random selection of EFT symbols that you should edit to limit the "Not Found" reminder for being different.

Script Use Details:

There are many ways to use the Auxiliary and Log window displays. This section provides a code example that sends new text information at the end of each year to a text-series

Example - Notes:

element row that increments each time text is created. In this test script the process is kept simple so the process capabilities can be the focus.

Create a BPV test-series:

This dialog creates the text-series "[windowTextString](#)." As each ends, the text message sent to the "[windowTextString](#)" reports the year that ended and the current change in available equity.

Block Permanent Variable [Block: windowTextStringSeries | Group: _Help]

Script Name:

Display Name:

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 [Create a Text String Series](#)
- Instrument - used to load and access alternate markets

Variable Options

Default Value:

Scope:

Reset Before Test [Block Scope should be enough.](#)

Auto Index [Series Must have a Manual Index](#)

Number of items in Series: [Rows of Text Should be OK.](#)

BPV Dialog String-Series Settings

Once you've named your string series, the example script used the Block's scope range. When everything is contained within the same blox, there is no need for a broader scope range. In this script example the process captures information and that is transformed to a text statement so it can be sent to a row in the newly created text series created earlier.

Example - Script:

Example - Script:

Execute this function: [SetAuxiliaryWindowText](#) statement in the After Test script to emulate the example's process.

This Example Script is contained in the Suite Link Below:

[SetAuxiliaryWindowText Suite with Blox](#)

```
' =====  
blockname: SetAuxiliaryWindowText - blox  
' =====  
' block Permanent VARIABLES
```

Example - Script:

```

' These variables are displayed so you'll understand more.
' -----
windowTextStringSeries [ Sample String Series ]; Series String; 0.000000;
block
iYearNdx [ Cerate a Msg For each test date Year change ]; Integer; 0;
block
iAuxWindowInit [ ]; Integer; 0; block
iThisYear [ Year being tested. ]; Integer; 0; block
sMsg [ ]; String; ; block
iTestEndDate [ Last test Date YYYYMMDD ]; Integer; 0; block
iTestStartDate [ Human readable name goes here... ]; Integer; 0; block
iPreviousYear [ ]; Integer; 0; block
iAnswer [ ]; Integer; 0; block
iTestYearsCount [ Human readable name goes here... ]; Integer; 0; block
lastYearEqty [ ]; Floating; 0.000000; block
ThisYearEqty [ ]; Floating; 0.000000; block
' -----
===== SCRIPT_CHANGE:
' =====
' SetAuxiliaryWindowText
' BEFORE TEST SCRIPT - START
' =====
' ~~~~~
' Initialize Auxiliary Window Init Status
iAuxWindowInit = FALSE
' Initialize Year Value
iThisYear = Year(test.currentDate)
' Start Test Date
iTestStartDate = test.testStart
' Last Test Date
iTestEndDate = test.testEnd
' Get the Number of years in the Test.
iTestYearsCount = Year(iTestEndDate) - Year(iTestStartDate) + 1
' Get Starting Equity
lastYearEqty = test.totalEquity
ThisYearEqty = test.totalEquity
' -----
' Display the Setup Details
PRINT
PRINT "Test Date Start          ", test.currentDate
PRINT "Script Section            ", block.scriptName
PRINT "iAuxWindowInit              = ", iAuxWindowInit
PRINT "iThisYear                    = ", iThisYear
PRINT "iTestStartDate               = ", iTestStartDate
PRINT "iTestEndDate                 = ", iTestEndDate
PRINT
PRINT "Year(iTestStartDate) = ", Year(iTestStartDate)
PRINT "Year(iTestEndDate)   = ", Year(iTestEndDate)
PRINT "iTestYearsCount      = ", iTestYearsCount
PRINT "ThisYearEqty         = ", AsString(ThisYearEqty, 2)
PRINT "lastYearEqty         = ", AsString(lastYearEqty, 2)
PRINT "Trading Equity       = ", AsString(test.totalEquity, 2)
PRINT
PRINT "ThisYearEqty         = ", ThisYearEqty

```

Example - Script:

```

PRINT "lastYearEqty          = ", AsString(lastYearEqty , 2)
PRINT
' ~~~~~
' =====
' BEFORE TEST SCRIPT - END
' SetAuxiliaryWindowText
' =====
===== SCRIPT_CHANGE:
' =====
' SetAuxiliaryWindowText
' BEFORE TRADING DAY SCRIPT - START
' =====
' ~~~~~
' Initialize Year Value
iThisYear = Year(test.currentDate)
' Calculate This Year equity change
ThisYearEqty = test.totalEquity - lastYearEqty
' -----
' When the Year Changes, display change
If iThisYear > iPreviousYear THEN
' Show Year Change
PRINT
PRINT "This Test Date          ", test.currentDate
PRINT "Script Section          ", block.scriptName
PRINT "iAuxWindowInit           = ", iAuxWindowInit
PRINT "iThisYear                 = ", iThisYear
PRINT
PRINT "ThisYearEqty              = ", AsString(ThisYearEqty, 2)
PRINT "lastYearEqty              = ", AsString(lastYearEqty , 2)
PRINT "Available Equity          = ", AsString(test.totalEquity, 2)
PRINT
ENDIF ' iThisYear > iPreviousYear
' ~~~~~
' =====
' BEFORE TRADING DAY SCRIPT - END
' SetAuxiliaryWindowText
' =====
===== SCRIPT_CHANGE:
' =====
' SetAuxiliaryWindowText
' AFTER TRADING DAY SCRIPT - START
' =====
' ~~~~~
' Compare Current Year to Previous Year
If iThisYear > iPreviousYear THEN
' Calculate This Year equity change
ThisYearEqty = test.totalEquity - lastYearEqty

' If this is the first time, initialize
' iYearIndex & Update the Init-Flag
If iAuxWindowInit = FALSE THEN
' Update Annual Auxiliary Message
iYearNdx = 1
' Show Init is complete

```

Example - Script:

```

        iAuxWindowInit = TRUE
    ELSE
        ' Increment Year Index
        iYearNdx = iYearNdx + 1
    ENDIF
' -----

PRINT
PRINT "This test Date      ", test.currentDate
PRINT "Script Section      ", block.scriptName
PRINT "iAuxWindowInit      = ", iAuxWindowInit
PRINT "iPreviousYear       = ", iPreviousYear
PRINT "iThisYear            = ", iThisYear
PRINT "iYearNdx             = ", iYearNdx
PRINT
PRINT "ThisYearEqty        = ", AsString(ThisYearEqty, 2)
PRINT "lastYearEqty        = ", AsString(lastYearEqty, 2)
PRINT "Available Equity    = ", AsString(test.totalEquity, 2)
PRINT
PRINT "sMsg                  = ", "Year Changed"
' -----
' Update Previous Year
iPreviousYear = iThisYear
' Update This Year's Equity
lastYearEqty = ThisYearEqty
' -----
' This is anything goes text
sMsg = "For Year-End " + AsString(iThisYear, 0)_
      + ", Total Equity = " + AsString(ThisYearEqty, 2)
' Update StringSeries Next Element
windowTextStringSeries[iYearNdx] = sMsg
ENDIF
' ~~~~~
' =====
' AFTER TRADING DAY SCRIPT - END
' SetAuxiliaryWindowText
' =====
===== SCRIPT_CHANGE:
' =====
' SetAuxiliaryWindowText
' AFTER TEST SCRIPT - START
' =====
' ~~~~~

PRINT "Open Auxiliary Window"
' Open the Auxiliary Window
OpenAuxiliaryWindow
' -----

PRINT
PRINT "Test Date Start      ", test.currentDate
PRINT "Script Section      ", block.scriptName
PRINT "iAuxWindowInit      = ", iAuxWindowInit
PRINT "iThisYear            = ", iThisYear
PRINT "iTestStartDate       = ", iTestStartDate
PRINT "iTestEndDate         = ", iTestEndDate
PRINT

```

Example - Script:

```

PRINT "ThisYearEqty      = ", AsString(ThisYearEqty, 2)
PRINT "lastYearEqty     = ", AsString(lastYearEqty , 2)
PRINT "Test End Equity  = ", AsString(test.totalEquity, 2)
PRINT
PRINT "SetAuxiliaryWindowText( windowTextStringSeries )"
PRINT
' -----
' Send the windowTextStringSeries Records to
' the Auxiliary Window display
SetAuxiliaryWindowText( windowTextStringSeries )
' ~~~~~
' =====
' AFTER TEST SCRIPT - END
' SetAuxiliaryWindowText
' =====
===== SCRIPT_CHANGE:
' =====
' SetAuxiliaryWindowText
' AFTER SIMULATION SCRIPT - END
' =====
' ~~~~~

PRINT
PRINT "Test Date Start      ", test.currentDate
PRINT "Script Section       ", block.scriptName
PRINT "Send Aux. Window Open Close Msg."
PRINT
' -----
' Message Box Display Test
sMsg = "OK to Leave Auxiliary Window Open -- Cancel to Close"
' Generate the Message Display
' iAnswer = MessageBox( sMsg , 1, 32 )
' If Cancel is used, Close the Auxiliary Window
If iAnswer = 2 THEN
' Close the Auxiliary Window
CloseAuxiliaryWindow
ENDIF
' ~~~~~
' =====
' AFTER TEST SCRIPT - END
' SetAuxiliaryWindowText
' =====
===== SCRIPT_CHANGE:

```

Returns:

```

Log Window
-----
This test Date      20200101
Script Section     After Trading Day
iAuxWindowInit     = 1
iPreviousYear      = 2019
iThisYear          = 2020
iYearNdx           = 11

ThisYearEqty      = 392822.60
lastYearEqty      = 716652.88
Available Equity   = 1109475.48

sMsg               = Year Changed
Open Auxiliary Window

Test Date Start    20201002
Script Section     After Test
iAuxWindowInit     = 1
iThisYear          = 2020
iTestStartDate     = 20100104
iTestEndDate       = 20201002

ThisYearEqty      = 521406.91
lastYearEqty      = 392822.60
Test End Equity    = 915668.35

SetAuxiliaryWindowText( windowTextStringSeries )

Test Date Start    20201002
Script Section     After Simulation
Send Aux. Window Open Close Msg.

Simulation: 17 seconds.
Ending Test Run at 10:51:31
Overall: 19 seconds.

```

```

Auxiliary Window
-----
For Year-End 2010, Total Equity = 0.00
For Year-End 2011, Total Equity = 521403.65
For Year-End 2012, Total Equity = 2992.27
For Year-End 2013, Total Equity = 548764.23
For Year-End 2014, Total Equity = 250285.09
For Year-End 2015, Total Equity = 787439.06
For Year-End 2016, Total Equity = 254549.25
For Year-End 2017, Total Equity = 815944.78
For Year-End 2018, Total Equity = 371645.07
For Year-End 2019, Total Equity = 716652.88
For Year-End 2020, Total Equity = 392822.60

```

Log & Auxiliary Undocked Window Output Examples

Links:

[OpenAuxiliaryWindow](#), [CloseAuxiliaryWindow](#),

See Also:

[Auxiliary Window](#), [Log Window](#)

SetIBTWSDataTimeZone**Syntax:**

--

Parameter:**Description:**

--	--

Example:

--

Returns:

--

Links:

--

See Also:

--

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 636

SetSeriesAutoIndex

This function can change an **IPV** Auto-Index series into a Manual-Index series.

Syntax:	
<code>instrument.SetSeriesAutoIndex(series, TRUE/FALSE)</code>	

Parameter:	Description:
<code>series</code>	IPV series name where the indexing method is to be change.
TRUE /FALSE	TRUE , will change a manual indexing series to Auto-Indexing. A FALSE value will change an Auto-Indexing series to Manual Indexing.

Example:
Returns:

Links:
See Also:
Series Functions

SetSeriesColorStyle**Requirement for Use:**

- Data series must be an **IPV** that is set to "**Auto-Index**".
- This function will also color Indicators created using Trading Blox built-in Indicator Creation Dialog.
- Each element of an IPV Indicator series can be colored.
- IPV series indicator can only be plotted on the instrument chart display.
- Indicator chart displays using this function override the indicator's dialog plot type selection

Syntax:

```
SetSeriesColorStyle(series, plotColor, [plotStyle], [offset] )
```

Parameter:	Description:
series	The name of series.
plotColor	Element Color to use with named "series"

Parameter:	Description:	
	<p>Preference Color Setup:</p> <div data-bbox="444 638 1211 1709" style="border: 1px solid green; padding: 10px;"> <p>Trade Chart Colors</p> <p>General Chart Colors</p> <p>Grid Color <input type="color" value="#ccccff"/></p> <p>Chart Background Color <input type="color" value="#ffffcc"/></p> <p>Chart CrossHair Color <input type="color" value="#cccccc"/></p> <hr/> <p>Trade Marker Colors</p> <p>Long Trade Color <input type="color" value="#008000"/></p> <p>Short Trade Color <input type="color" value="#ff0000"/></p> <p>Entry Price Color <input type="color" value="#add8e6"/></p> <p>Stop Price Color <input type="color" value="#e6b8ff"/></p> <p>Exit Price Color <input type="color" value="#ccccff"/></p> <hr/> <p>Bar and Candlestick Colors</p> <p>Up Bar Color <input type="color" value="#008000"/></p> <p>Down Bar Color <input type="color" value="#ff0000"/></p> <p>Up Candle Color <input type="color" value="#008000"/></p> <p>Down Candle Color <input type="color" value="#ff0000"/></p> <hr/> <p>Custom Colors</p> <p>Custom Color 1 <input type="color" value="#000000"/> <input type="text" value="0"/></p> <p>Custom Color 2 <input type="color" value="#800000"/> <input type="text" value="139"/></p> <p>Custom Color 3 <input type="color" value="#0000ff"/> <input type="text" value="16711680"/></p> <p>Custom Color 4 <input type="color" value="#ffffff"/> <input type="text" value="16777215"/></p> </div> <p style="text-align: center;">User Preference Color Table</p>	<p>Color Constant Name:</p> <p>COLOR_GRID</p> <p>COLOR_BACKGROUND</p> <p>COLOR_CROSSHAIR</p>

Parameter:	Description:	
	<p>Preference Color Setup:</p>	<p>Color Constant Name:</p>
		<p>C O L O R L O N G H I G H R A P H I C C O L O R S H O R T H I G H R A P H I C</p>

Parameter:	Description:	
	<p>Preference Color Setup:</p>	<p>Color Constant Name:</p>
		<p>COLORHEADENTRY COLORHEADSETUP</p>

Parameter:	Description:	
	<p>Preference Color Setup:</p>	<p>Color Constant Name:</p>
		<p>COLORHEADEXIT</p> <p>COLORUPBAR</p> <p>COLORDOWNBAR</p>

Parameter:	Description:	
	<p>Preference Color Setup:</p>	<p>Color Constant Name:</p>
		<p>ColorUpHandle</p>
		<p>ColorDownHandle</p>

Parameter:	Description:	
	<p>Preference Color Setup:</p>	<p>Color Constant Name:</p>
		<p>COLORCONSTANT1</p>
		<p>COLORCONSTANT2</p>
		<p>COLORCONSTANT3</p>

Parameter:	Description:																			
	<p>Preference Color Setup:</p>	<p>Color Constant Name:</p> <p>C O L O R C O N S T A N T N A M E</p>																		
[plotStyle]	<p>Note:</p> <p>If you use one of the custom colors from Preferences' Trade Chart Colors section, the user will be able to modify the colors as desired by change the color associated with a color name.</p> <p>The Color picker displays the number for each custom color. This number can be used to set a fixed color, or you can create or use a color number like 8388736 which is violet.</p> <p>More information about colors is available here: Colors.</p> <table border="1" data-bbox="427 1413 865 1917"> <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> </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
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																			

Parameter:	Description:							
	<table border="1"> <thead> <tr> <th>Style Value:</th> <th>Style Type:</th> </tr> </thead> <tbody> <tr> <td>9</td> <td>Area</td> </tr> <tr> <td>10</td> <td>Staircase</td> </tr> </tbody> </table>	Style Value:	Style Type:	9	Area	10	Staircase	
Style Value:	Style Type:							
9	Area							
10	Staircase							
	Note: Plot Style number used overrides indicator's dialog plot selection.							
[offset]	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 - 1:

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 THEN
  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
' ~~~~~

```

- 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.

Example - 2:

Example - 1:

```

' ~~~~~
' 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 - 3:

```

' ~~~~~
' Color of the up bar AND a thin line:
SetSeriesColorStyle( averageTrueRange, ColorUpBar, 1 )
' ~~~~~

```

Example - 4:

```

' ~~~~~
' 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:

Display Result:

Example - 1:



Links:

[Colors](#), [ColorRGB](#)

See Also:

[Prefernces](#), [Series Functions](#)

SetSeriesEnable

This function can Enable or Disable any **IPV** Series, or Regular Indicator in the test. It will not work with any script calculated indicators, but it will work with indicators that are created using the Indicator Dialog that creates a selected indicator. When this is used, it Enables or Disables the series. To change its Enable or Disable state, True will enable and False will disable. Changing the state doesn't recompute the series. It works this way so the changes are processed before all the indicators are computed in the [Before Test](#) script section.

To use this function, it must be executed in the [SetParameters](#) script section.

Syntax:	
<code>instrument.SetSeriesEnable (series, TRUE/FALSE)</code>	

Parameter:	Description:
<code>series</code>	Any IPV , BPV or indicator series name in the test.
TRUE /FALSE	TRUE will enable a series or indicator. FALSE will disable a series or indicator.

Example:	
SetParameters Methods	

Returns:	
None	

Links:	
SetParameters , SetParameters Methods , DisableTrading ,	

See Also:	
Series Functions	

SetSeriesSize

This method will set the series size of a manually sized and manually indexed series of floating numbers or strings.

Two Column Number Series:

- 1) Selecting either the Number or String data-type option available in the IPV and BPV variable creation dialog.
- 2) Disable the default "**Auto-Index -- Uses [n] as Lookback from Current Day**"
- 3) When the "**Auto-Index**" option is disabled, the series will be a "**Manual Index**" series with 100 rows.
- 4) If you know the number of rows will be different, you can change them now, or change is scripting using this topic's function.

- All Manual-Indexed series have a minimum size of 1, and a maximum size of 1,000,000 elements. Do not attempt to access a manual series element location with an index value less than 1, or greater than the actual size of the series.
- Series index number violations will errors that will stop execution.
- Manual series created with the default or any other value can be re-sized using this function.
- Size of a series can be obtained by using the [GetSeriesSize](#) function.
- This function can be used to increase or decrease the size of a manually Indexed series.
- When a manual series size is changed, the values currently in the series row location are retained.
- When rows are removed, the values in the removed rows will be deleted.
- When the size is increased, the new element rows are initialize with the default value shown in series creation dialog.

Two-Column Series Note:

A manually indexed single column number series can have two columns. To add a column to one column series, use the SetSeriesSize [optional] parameter (See "Add Column Example").

Syntax:

```
SetSeriesSize( SeriesName, column_1_Size, [column_2_Size] )
```

Parameter:	Description:
SeriesName	Adjust size series name.
column_1_Size	New series size integer value.
[column_2_Size]	Optional second series integer size.

Example - One Column Series:

```

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

```

Return - One Column Series:

SetSeriesSize doesn't return any value, but the size of a series can be obtained using the GetSeriesSize function.

Example - Add Column Example:

```

' -----
' Convert a One-Column Number Series to a Two-Column Series
' -----
VARIABLES: iSeriesSize, iColumn_1_Size, iColumn_2_Size Type: Integer
' -----
' Help File Function Statement
'   SetSeriesSize( SeriesName, column_1_Size, [column_2_Size] )
' -----
' Reset Working Variables
iSeriesSize = 0
iColumn_1_Size = 10
iColumn_2_Size = 10
' -----
' Discover Current Size of a Manual Series
iSeriesSize = GetSeriesSize( aNumeric2ColSeries )
' Report Manual Series Size
PRINT "Original iSeriesSize = ", iSeriesSize
' -----
PRINT
PRINT "Change a One-Column Series to a Two-Column Series"
' Add a second column to the Manual Number Series
SetSeriesSize( aNumeric2ColSeries, iColumn_1_Size, iColumn_2_Size )
' -----
' Discover Current Size of a Manual Series
iSeriesSize = GetSeriesSize( aNumeric2ColSeries )
' Report Manual Series Size
PRINT "After iSeriesSize = ", iSeriesSize
' -----
' BEFORE TEST - END
' -----

```

Return - Add Column Example:

Example - One Column Series:

Original iSeriesSize = 100

Change a One-Column Series to a Two-Column Series

After iSeriesSize = 10

Links:

[GetSeriesSize](#), [SetSeriesValue](#)

See Also:

[Series Functions](#)

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	The name of series to be changed.
seedValue	The replacement value to use for the series, or for the range of elements designated between the Start and End element number range.
[Start-Element-Num]	The earliest, or lowest series index value from which seed-value will begin replacing the values in the series.
[End-Element-Num]	The 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
  ' Clear Rank Adjustments
  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
' ~~~~~

```

Returns:

```

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

```

Example:

```
TestSeries[3] = 0.000000000  
TestSeries[4] = 0.000000000  
TestSeries[5] = 0.000000000
```

Links:

[GetSeriesSize](#), [SetSeriesSize](#)

See Also:

[Series Functions](#)

SetSeriesValueType

This function will change the assigned price item in a Trading Blox created Indicator dialog. To change the assigned price, this function must be executed in the [SetParameters](#) script section.

Syntax:
<code>SetSeriesValueType(series-name, "Price-Column")</code>

Parameter:	Description:
series-name	Series name is the name of the indicator, or IPV data series where the column name of the data contains more than one data series. For example, the WeightedAverage selected for this indicator is probably using the Close of an instrument's Price-Bar.
"Price-Name"	One of the four text price-names that can be used to change the price record value in an indicator series are: " Open, High, Low, Close " Be sure to place Quotation marks around the price name.

Example:
<p>This example needs to be used in the SetParameter script section.</p> <pre>' In this example, the WeightedAverage indicator is using ' the close of the price-bar to determine a Weighted-Average result. ' The user wants to see how the High of the price-bar would ' effect how the system responds. SetSeriesValueType(WeightedAverage, "High")</pre>

Returns:
After the function executes, the Weighted-Average of the " High " price, instead of the previous "Close" price will be the values available in all of the element locations in this indicator.

Links:
Creating Indicators , SetParameters , SetParameters Methods
See Also:

SortSeries

Sorts the series in ascending and descending order. Will sort a limited part of the series any where in the series.

Can only be use only with non-auto indexed series.

Auto Indexed series must always be kept align with their indexing method.

- BPV Auto-Indexed series are aligned with test.currentDay.
- IPV Auto-Indexed series are aligned with instrument.bar.

Syntax:

```
SortSeries( seriesName, [element count], [offset], [direction] )
```

Parameter:	Description:
seriesName	The series name to be sorted.
[element count]	Optional function value. When omitted, the default is to sort all the elements in the series unless a specified number of elements is entered into this field. When a value is entered, that value will specify how many elements to sort.
[offset]	Optional function value. When omitted, the default is to sort all the elements in the series. When a value is entered into the optional "element count" field, the value entered in this field will be the starting index from which element count will use to start the optional "element count" number of elements to sort. This means when the intent is to sort the entire series without any exclusions the value entered into this area should be the same as "element count" value.
[direction]	Optional function value. The default, or no entry in this field, is to sort the elements in an ascending order. Note: 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
' ~~~~~

```

Return - 1:

```

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

```

Example - 1:

```
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](#)

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:	
<pre>SortSeriesDual(series1, series2, [element count], [offset], [direction])</pre>	

Parameter:	Description:
series1	The series name that will be sorted based upon the values in series2.
series2	The series name that contains the values sorting will use to determine the sort
[element count]	Optional value. Function default sorting action is to sort all the elements in the series. When a value is entered, that value determines how many elements in the series are sorted.
[offset]	Optional value. The starting index from which element count goes back for non auto indexed series. Default value is zero so the function will sort the [element count] specified elements.
[direction]	When parameter is not used, the default sort is in ascending order. Note: Positive value sorts the elements in ascending order. Negative value sorts the elements in descending order.

Example:
<pre>' Sort resultsIndex based on the values of the results series. SortSeriesDual(resultsIndex, results, elementsToSort, elementsToSort, - 1) ' Sort the series element in descending order. SortSeries(results, elementsToSort, elementsToSort, -1)</pre> <p>This is an example of how a dual series numeric series can be used:</p>

Example:

```

' =====
' _SortDualSeries_Example
' BEFORE TEST SCRIPT - START
' =====
' ~~~~~
' Create Character Spaces for Symbol Padding
iTxtLen = 5
sPad = ""

For x = 1 TO iTxtLen STEP 1
    sPad = sPad + " "
Next ' x
' ~~~~~
' Determine How Many Symbols to Create
SymbolCount = system.totalInstruments

' Size Series To Symbol Count
SetSeriesSize(Series1, SymbolCount)
SetSeriesSize(Series2, SymbolCount)
SetSeriesSize(Series3, SymbolCount)
SetSeriesSize(Series4, SymbolCount)

PRINT
PRINT "Random Seed Value = ", RandomSeed(1)
PRINT
PRINT "-----"
PRINT "GENERATE RANDOM NUMBERS"
PRINT "-----"
PRINT
' ~~~~~
' Build Sample Dual Data Series
For x = 1 TO SymbolCount STEP 1
    ' Load Each Symbol in the Portfolio
    Load_OK = Mkt.LoadSymbol( x )

    ' Assign Values to Each Series
    Series1[x] = x
    Series2[x] = Random( 0, 30000 ) * 0.001

    ' Randomly Convert To Negative Value
    If Random(1, 10) < 5 THEN Series2[x] = Series2[x] * -1
    ' Duplicate Series2 Values in Series3
    Series3[x] = Series2[x]

    ' Provide Character Third Column Sort
    sTxt1 = Mkt.symbol
    sTxt1 = sTxt1 + Left(sPad, iTxtLen - Len(sTxt1))
    snum = ""

    For y = 1 TO Len(sTxt1) STEP 1
        snum = snum + ASCII(mid(sTxt1, y, 1))
    Next ' y

```

Example:

```

Series4[x] = AsFloating(snum)

' Show Assembled Random Character Symbol
PRINT AsString(x,0), " - ", _
      Series1[x], " - ", _
      Series2[x], " - ", _
      Series3[x], " - ", _
      Series4[x], " - ", _
      sTxt1
Next ' x
' ~~~~~

PRINT
PRINT "-----"
PRINT "SortSeriesDual"
PRINT "-----"
PRINT
' ~~~~~
' PERFORM DUAL SERIES SORTING STEPS
' Sort Series1 based on the values in Series2
SortSeriesDual( Series1, Series2, SymbolCount, SymbolCount, 1 )

' To Finish DualSort, sort the results of Series2 to Match
' the sorted index in Series1.
SortSeries( Series2, SymbolCount, SymbolCount, 1 )

' Sort Series4 based on the values in Series3
SortSeriesDual( Series4, Series3, SymbolCount, SymbolCount, 1 )

' To Finish DualSort, sort the results of Series4 to Match
' the sorted index in Series1.
SortSeries( Series3, SymbolCount, SymbolCount, 1 )
' ~~~~~
' SHOW SORTED RESULTS
' Display Column Titles
PRINT "Count", " - ", "GrpNdex", _
      " - ", "Data 1", _
      " - ", "Data 2"
' Display Sorted Series2 and the Aligned Series1 Index
For x = 1 TO SymbolCount STEP 1
  ' Get the series reference
  sNum = AsString(Series4[x], 0)

  Load_OK = Len(sNum)
  sTxt1 = ""

  For y = 1 TO Len(sNum) STEP 2
    sTxt1 = sTxt1 + mid(sNum, y, 2)
  Next ' y

  PRINT AsString(x,0), " - ", AsString(Series1[x], 0), _
        " - ", AsString(Series2[x], 3), _
        " - ", Series4[x], _
        " - ", sTxt1 )
Next ' x

```

Example:

```

| ~~~~~
| =====
| BEFORE TEST SCRIPT - END
| _SortDualSeries_Example
| =====

```

[SortSeriesDual Usage Example](#) The the link on the left to access the above example that is a posted in the **Trading Blox Marketplace** section.

Returns:

Sorted series based upon the values in the series 2.

Links:

[SortSeries](#)

See Also:

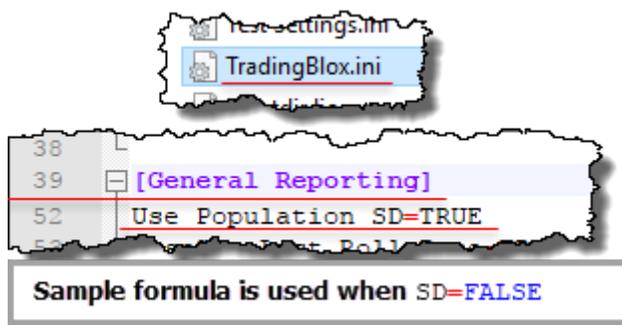
[Series Function](#)

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 572

StandardDeviation

By default, Trading Blox Builder Standard Deviation function uses Population formula instead of Sample formula to be more consistent with industry standards. To change the default to the Sample formula, access the Trading Blox Builder.ini file:



Trading Blox installation folder default setting

In terms of the Sample vs. Population Standard Deviation, TRUE enables Population formula, FALSE enables Sample formula.

This For information on using functions with non auto indexed series review [Series Functions](#).

Syntax:

```
x = StandardDeviation( series, bars, [offset] )
```

Parameter:	Description:
series	The name of the series.
bars	The number of bars over which to find the value.
[offset]	The optional number of bars to offset before finding the value.

Example:

```
' Example shows the use of the common auto indexed series.
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 )
```

Returns:

Standard Deviation value for each location in a series.

Links:

[Series Functions](#)

See Also:

[Math Functions](#)

StandardDeviationLog

Finds the standard deviation of the series. Uses the change in the log of the values.

Syntax:	
<code>StandardDeviationLog(series, bars, [offset])</code>	

Parameter:	Description:
<code>series</code>	The name of the series
<code>bars</code>	The number of bars over which to find the value
<code>[offset]</code>	The number of bars to offset before finding the 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 = StandardDeviationLog( instrument.close, 100 )

```

Returns:

The standard deviation of the series used over the period specified.

This example shows the use of a common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

Links:

Links:**See Also:**[Math Functions](#), [Series Functions](#)

Sum

Finds the sum of the series. This function will not sum or total a sequence of comma limited values or expressions. For a non-array values totaling function see [SumValues](#).

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:

```
Sum( series, bars, [offset] )
```

Parameter:	Description:
series	name of the series
bars	number of bars over which to find the value
offset	number of bars to offset before finding the value

Example:

```
' -----
' Auto-Indexed Series Example
' Example Shows Sum of Bars 11 to 20
' -----
' Find the sum of the 10 closes starting 20 bars ago
sumCloses = Sum( instrument.close, 10, 20 )

' -----
' Manually Index Series Example
' Example Shows Sum Values in Series
```

Example:

```

' for values shown.
' -----
' Set Series Size
iSize = 10
' Offset Determines Range
iOffset = 10
' Size Manual Index Series
SetSeriesSize(X, iSize )
' -----
' Loop Through Series
For i = 1 TO iSize STEP 1
    ' Assign Index value
    X[i] = i
Next ' i
' -----
' Sum Value of X[] Series
SumOfX = Sum(X, iSize, iOffset )
' -----
PRINT "Total Value of Series = ", SumOfX
' -----
' Sum Value of X[] Series
SumOfX = Sum(X, iSize-8, iOffset )
' -----
PRINT "Total Value of Series = ", SumOfX
' -----

```

Returns:

Function returns the Sum or Total of the series elements specified.

```

1 = 1.000000000 SumX = 1
2 = 2.000000000 SumX = 3
3 = 3.000000000 SumX = 6
4 = 4.000000000 SumX = 10
5 = 5.000000000 SumX = 15
6 = 6.000000000 SumX = 21
7 = 7.000000000 SumX = 28
8 = 8.000000000 SumX = 36
9 = 9.000000000 SumX = 45
10 = 10.000000000 SumX = 55
Total Value of Series = 55
Total Value of Series = 19

```

Links:**See Also:**

[Series Functions](#)

SwingHigh

Finds the swing high value. An example shows the use of a common auto indexed series. For information on using functions with non auto indexed series, review [Series Functions](#).

Syntax:
<code>SwingHigh(series, [occurrence], [forwardStrengthBars], [backwardStrengthBars], [lookbackBars], [offsetBars])</code>

Parameter:	Description:
<code>series</code>	Series name to search.
<code>[occurrence]</code>	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.
<code>[forwardStrengthBars]</code>	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.
<code>[backwardStrengthBars]</code>	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.
<code>[lookbackBars]</code>	Total number of look-back bars to check for a swing. Default is to use all available bars.
<code>[offsetBars]</code>	Number of bars to offset before finding the value. The default is 0, using the current bar.

Example:

```
' Checks the last 500 bars from the current bar location.
' Find the second occurrence back where the
' later 4 and prior 5 bar values were lower
' than the current value.
swingValue = SwingHigh( instrument.high, 2, 4, 5, 500 )
```

Returns:

Value returned is the Swing-High price when a Swing-High price is found. When a Swing-High price is not found, function returns a -1.

Links:

[SwingHighBars](#), [SwingLow](#), [SwingLowBars](#)

Links:**See Also :**[Series Functions](#)

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 595

SwingHighBars

Finds the swing high bar.

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:
<code>SwingHighBar(series, [occurrence], [forwardStrengthBars], [backwardStrengthBars], [lookbackBars], [offsetBars])</code>

Parameter:	Description:
<code>series</code>	Series name to use
<code>[occurrence]</code>	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.
<code>[forwardStrengthBars]</code>	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.
<code>[backwardStrengthBars]</code>	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.
<code>[lookbackBars]</code>	Total number of look-back bars to check for a swing. Default is to use all available bars.
<code>[offsetBars]</code>	Number of bars to offset before finding the value. The default is 0, using the current bar

Example:
<pre>' Checks the last 500 bars from the current bar location. ' Find the second occurrence back where the later 4 ' and prior 5 bar values were lower than the current value. 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.</pre>

Example:**Returns:**

Function value returns a positive integer number that represent the bar offset number where a Swing-High bar was found. When a Swing-High bar is not found, function returns a [-1](#).

The positive integer number represents the bar count back from the offset starting index of the swing bar. Current bar is zero, and a return of [1](#), indicates one bar back.

Links:

[SwingHigh](#), [SwingLow](#), [SwingLowBars](#)

See Also:

[Series Functions](#)

SwingLow

Finds the swing low value.

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:
<pre>swingValue = SwingLow(series, [occurrence], [forwardStrengthBars], [backwardStrengthBars], [lookbackBars], [offsetBars])</pre>

Parameter:	Description:
series	Series to search for Swing-Low.
occurrence	Swing-Low occurrence to locate. Default is to enter 1 for the most recent Swing-Low location. First occurrence of a Swing-Low return value is the count of bars back from the current bar's location. A value of 2 will locate the second occurrence back, etc.
forwardStrengthBars	Value entered determines how many bars forward to check to determine if a Swing-Low occurred. Default value is to check 1 bar. If all forwardStrengthBars are higher than the bar, then the Swing-Low occurred. If any are equal, then the Swing-Low did not occur. Flat bottoms are ignored.
backwardStrengthBars	Number of bars backward to check for a given bar to determine if a Swing-Low 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 bottoms are ignored.
lookbackBars	Number of look-back bars to offset before searching the data. Default is 0 to start the search using the current bar.

Parameter:	Description:
offset	Number of bars back from the offset value to start-value to search for a Swing-Low.

Example:

```
' Checks the last 500 bars.
' Find the second occurrence back
' where the later 4 and prior 5 bar
' values were higher than the current value.
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.
```

Returns:

Value returns the Low price found when Swing-Low price is found. When a Swing-Low price is not found, function returns a **-1**.

Links:

[SwingHigh](#), [SwingHighBars](#), [SwingLowBars](#)

See Also:

[Series Functions](#)

SwingLowBars

Finds the specified Swing-Low bar occurrence location. An example below shows the use of the common auto indexed series. For information on using functions with non auto indexed series review [Series Functions](#).

Syntax:

```
iBarIndex = SwingLowBar( series, [occurrence], [forwardStrengthBars],
[backwardStrengthBars], [lookbackBars], [offsetBars]
```

Parameter:	Description:
series	Name of data series to use for finding a Swing-Low bar.
occurrence	Swing-Low occurrence to locate. Default is to enter 1 for the most recent Swing-Low location. First occurrence of a Swing-Low return value is the count of bars back from the current bar's location. A value of 2 will locate the second occurrence back, etc.
forwardStrengthBars	Value entered determines how many bars forward to check to determine if a Swing-Low occurred. Default value is to check 1 bar. If all forwardStrengthBars are higher than the bar, then the Swing-Low occurred. If any are equal, then the Swing-Low did not occur. Flat bottoms are ignored.
backwardStrengthBars	Number of bars backward to check for a given bar to determine if a Swing-Low 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 bottoms are ignored.
lookbackBars	Number of bars to offset before searching the data. Default is 0 to start the search using the current bar.
offset	Number of bars back from the offset value to start-value to search for a Swing-Low bar.

Example:

```

' Check the last 500 bars.
' Find the 2nd Swing-Low occurrence back
' where the 4 bars after and the 5 bars
' prior to the current bar.
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.

```

Returns:

Value returns a positive bar integer number when found, or a **-1** when a Swing-Low is not found.

The positive number represent the bar count back from the offset starting index of the swing bar. Current bar is number zero.

Links:

[SwingHigh](#), [SwingHighBars](#), [SwingLow](#)

See Also:

[Series Functions](#)

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 598

SetSeriesNameColor**Syntax:**

SetSeriesNameColor

Parameter:**Description:****Returns:**

Example:

--

Links:

--

See Also:

--

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 730

3.7 String

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.
AsString (Type Conversion Function)	Converts a numeric expression and it assigns the result to a text variable. Optional [addCommas] parameter when set to True will add commas to the integer portion of a large number.
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 (Replaced: GetNumberField GetStringField)	This function will parse a comma delimited string and return the nth field. Function will access both number and text character fields. NOTE: GetStringField was previously required for text character fields, and GetNumberField was required for number fields before GetField was created to handle both data types.
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 a string text expression

Parameter:	Description:
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.

Example:
<p>Note: Note that to concatenate strings, the Plus (+) sign is used as follows:</p> <pre>string1 = "Hello" string2 = "World" resultString = string1 + string2 PRINT resultString</pre>
Results:
Prints "HelloWorld"

Links:
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:	Description:
expression	expression, if it is not a string it will be converted to a string
value	ASCII character code

Standard ASCII Table:

This is the Standard ASCII table for converting text to an ASCII value, and for using the Chr() function to convert the ASCII number to a text character.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYM (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

ASCII Character Reference Table

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:

[ASCII To Characters](#), [Character to ASCII to Character](#)

See Also:

[Data Groups and Types](#)

ASCII To Characters

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.

Standard ASCII Table:

This is the Standard ASCII table for converting text to an ASCII value, and for using the Chr() function to convert the ASCII number to a text character.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DCL (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	:	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

ASCII Character Reference Table

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, Character to ASCII to Character](#)

See Also:

[Data Groups and Types](#)

AsString

This is a Conversion Function where it will accept a numeric value or variable and output the information as a text variable.

Converts a numeric expression and it assigns the result to a text variable.

Optional [addCommas] parameter when set to True will add commas to the integer portion of a large number.

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)

```

Result:

```

This prints "579"
This prints "123456"

```

```

-----
VARIABLES: decimalNum TYPE: FLOATING

decimalNum = 1235687.14254

PRINT "No Commas in number: " + AsString(decimalNum, 2)
PRINT "Account Balance is " + AsString(decimalNum, 2, TRUE)

```

Result:

```

No Commas in number: 1235687.14
Account Balance is 1,235,687.14

```

```

-----
VARIABLES: floatVar TYPE: FLOATING

```

Example:

```
floatVar = 123.456789
```

```
PRINT AsString( floatVar, 2 )
```

Result:

This prints "123.45"

```
|-----
```

```
VARIABLES: floatVar TYPE: FLOATING
```

```
floatVar = 123456.456789
```

```
PRINT AsString( floatVar, 2, true )
```

Result:

This prints "123,456.45"

Links:

[AsFloating](#), [AsInteger](#), [AsSeries](#), [IsFloating](#), [IsInteger](#), [IsString](#), [FormatString](#)

See Also:

Edit Time: 9/11/2020 4:48:24 PM

Topic ID#: 81

Character to ASCII to Character

This is an example that demonstrates how to convert a text string into its ASCII values. In this example, conversion will also create a log of how each loop in the example script shows with text character the computer has assigned a number, and how that number can be used to convert the number back into its original character.

Standard ASCII Table:

This is the Standard ASCII table for converting text to an ASCII value, and for using the Chr() function to convert the ASCII number to a text character.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	€#32;	Space	64	40	100	€#64;	Ø	96	60	140	€#96;	`
1	1	001	SOH (start of heading)	33	21	041	€#33;	!	65	41	101	€#65;	A	97	61	141	€#97;	a
2	2	002	STX (start of text)	34	22	042	€#34;	"	66	42	102	€#66;	B	98	62	142	€#98;	b
3	3	003	ETX (end of text)	35	23	043	€#35;	#	67	43	103	€#67;	C	99	63	143	€#99;	c
4	4	004	EOT (end of transmission)	36	24	044	€#36;	€	68	44	104	€#68;	D	100	64	144	€#100;	d
5	5	005	ENQ (enquiry)	37	25	045	€#37;	€	69	45	105	€#69;	E	101	65	145	€#101;	e
6	6	006	ACK (acknowledge)	38	26	046	€#38;	€	70	46	106	€#70;	F	102	66	146	€#102;	f
7	7	007	BEL (bell)	39	27	047	€#39;	'	71	47	107	€#71;	G	103	67	147	€#103;	g
8	8	010	BS (backspace)	40	28	050	€#40;	(72	48	110	€#72;	H	104	68	150	€#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	€#41;)	73	49	111	€#73;	I	105	69	151	€#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	€#42;	*	74	4A	112	€#74;	J	106	6A	152	€#106;	j
11	B	013	VT (vertical tab)	43	2B	053	€#43;	+	75	4B	113	€#75;	K	107	6B	153	€#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	€#44;	,	76	4C	114	€#76;	L	108	6C	154	€#108;	l
13	D	015	CR (carriage return)	45	2D	055	€#45;	-	77	4D	115	€#77;	M	109	6D	155	€#109;	m
14	E	016	SO (shift out)	46	2E	056	€#46;	.	78	4E	116	€#78;	N	110	6E	156	€#110;	n
15	F	017	SI (shift in)	47	2F	057	€#47;	/	79	4F	117	€#79;	O	111	6F	157	€#111;	o
16	10	020	DLE (data link escape)	48	30	060	€#48;	0	80	50	120	€#80;	P	112	70	160	€#112;	p
17	11	021	DC1 (device control 1)	49	31	061	€#49;	1	81	51	121	€#81;	Q	113	71	161	€#113;	q
18	12	022	DC2 (device control 2)	50	32	062	€#50;	2	82	52	122	€#82;	R	114	72	162	€#114;	r
19	13	023	DC3 (device control 3)	51	33	063	€#51;	3	83	53	123	€#83;	S	115	73	163	€#115;	s
20	14	024	DC4 (device control 4)	52	34	064	€#52;	4	84	54	124	€#84;	T	116	74	164	€#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	€#53;	5	85	55	125	€#85;	U	117	75	165	€#117;	u
22	16	026	SYM (synchronous idle)	54	36	066	€#54;	6	86	56	126	€#86;	V	118	76	166	€#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	€#55;	7	87	57	127	€#87;	W	119	77	167	€#119;	w
24	18	030	CAN (cancel)	56	38	070	€#56;	8	88	58	130	€#88;	X	120	78	170	€#120;	x
25	19	031	EM (end of medium)	57	39	071	€#57;	9	89	59	131	€#89;	Y	121	79	171	€#121;	y
26	1A	032	SUB (substitute)	58	3A	072	€#58;	:	90	5A	132	€#90;	Z	122	7A	172	€#122;	z
27	1B	033	ESC (escape)	59	3B	073	€#59;	;	91	5B	133	€#91;	[123	7B	173	€#123;	{
28	1C	034	FS (file separator)	60	3C	074	€#60;	<	92	5C	134	€#92;	\	124	7C	174	€#124;	
29	1D	035	GS (group separator)	61	3D	075	€#61;	=	93	5D	135	€#93;]	125	7D	175	€#125;	}
30	1E	036	RS (record separator)	62	3E	076	€#62;	>	94	5E	136	€#94;	^	126	7E	176	€#126;	~
31	1F	037	US (unit separator)	63	3F	077	€#63;	?	95	5F	137	€#95;	_	127	7F	177	€#127;	DEL

ASCII Character Reference Table

Example:

```

VARIABLES: n, iTextLen, iCharNum, iChar  Type: Integer
VARIABLES: sText, sTxtChar, sTxtChar2, MyString  Type: String

' Assign String to Convert Character Values
sText = "<![CDATA[ xxxxxxxx"
' Determine the Length of the String
iTextLen = Len(sText)
' Clear the Working Output String
MyString = ""

' Let TB show the ASCII Characters for the String
For n = 1 TO iTextLen STEP 1
  ' Get 1 Character from a String
  sTxtChar = mid( sText, n, 1 )
  ' Convert Character to its ASCII Number
  iCharNum = ASCII(sTxtChar)
  ' Show What is happening on each cycle of the loop
  PRINT iCharNum, " = ", sTxtChar
  ' Concatenate the results of the looping changes
  MyString = MyString + "Chr(" + AsString(iCharNum)+")+"
Next ' n

' Remove the Last "+" from the created string
iTextLen = Len(MyString) - 1

' Update the created string without the "+" sign
MyString = Left(MyString, iTextLen)

' Show processing Results
PRINT MyString

```

Results:

The information shown below is the output of create when each character is converted to ASCII, and the text value that number illustrates.

```

60 = <
33 = !
91 = [
67 = C
68 = D
65 = A
84 = T
65 = A
91 = [
32 =
120 = x
Chr(60)+Chr(33)+Chr(91)+Chr(67)+Chr(68)+Chr(65)+Chr(84)+Chr(65)+Chr(91)
+Chr(32)+Chr(120)+Chr(120)+Chr(120)+Chr(120)+Chr(120)+Chr(120)+Chr(120)

```

Example:

Links:

[ASCII, ASCII To Characters](#)

See Also:

[Data Groups and Types](#)

EncloseInQuotes

This string function will place a pair of quotation characters "" around the text information contained in the text contained in the string variable used with this function.

Syntax:

```
textOutput = EncloseInQuotes( anyStringText )
```

Parameter	Description:
:	
anyStringText	Any word or phrase group of characters assigned to a string.

Returns:

The supplied string of characters enclosed in a pair of quotation marks.

Example:

```
VARIABLES: anyStringText, textOutput Type: String
' Assign Text to a string variable.
anyStringText = "Trading Blox!"
' Execute the EncloseInQuote function.
textOutput = EncloseInQuotes( anyStringText )
' Display the text results
PRINT "I like " + textOutput
```

Returns:

I like "Trading Blox!"

Links:

[String Functions](#),

See Also:

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

FormatString function formats numbers, or a string.

It powerful tool that is also basically the same C++ [printf function](#) explain on the page the link displays.

NOTE:

When it is reducing a number of decimal places need to be removed during a conversion from a decimal number to STRING representation of that number, and a requirement that the current numbers being displayed must not be rounded, the example near the bottom of this topic will provide [a process that will prevent the rounding of any value greater than 5](#) from changing the displayed digits during the conversion.

Syntax:

```
formattedString = FormatString( stringToFormat, value1, [value2],
[value3] )
```

Parameter:	Description:
stringToFormat	String to be formatted: "%[flags] [width] [.precision]" Use: %i when formatting integers, %f when formatting floats, and %s when formatting strings.
value1	Can be any Integer , Floating or String TYPE expression or value.
[value2]	Optional: Integer , Floating or String TYPE expression of value.
[value3]	Optional: This optional parameter number must be a Floating TYPE variable.
formattedString	Result returns a formatted string.

Additional Arguments:

Depending on the format string, function may expect a sequence of additional arguments. Each additional argument containing one value to be inserted instead of each **%-** tag specified in the *format* parameter, when used.

There should be the same number of 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:

Additional Arguments:		
<i>specifier</i>	Output	Example
c	Character	a
d or i	Signed decimal integer	392
e	Scientific notation (mantise/ exponent) using e character	3.9265e+2
E	Scientific notation (mantise/ exponent) using E character	3.9265E+2
f	Decimal floating point	392.65
g	Use the shorter of %e or %f	392.65
G	Use the shorter of %E or %f	392.65
o	Signed octal	610
s	String of characters	sample
u	Unsigned decimal integer	7235
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 <i>width sub-specifier</i>).
+	Forces to preceded 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.

Additional Arguments:	
<i>flags</i>	Description:
	Used with <code>e</code> , <code>E</code> and <code>f</code> , forces the written output to contain a decimal point even if no digits would follow. If no digits follow, no decimal point is written by default.
	Used with <code>g</code> or <code>G</code> the result is the same as with <code>e</code> or <code>E</code> but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width 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 format 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 (<code>d</code>, <code>i</code>, <code>o</code>, <code>u</code>, <code>x</code>, <code>X</code>): precision specifies the minimum number of digits to be written.</p> <p>If the value to be written is shorter than this number, the result is padded with leading zeros.</p> <p>The value is not truncated even if the result is longer.</p> <p>A precision of 0 means that no character is written for the value 0.</p> <p>For <code>e</code>, <code>E</code> and <code>f</code> specifiers: this is the number of digits to be printed <u>after</u> the decimal point.</p> <p>For <code>g</code> and <code>G</code> specifiers: This is the maximum number of significant digits to be printed.</p> <p>For <code>s</code>: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered.</p>

Additional Arguments:	
<i>.precision</i>	Description:
	For <code>c</code> type, there is no effect. Without a specified precision , the default precision is 1. If the period is specified without an explicit value for precision , 0 is assumed.
<code>.*</code>	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.
length Description:	
<code>h</code>	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: <code>i, d, o, u, x</code> and <code>X</code>).
<code>l</code>	The argument is interpreted as a long int or unsigned long int for integer specifiers (<code>i, d, o, u, x</code> and <code>X</code>), and as a wide character or wide character string for specifiers <code>c</code> and <code>s</code> .
<code>L</code>	The argument is interpreted as a long double (only applies to floating point specifiers: <code>e, E, f, g</code> and <code>G</code>).

Note: Use `%i` when formatting integers, `%f` when formatting floats, and `%s` when formatting strings.

Example:
<pre>PRINT FormatString("I am %i years old.", 5) PRINT FormatString("The price of an %s is \$%.2f today.", "apple", 100.123456) PRINT FormatString("I'd like to say %s to %s.", "hello", "Sam")</pre>
<p>Results:</p> <pre>I am 5 years old. The price of an apple is \$100.12 today. I'd like to say hello to Sam.</pre>

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 convert a larger number to a string variable with commas:

Example:

```

VARIABLES: decimalNum    TYPE: FLOATING

decimalNum = 1235687.14254

PRINT "No Commas in number: " + AsString(decimalNum, 2)
PRINT "Account Balance is $" + AsString(decimalNum, 2, TRUE)

```

Result:

```

No Commas in number: 1235687.14
Account Balance is $1,235,687.14

```

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|
```

Example - Rounding Control:

```
' -----
'   Decimal Rounding Control Example
' -----
' Local Working value variables
VARIABLES: f0, f1, f2 Type: Floating
VARIABLES: s1 Type: String
VARIABLES: x1 Type: Integer
' -----
'   Example without control
f0 = 1.53651      ' Default Number
' Use Default Value
f1 = f0
' Display Default Value
PRINT "f1 = ", f1
' Convert the number to a string with 3-decimal numbers.
s1 = FormatString("%.3f", f1)
' Display Number Converted to STRING
PRINT "s1 = ", s1
PRINT
```

Example - Rounding Control:

```

' -----
' Example with rounding control
f1 = f0
PRINT "f1 = ", f1
' Adjust number value to preserved 3-decimal numbers
f2 = f1 * 1000
PRINT "f2 = ", f2
' Convert adjusted capture decimal portion of number
x1 = AsInteger(f2)
' Show Integer value
PRINT "x1 = ", x1
' Move 3-Decimal of the number back
f2 = x1 * 0.001
' Show Integer converted to a decimal value
PRINT "f2 = ", f2
' Convert the number to a text value
s1 = FormatString("%.3f", f2)
' Display converted FLOATING number to a text value
PRINT "s1 = ", s1
PRINT
' -----

```

Results:**Example without control**

```

Use Default Value
Display Default Value
  f1 = 1.536510000
Convert the number to a string with 3-decimal numbers.
Display Number Converted to STRING
s1 = 1.537

```

Example with rounding control

```

f1 = 1.536510000
Adjust number value TO preserve 3-decimal numbers
f2 = 1536.510000000
Convert adjusted capture decimal portion of number
Show Integer value
x1 = 1536
Move 3-Decimal of number back
Show Integer converted to a decimal value
f2 = 1.536000000
Convert the number to a text value
Display converted FLOATING number to a text value
s1 = 1.536

```

Links:[AsString](#), [Random](#)

Links:**See Also:**[Data Groups and Types](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 347

GetField

This function will parse a comma delimited string and return the nth field.

NOTE:

[GetStringField](#) was previously required for text character fields, and [GetNumberField](#) was required for number fields before [GetField](#) was created to handle both data types.

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, and with the [GetFieldCount](#) function to discover how many fields are available.

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 prior versions.

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:

[GetFieldCount](#), [ReadLines](#), [RemoveCommasBetweenQuote](#), [RemoveNonDigits](#)

Links:

See Also:

[Data Groups and Types](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 352

GetFieldCount

This function returns the number of comma delimited values in a text string. When there are no values in the string function the return value will be 1. One is returned because the function counts commas and adds 1 comma because the last value won't always be followed by a comma. This process is shown in the examples below.

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:

```
' No Commas and no values
sStringRecord = ""
x = GetFieldCount(sStringRecord)
PRINT "sStringRecord : ", sStringRecord
PRINT " Return Count : ", x
```

Results:

```
sStringRecord :
Return Count : 1
```

Example:

```
' When a String variable can be empty, consider this modification
' No Commas and no values
sStringRecord = ""
x = Min(Len(Trim(sStringRecord)), GetFieldCount(sStringRecord))
PRINT "sStringRecord : ", sStringRecord
PRINT " Return Count : ", x
```

Results:

```
sStringRecord :
Return Count : 0
```

Example:

Example:

```
' No Comma After Last Value
sStringRecord = "S,10,20,30,40"
x = GetFieldCount(sStringRecord)
PRINT "sStringRecord : ", sStringRecord
PRINT " Return Count : ", x
```

Results:

```
sStringRecord : S,10,20,30,40
Return Count : 5
```

Example:

```
' Comma After Last Value
sStringRecord = "S,10,20,30,40,"
x = GetFieldCount(sStringRecord)
PRINT "sStringRecord : ", sStringRecord
PRINT " Return Count : ", x
```

Results:

```
sStringRecord : S,10,20,30,40,
Return Count : 6
```

Links:

[GetField](#), [ReadLines](#), [RemoveCommasBetweenQuote](#), [RemoveNonDigits](#)

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](#)

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:
<pre>value = LeftCharacters(inputString, length)</pre>
<p>OR</p> <pre>value = Left(inputString, length) ' Short form usage: "Left"</pre>

Parameter:	Description:
<code>inputString</code>	String from which to extract the left-side characters.
<code>length</code>	Number of characters on the right side of text to extract.
<code>value</code>	Extracted characters.

Example:
<pre>value = LeftCharacters("Hello", 3) ' Returns "Hel" value = Left("Hello", 2) ' Returns "He" value = Left("Hello", 40) ' Returns "Hello"</pre>
<p>Results: See example comments.</p>

Links:
MiddleCharacters , RightCharacters
See Also:
Data Groups and Types

LowerCase

Converts any Upper-Case text characters to Lower-Case.

Syntax:

```
value = LowerCase( inputString )    ' Short form: "LCase"
```

Parameter:	Description:
inputString	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](#)

MiddleCharacters

Returns a specified number of characters from the middle of a string.

Syntax:
<pre>value = MiddleCharacters(inputString, start, length)</pre>
OR
<pre>value = Mid(inputString, start, length) ' Short form usage: "Mid"</pre>

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:
<pre>value = MiddleCharacters("Hello", 2, 3) ' Returns "ell" value = MiddleCharacters("Hello", 3, 2) ' Returns "ll"</pre>
Results: See example comments.

Links:
LeftCharacters , RightCharacters
See Also: Data Groups and Types

RemoveCommasBetweenQuotes

Function removes commas that are between quotes. This is useful when an imported text file's record string values are separated by commas.

Note:

This option does not remove commas in a comma separated string (see [AsString](#) function).

When numbers are not created within Trading Blox Builder, use the [ReplaceString](#) function to remove commas.

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](#), [ReplaceString](#)

See Also:

[Data Groups and Types](#)

RemoveNonDigits

Removes any characters that are not numbers from the string.

Note:

This option does not remove commas in a comma separated string (see [AsString](#) function).

When numbers are not created within Trading Blox Builder, use the [ReplaceString](#) function to remove commas.

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:

[GetField](#), [RemoveCommasBetweenQuotes](#), [ReplaceString](#)

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,
stringToReplaceWith )
```

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: Replace Word

```
' Word Replacement Example
VARIABLES: sNewText  Type: String

' Remove Any Text That looks like: 'Hello' and append 'World'
sNewText = ReplaceString( "Hello World", "Hello", "Goobye" )

PRINT "ReplaceString = ", sNewText
```

Results:

```
ReplaceString =  Goobye World
```

Example: Replace Colons with Commas

```
' Correlation Matrix Description Example
VARIABLES: sNewTextNum  TYPE: STRING

sNewTextNum = ReplaceString( "123:456:789", ":", "," )

PRINT "ReplaceString = ", sNewTextNum
```

Results:

```
ReplaceString =  123,456,789
```

Example: Remove Commas in Numbers

```
' Remove Commas from a number with comma seperations
VARIABLES: sNewTextNum  TYPE: STRING

sNewTextNum = ReplaceString( "123:456:789", ":", "" )
```

Example: Replace Word

```
PRINT "ReplaceString = ", sNewTextNum
```

Results:

```
ReplaceString = 123456789
```

Links:

[AsString](#), [FormatString](#), [RemoveNonDigits](#)

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 = <code>StringLength(inputString)</code>	
OR	
value = <code>Len(inputString)</code>	' 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 = <code>StringLength(inputString)</code>	' Returns 5
inputString = "Goodbye"	
value = <code>Len(inputString)</code>	' Returns 7
value = <code>StringLength("Hello")</code>	' Returns 5
value = <code>Len("Goodbye")</code>	' 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:	
<code>value = UpperCase(inputString)</code>	
OR	
<code>value = UCase(inputString)</code> ' Short form usage: "UCase"	

Parameter:	Description:
inputString	Text or string expression with lower case characters
value	Lower Case characters converted to Upper Case characters.

Example:	
<code>value = UpperCase("Hello")</code>	' Returns "HELLO"
<code>value = UCase("hello")</code>	' Returns "HELLO"
Results:	

Links:
See Also:
Data Groups and Types

3.8 Type Conversion

"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:
AsDate	This function converts an Integer date value into a string date value that can be used for display or assignment to a text String.
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 .

Links:

[Data Groups and Types](#), [Types](#), [Variables](#),

See Also:

AsFloating

Converts an passed expression to a **TYPE FLOATING** point numeric value.

Syntax:	
<code>value = AsFloating(expression)</code>	

Parameter:	Description:
expression	Expression to convert to a floating numeric value
value	Converted expression is returned as a TYPE Floating number.

Example:
<pre>VARIABLES: stringOne, stringTwo TYPE: STRING stringOne = "123.456" stringTwo = "456.789" print stringOne + stringTwo print AsFloating(stringOne) + AsFloating(stringTwo)</pre>
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:**Description:**

Parameter:	Description:
AnyValue	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[Ndx] )
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](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 45

AsSeries

Changes how a **TYPE SERIES** array is passed to a [Chart Object](#) or [Scrip Object](#) Custom function.

Most functions can accept values from variables by the value in the variable that is passed to the function. Series data cannot be passed by value, unless a single indexed element is being passed. To pass all the elements in a series array it must be passed by reference. **AsSeries()** provides the reference information required to access the elements in the series.

Script Execute functions and Custom Chart Director functions require the first elements memory location of data's for the first element the series. Charting functions also require data passed to their functions also include an element count.

This function is not required for static variables or properties that are not an Auto-Index or Manually Indexed series.

Syntax:

```
AsSeries( AnySeries )
```

Parameter:	Description:
AnySeries	Any series assigned to any Custom Chart or Script Object parameter is required to use this conversion function.

Example:

```
' Add 5 element values to represent "bar1"
chart.AddBarSeries( AsSeries( bar1 ), 5 )

' Passing Series to a Custom Function
script.Execute("ExampleFunction", AsSeries(anySeries))
```

Links:

[Chart Object](#), [AsFloating](#), [AsInteger](#), [AsString](#), [IsFloating](#), [IsInteger](#), [IsString](#), [Scrip Object](#)

See Also:

[Data Groups and Types](#)

AsString

Converts a numeric expression and it assigns the result to a text variable.

Optional [addCommas] parameter when set to True will add commas to the integer portion of a large number.

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)
```

Result:

```
This prints "579"
This prints "123456"
```

```
' -----
VARIABLES: decimalNum TYPE: FLOATING
```

```
decimalNum = 1235687.14254
```

```
PRINT "No Commas in number: " + AsString(decimalNum, 2)
PRINT "Account Balance is " + AsString(decimalNum, 2, TRUE)
```

Result:

```
No Commas in number: 1235687.14
Account Balance is 1,235,687.14
```

Example:

```
'-----  
VARIABLES: floatVar TYPE: FLOATING  
floatVar = 123.456789
```

```
PRINT AsString( floatVar, 2 )
```

Result:

This prints "123.45"

```
'-----  
VARIABLES: floatVar TYPE: FLOATING  
floatVar = 123456.456789
```

```
PRINT AsString( floatVar, 2, true )
```

Result:

This prints "123,456.45"

Links:

[AsFloating](#), [AsInteger](#), [AsSeries](#), [IsFloating](#), [IsInteger](#), [IsString](#), [FormatString](#)

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:	Description:
expression	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:	
<code>value = IsString(expression)</code>	

Parameter:	Description:
expression	expression to check
value	TRUE when the expression is of TYPE STRING value.

Example:
<pre>VARIABLES: variableOne TYPE: STRING variableOne = "ABC" IF (IsString(variableOne)) THEN print variableOne, " is a string." ELSE print variableOne, " is NOT a string." ENDIF</pre> <p>Results: This prints "ABC is a string."</p>

Links:
AsFloating , AsInteger , AsSeries , AsString , IsFloating , IsInteger
See Also:
Data Groups and Types

Section 4 – 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:

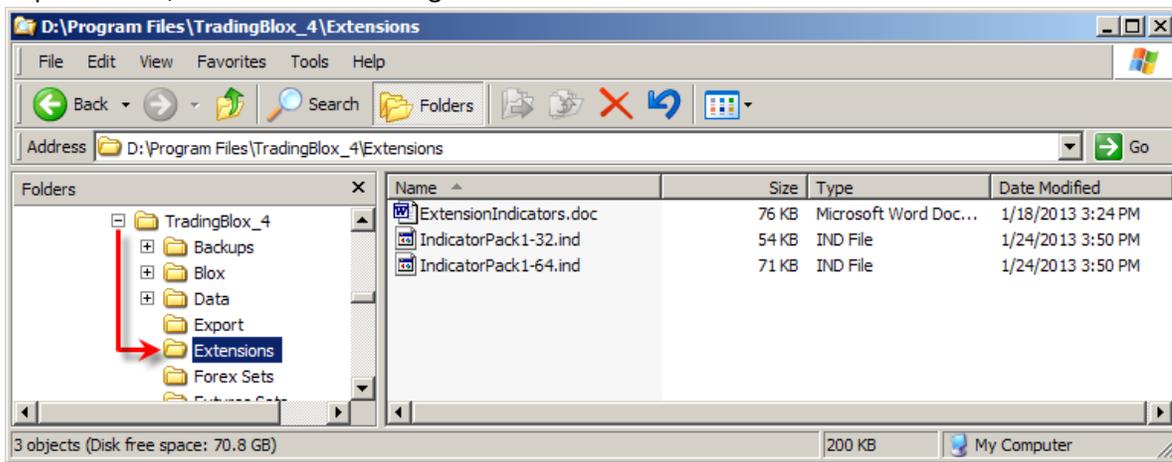
Indicators:

[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:



Indicator Pak Extension Folder Location

4.1 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.</p> <p>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)

Indicators:	Description:
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 Sine 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 sine 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. Can use the Square Root of the Sum of Variations multiplied by 100 as a percentage return.
Hull Moving Average	The Hull Moving Average (HMA) developed by Alan Hull to reduce calculation lag, increasing responsiveness and eliminates noise. Its calculation is elaborate and makes use of the Weighted Moving Average (WMA). This is a scripted version of the Hull Moving Average function as a Blox.
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.

Indicators:	Description:
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.
Klinger Oscillator Indicator	Klinger Oscillator Indicator is often known as the Klinger Volume Oscillator (KVO) was developed by Stephen Klinger. This indicator is a volume and price based oscillator that measures both short and long period volume changes of an instrument.
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

Indicators:	Description:
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 Strength Indicator	True Strength as a chart Indicator is available in the Variables section of the Blox Editor.
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.

Links:

[Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Groups and Types](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 381

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 fields are as follows:

- Name for Code: AvgTrendChannel
- Type: Average Trend Channel
- Value: DHLC / 4
- Time Frame: Bar
- Period: TrendChannelBars
- Not Applicable: (empty)
- Not Applicable: (empty)
- Indicator Value Expression: (empty)
- Scope: Block
- Plots: Plots
- Graph Title: AvgTrendChannel
- Plot Color: (Green)
- Graph Area: Price Chart
- Graph Style: Thin Line
- Offset Plot Ahead One B:
- Display Value:

Average Trend Channel setup dialog

Setup Example (Click on Images to enlarge):



Average Trend Channel Indicator

Links:

[Basic Indicators](#), [Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Group and Types](#), [Indicator Pack 1 Indicators](#)

Chaikin 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	<input type="text" value="ChaikenOsc"/>	<input type="button" value="OK"/>
Type	<input type="text" value="Chaikin Money Flow"/>	<input type="button" value="Cancel"/>
Value	<input type="text"/>	
Time Frame	<input type="text" value="Bar"/>	
Period	<input type="text" value="ChaikenCalcBars"/>	
Not Applicable	<input type="text"/>	
Not Applicable	<input type="text"/>	

Indicator Value Expression

The parser likes the expression.

Scope	<input type="text" value="Block"/>	<input checked="" type="checkbox"/> Plots	<input checked="" type="checkbox"/> Display Value
		Graph Title	<input type="text" value="ChaikenOsc"/>
		Plot Color	<input type="text" value=""/>
		Graph Area	<input type="text" value="Chaikin Money Flow"/>
		Graph Style	<input type="text" value="Histogram"/>
		<input type="checkbox"/> Offset Plot Ahead One Bar	

Chaikin Money Flow Oscillator setup dialog

Setup Example (Click on Images to enlarge):

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 dropdown)
- Time Frame:** Bar
- Smoothing:** Enter Value (dropdown) with a value of 1 in the adjacent input field.
- Not Applicable:** (empty dropdown)
- Not Applicable:** (empty dropdown)
- Indicator Value Expression:** A text area containing the number '0'.
- Expression looks ok:** A text area that is currently empty.
- Scope:** Block
- Plots:** Plots
- Display Value:** Display Value
- Graph Title:** Chaiken Zero
- Plot Color:** (color selection dropdown)
- 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 of the dialog.

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.

Setup Example (Click on Images to enlarge):



Chaiken Money Flow Oscillator

Links:

[Basic Indicators](#), [Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Group and Types](#), [Indicator Pack 1 Indicators](#)

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)
- Scope:** Block
- Graph Settings:**
 - Plots
 - Display Value
 - Graph Title:** CCI
 - Plot Color:** (Red color selected)
 - Graph Area:** Commodity Channel Index
 - Graph Style:** Thin Line
 - Offset Plot Ahead One B:

Buttons for 'OK' and 'Cancel' are located in the top right corner.

Commodity Channel Index setup dialog

Setup Example (Click on Images to enlarge):

The screenshot shows the 'Edit Indicator' dialog box for the 'CCIZeroLine' indicator. The dialog is titled 'Edit Indicator' and has a light blue header. It contains several fields and controls:

- Name for Code:** CCIZeroLine
- Type:** Calculated
- Value:** (empty)
- Time Frame:** Bar
- Smoothing:** Enter Value, with a text input field containing '1'.
- Not Applicable:** (empty)
- Indicator Value Expression:** A text area containing '0'.
- Expression looks ok:** A text area that is currently empty.
- Scope:** Block
- Plots:** Plots
- Display Value:** Display Value
- Graph Title:** CCIZero
- Plot Color:** (dark grey)
- Graph Area:** Commodity Channel Index
- 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.

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.

Setup Example (Click on Images to enlarge):



Commodity Channel Index Indicator

Links:

[Basic Indicators](#), [Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Group and Types](#), [Indicator Pack 1 Indicators](#)

Dominant Cycle

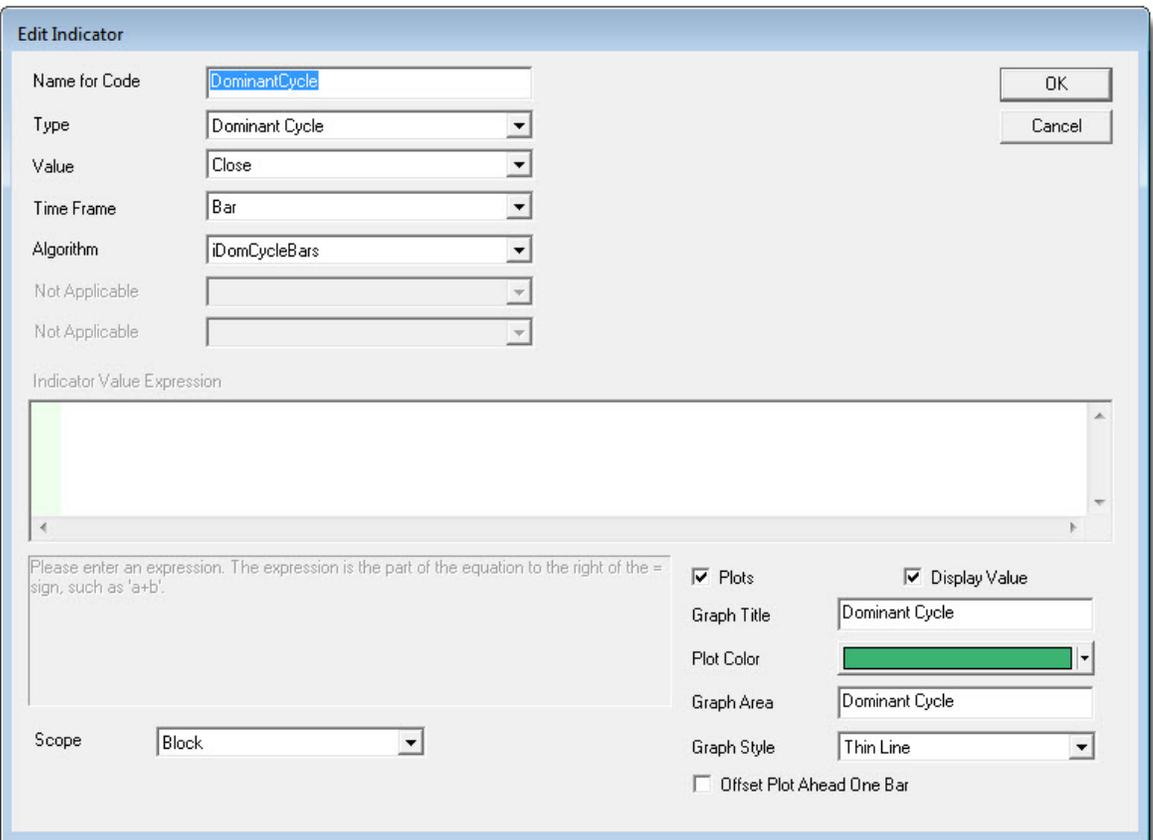
"Estimated dominant cycle period of the market.

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."

Algorithm:

- 1: Homodyne Discriminator Algorithm
- 2: Filter Bank Algorithm

Setup Example (Click on Images to enlarge) :



The screenshot shows the 'Edit Indicator' dialog box for the 'Dominant Cycle' indicator. The dialog is titled 'Edit Indicator' and has a light blue header. It contains several fields and options:

- Name for Code:** DominantCycle
- Type:** Dominant Cycle
- Value:** Close
- Time Frame:** Bar
- Algorithm:** iDomCycleBars
- Not Applicable:** (two empty dropdown menus)
- Indicator Value Expression:** (empty text area)
- Scope:** Block
- Plots:** Plots
- Display Value:** Display Value
- Graph Title:** Dominant Cycle
- Plot Color:** Green
- Graph Area:** Dominant Cycle
- 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.

Dominant Cycle Indicator Setup Example.

Setup Example (Click on Images to enlarge) :

Dominant Cycle IndicatorChart Example.

Links:[Indicator Pack 1 Indicators](#)**See Also:**[Data Group and Types](#), [Basic Indicators](#), [Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#), [Average](#), [SquareRoot](#), [Sum](#)

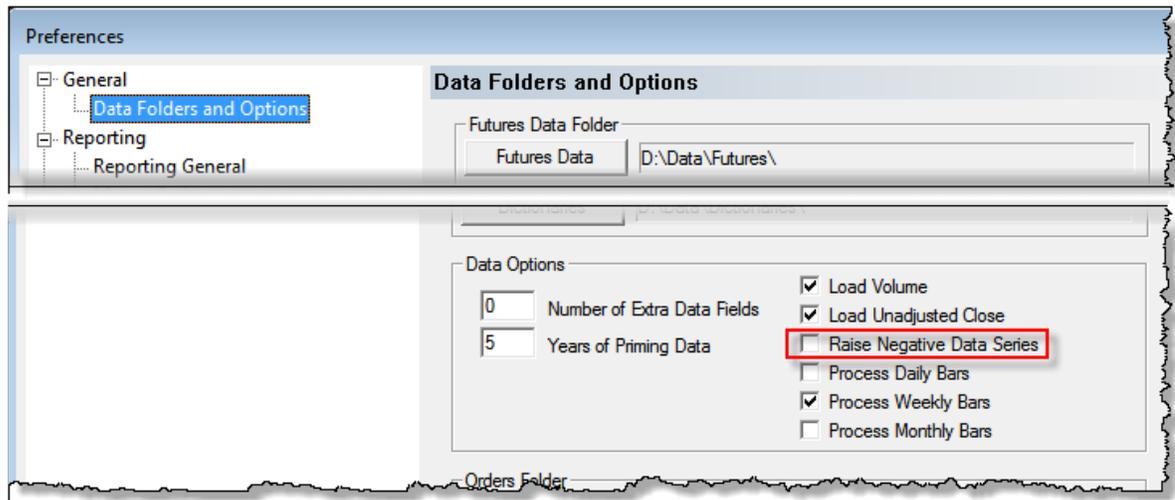
Historic Volatility

Historical Volatility is calculated over the length of the first parameter field named "Period". Initial calculation measures the average deviation from the average price of a financial instrument in a given time period. Standard deviation is the most common but not the only way to calculate historical volatility. It is the method used in this indicator.

Historical Volatility uses second parameter field named "Span" to control the square root of the length of the Sum of the Squares Variations average results. Square root results returned are multiplied by 100 for a percentage compatible value.

Daily Return values are Log values of the result from current-price divided by the previous-price.

Negative prices are not compatible with a Log function. If your selected price series contains any negative price values use the Trading Blox Preference option to raise entire series above zero, or select a difference method for determining volatility.



Trading Blox Preference options are available under the Edit menu.

Creating a Historical Volatility Indicator:

Historical Volatility will use two parameter length integers. Period length is the length of all calculations used to generate the Sum of the Square Variation values. Span length is the calculation length over which the Square Root of the Sum of the Squares is determined before being multiplied by 100.

Setup Example (Click on Images to enlarge):

Edit Indicator

Name for Code	<input type="text" value="VolatilityHistorySpan"/>	<input type="button" value="OK"/>
Type	<input type="text" value="Historic Volatility"/>	<input type="button" value="Cancel"/>
Value	<input type="text" value="Close"/>	
Time Frame	<input type="text" value="Bar"/>	
Period	<input type="text" value="VolatilityLength"/>	
Span	<input type="text" value="iSqrRootSpan"/>	
Not Applicable	<input type="text" value=""/>	

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	<input type="text" value="Block"/>	<input checked="" type="checkbox"/> Plots	<input checked="" type="checkbox"/> Display Value
		Graph Title	<input type="text" value="Vol.Hist.Span"/>
		Plot Color	<input type="text" value="Red"/>
		Graph Area	<input type="text" value="Historic Volatility"/>
		Graph Style	<input type="text" value="Thin Line"/>
		<input type="checkbox"/> Offset Plot Ahead One Bar	

Chart image shows the Red-Line uses a Period Length = 20
Span Length = 10

Blue-Line uses a Period Length = 20
Span Length = 1

Setup Example (Click on Images to enlarge):

Chart example above shows two Historical Volatility indicators for a comparison of how the Span length parameter can amplify the results.

In this image the Light blue line is created using a parameter value of 20 bars and a Span length of 1. Dark red line is created using a period length of 20 bars, and a span length of 10 bars.

Links:

[Basic Indicators](#), [Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#), [Average](#), [SquareRoot](#), [Sum](#)

See Also:

[Data Group and Types](#), [Indicator Pack 1 Indicators](#)

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):

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 Bar

Kaufman Adaptive High Dialog Settings

Setup Example (Click on Images to enlarge):

Edit Indicator

Name for Code: OK

Type: Cancel

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 Bar

Kaufman Adaptive Low Dialog Settings

Setup Example (Click on Images to enlarge):



Links:

[Basic Indicators](#), [Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Group and Types](#), [Indicator Pack 1 Indicators](#)

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 screenshot shows the 'Edit Indicator' dialog box for the Keltner Upper Channel. The dialog is titled 'Edit Indicator' and contains the following fields and options:

- Name for Code:** KeltnerUpperBand
- Type:** Keltner Upper
- Value:** Close
- Time Frame:** Bar
- Moving Average Bars:** KeltnerBars
- ATR Bars:** KeltnerBars
- Channel Width:** KeltnerBandOffset
- Indicator Value Expression:** A large text area containing a green vertical bar.
- Scope:** Block
- Plots:** Plots
- Display Value:** Display Value
- Graph Title:** Keltner Upper Band
- Plot Color:** Green
- 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.

Keltner Upper Channel setup dialog.

Setup Example (Click on Images to enlarge):

Edit Indicator

Name for Code	<input type="text" value="KeltnerLowerBand"/>	<input type="button" value="OK"/>
Type	<input type="text" value="Keltner Lower"/>	<input type="button" value="Cancel"/>
Value	<input type="text" value="Close"/>	
Time Frame	<input type="text" value="Bar"/>	
Moving Average Bars	<input type="text" value="KeltnerBars"/>	
ATR Bars	<input type="text" value="KeltnerBars"/>	
Channel Width	<input type="text" value="KeltnerBandOffset"/>	

Indicator Value Expression

The parser likes the expression.

Scope	<input type="text" value="Block"/>	<input checked="" type="checkbox"/> Plots	<input checked="" type="checkbox"/> Display Value
Graph Title	<input type="text" value="Keltner Lower Band"/>	<input type="checkbox"/> Offset Plot Ahead One Bar	
Plot Color	<input type="text" value="Green"/>		
Graph Area	<input type="text" value="Price Chart"/>		
Graph Style	<input type="text" value="Thin Line"/>		

Keltner Lower Channel setup dialog.

Setup Example (Click on Images to enlarge):

Edit Indicator

Name for Code:

Type:

Value:

Time Frame:

Moving Average Bars:

Smoothing:

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:

Plots Display Value

Graph Title:

Plot Color:

Graph Area:

Graph Style:

Offset Plot Ahead One Bar

Keltner Average Price setup dialog

Setup Example (Click on Images to enlarge):

Keltner Upper, Lower and Average Price indicators

Links:

[Basic Indicators](#), [Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Group and Types](#), [Indicator Pack 1 Indicators](#)

Klinger Oscillator Indicator

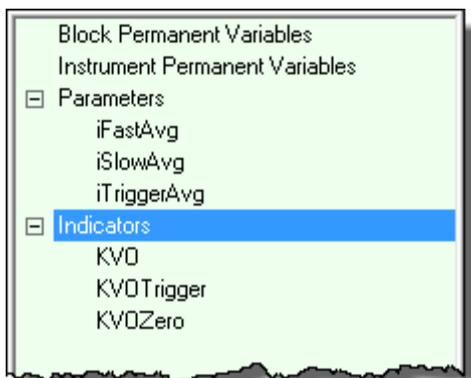
Klinger Oscillator Indicator is often known as the Klinger Volume Oscillator (KVO) was developed by Stephen Klinger. This indicator is a volume and price based oscillator that measures both short and long period volume changes of an instrument.

Klinger Oscillator and Klinger Trigger Chart Example:



Klinger Volume Oscillator Chart Example.

Klinger Oscillator is created in Blox Editor's Indicator area in the Variable section:



Blox Editor Variable area Instrument Section.

Parameter Names and Values Used in Chart Example:

Parameter Name:	Parameter Description:	Chart Display Value:
iFastAvg	KVO Fast Avg. Length:	34
iSlowAvg	KVO Slow Avg. Length:	55
iTriggerAvg	KVO Trigger Avg. Length:	13

Klinger Indicator Settings:

The 'Edit Indicator' dialog box contains the following settings:

- Name for Code: KVO
- Type: Klinger Oscillator
- Value: (empty)
- Time Frame: Bar
- Fast: iFastAvg
- Slow: iSlowAvg
- Signal: Enter Value

Klinger Volume Oscillator Settings Example.

Edit Indicator

Name for Code	<input type="text" value="KVOTrigger"/>
Type	<input type="text" value="Klinger Oscillator"/>
Value	<input type="text"/>
Time Frame	<input type="text" value="Bar"/>
Fast	<input type="text" value="iFastAvg"/>
Slow	<input type="text" value="iSlowAvg"/>
Signal	<input type="text" value="iTriggerAvg"/>

Klinger Volume Trigger Settings Example.

Edit Indicator

Name for Code	<input type="text" value="KV0Zero"/>	
Type	<input type="text" value="Calculated"/>	
Value	<input type="text"/>	
Time Frame	<input type="text" value="Bar"/>	
Smoothing	<input type="text" value="Enter Value"/>	<input type="text" value="1"/>
Not Applicable	<input type="text"/>	
Not Applicable	<input type="text"/>	
Indicator Value Expression	<input type="text" value="0"/>	

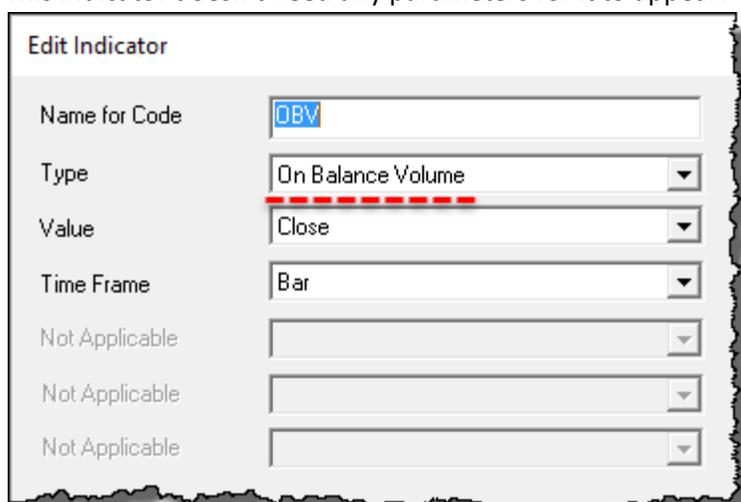
Klinger Oscillator Zero Level Settings.

On Balance Volume

On Balance Volume (OBV) was developed by Joe Granville. It was introduced in his 1963 book, Granville's New Key to Stock Market Profits. Its operation measures buying and selling pressure as a cumulative indicator. On price up-bars, it adds volume of the price bar. On price down-bars, it subtracts the bar's volume from the accumulated series.

This indicator's purpose was to measure positive and negative price volume flows. It is often applied with a divergence calculation that measures agreement between OBV and price. When the price and OBV directions agree, OBV is viewed as a supporting indicator of the current near term price movements.

This indicator doesn't need any parameters for it to appear:



The image shows a screenshot of the 'Edit Indicator' dialog box. The title is 'Edit Indicator'. The settings are as follows:

Name for Code	OBV
Type	On Balance Volume
Value	Close
Time Frame	Bar
Not Applicable	
Not Applicable	
Not Applicable	

On Balance Volume Indicator Settings Example.

Display below a chart indicator as a histogram will appear this way:



On Balance Volume Indicator Chart Example.

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 9

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.

The trend vigor indicator is not the same as the Relative Vigor Index (RVI). Trend Vigor measures the strength of the trend relative to the cycle. The larger the value of the Trend Vigor, the more powerful the trend is relative to the cycle. Trend Vigor, as best I can recall, was first mentioned in Ehlers November 2008 article "Corona Charts".

The trend vigor is the slope of the close (or any other price input) over one full dominant cycle divided by the amplitude of the dominant cycle.

Example Approach:

```
offset = measured dominant cycle
```

```
close - close[offset]/amplitude
```

The amplitude is more complicated. It is defined as:

```
sqrt( (InPhase * InPhase) + (Quadrature * Quadrature) )
```

The Inphase and Quadrature are components derived when calculating the dominant cycle. Detailed discussion of the InPhase and Quadrature components can be found in Ehler's book, "Rocket Science for Traders" in the chapter discussing the Homodyne Discriminator dominant cycle calculation.

Indicator Settings:

Parameter:	Description:
TrendVigor Bars	Number of price-bars to use in the Trend Vigor calculation.

Setup Example:

Edit Indicator

Name for Code: TrendVigor OK

Type: Trend Vigor Cancel

Value: Close

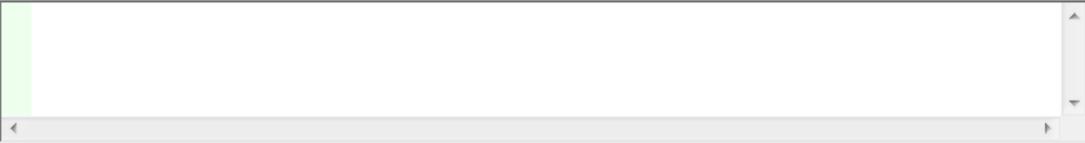
Time Frame: Bar

Algorithm: TrendVigorBars

Not Applicable:

Not Applicable:

Indicator Value Expression



The parser likes the expression.

Scope: Block

Plots Display Value

Graph Title: TrendVigor

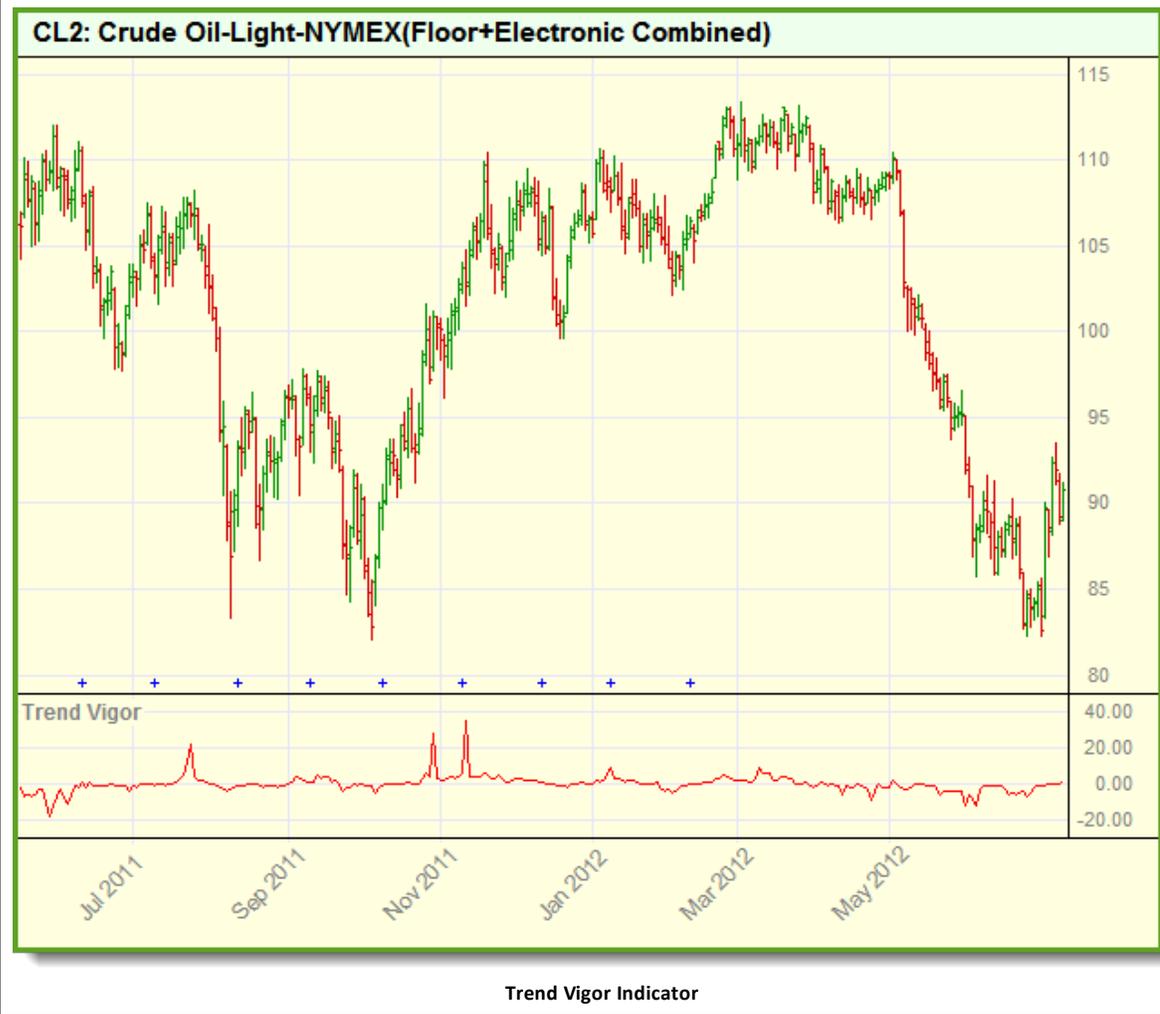
Plot Color:

Graph Area: Trend Vigor

Graph Style: Thin Line

Offset Plot Ahead One Bar

Setup Example:



Links:

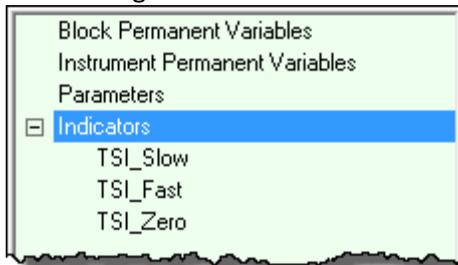
[Basic Indicators](#), [Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Group and Types](#), [Indicator Pack 1 Indicators](#)

True Strength Indicator

True Strength as a chart Indicator is available in the Variables section of the Blox Editor.



Bill Blau published this indicator in Stocks & Commodities November of 1991 issue. A PDF of the indicator's description was found here: <https://tinyurl.com/h8d7krf>

This indicator is shown in the article as a two line crossover oscillator:



True Strength Dual Line Oscillator Example

Indicator's process applies an exponential average (EMA) to two different approaches of momentum series. For example, a first calculation method uses the raw momentum value of x-period EMA that is then smoothed again by a y-period EMA.

The second momentum method uses the ABS of the raw momentum calculation the x-period EMA that is also smoothed by an EMA over a y-period. Each line's oscillator display is created by dividing the raw smoothed result of the first method by the ABS smoothed results of the second method. Division results creates one of the lines of a directional index oscillator that can be smoothed again. A second line display is created using the same division result with a no smoothing or smoothing of a different period.

Edit Indicator

Name for Code	TSL Slow	
Type	True Strength Blau	
Value	Close	
Time Frame	Bar	
Smooth 1	Enter Value	25
Smooth 2	Enter Value	13
Signal	Enter Value	13

Default parameter values entered manually for this example.

x-period parameter
y-period parameter
z-period parameter

True Strength Dual Line Oscillator Dialog Setting Example.

Above image shows the settings for the first line. Settings for the second line are the same except for the Signal parameter value. Second line, colored red shows a Signal value of one. Zero line is created by creating a calculated indicator with the value of zero placed into its script coding section:

Edit Indicator

Name for Code	TSL Zero	
Type	Calculated	
Value		
Time Frame	Bar	
Smoothing	Enter Value	1
Not Applicable		
Not Applicable		
Indicator Value Expression	0	

True Strength Dual Line Oscillator Dialog Zero Line Settings.

Edit Time: 9/11/2020 4:48:30 PM

Topic ID#: 52

4.2 Indicator Pack 1 Series Functions

All the Series Functions listed in this topic are contained in both version of [Indicator Pack 1](#) extension.

Function Name:	Description:
EhlersZeroLagEma	Adaptive exponential moving average. The moving average adapts based on the distance between the moving average and the current price.
InstantaneousTrendLine	Adaptive moving average whose length is determined each bar by the dominantCycleSeries parameter.
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.
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
Momentum	Momentum is the difference between current price and the price a specified number of bars ago.
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.
Percentile	Returns a price value estimate of the specified percentile level.
PercentRank	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.
RateOfChange	The Rate of Change (ROC) indicator provides a percentage that the security's price has changed over the specified period.
SpearmanCorrelation	Spearman correlation is an alternative to the Pearsons correlation and correlates based on the ranking of the series values.
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.
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.

Function Name:	Description:
SpearmanLogCorrelationSync	Spearman correlation is an alternative to the Pearson's correlation and correlates based on the ranking of the series values. Used for IPV Auto Indexed Series or Price Series and syncs the dates of the two series before computing the correlation. This version computes the log returns before computing the correlation.
WMA	Weighted Moving Average calculation method 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.
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.

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

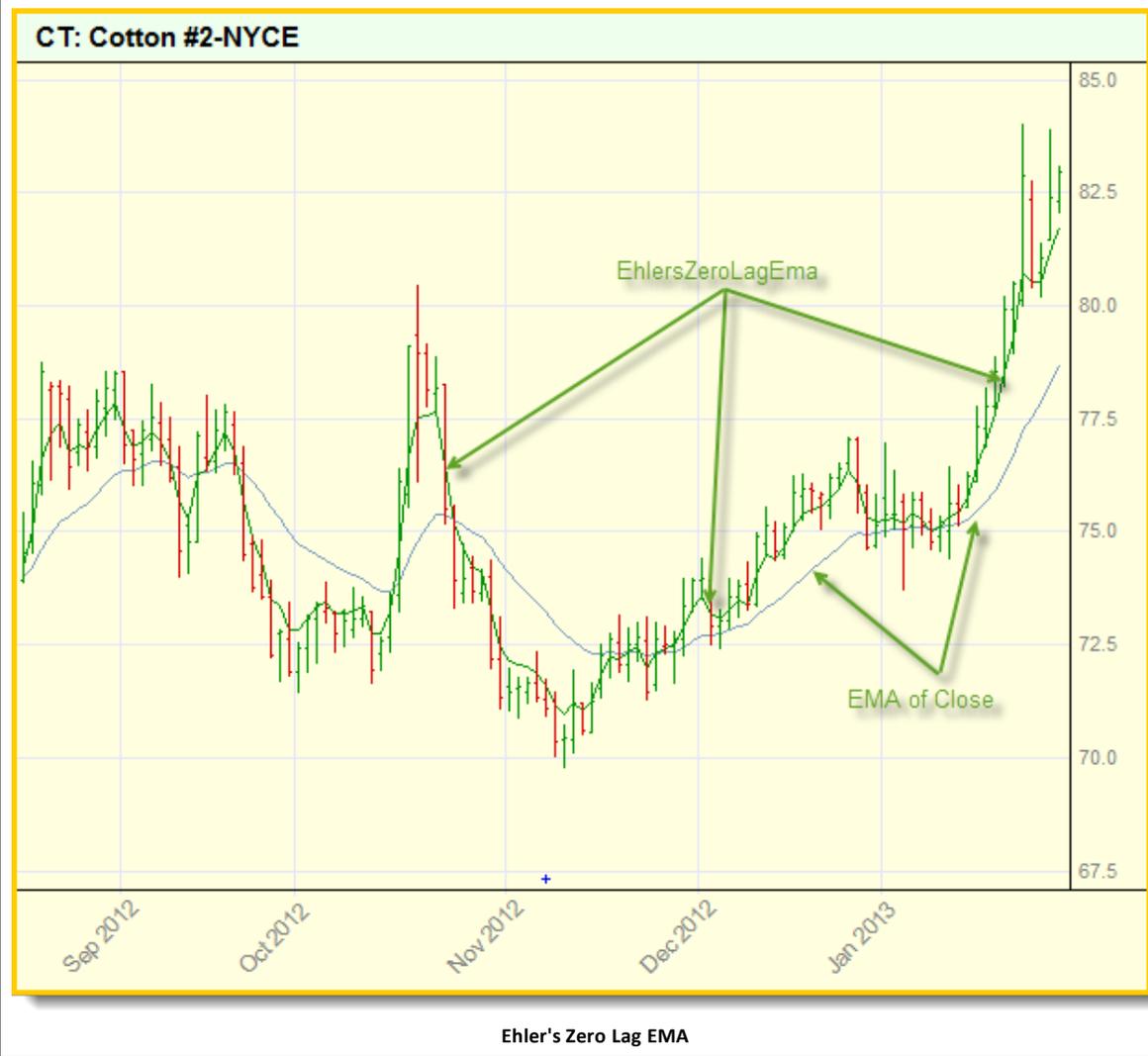
Returns:

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:

Example:**Links:**[EMA](#)**See Also:**[Data Group and Types](#)

InstantaneousTrendLine

Adaptive moving average whose length is determined by each bar in the dominantCycleSeries data loaded length.

The [Instantaneous Trendline](#) was created by John [Ehlers](#) (Rocket Science For Traders [pgs 109-110](#)) and this indicator is perfect for determining the medium to long term trend. Buy when the indicator line is green and sell when it is red. I will be introducing a different version of this indicator which is perfect for short term trends so these will pair great together.

Syntax:

```
InstantaneousTrendLine(series, dominantCycleSeries)
```

Parameter:	Description:
series	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.

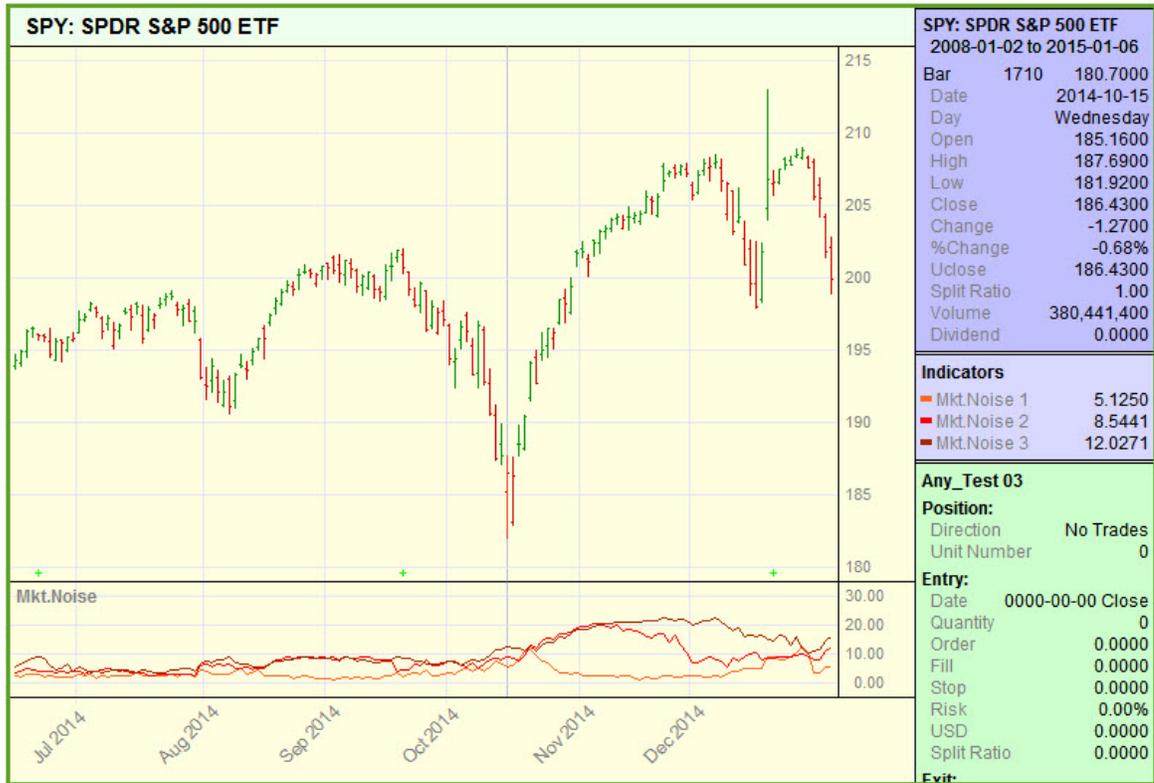
Syntax:

```
MarketNoise(series, bars)
```

Parameter:	Description:
series	The name of the series.
bars	The number of bars over which to calculate noise amount.

Returns:

Estimate of market noise indicators showing how each of the three different number of records influences results.



Market Noise Function with 3 different Period Lengths.

Returns:

MarketNoise Indicator	
in: Records in Noise Calculation 1:	Step <input type="checkbox"/> <input type="text" value="8"/>
in: Records in Noise Calculation 2:	Step <input type="checkbox"/> <input type="text" value="34"/>
in: Records in Noise Calculation 3:	Step <input type="checkbox"/> <input type="text" value="55"/>

Market Noise Indicator

Example:

```

' -----
' UPDATE INDICATORS SCRIPT - START
' ~~~~~
' Delay Calculations until enough records are available
If instrument.bar > iBarCount1 THEN
' Max absolute deviation from the momentum trend line - IPV.
aPlotMktNoise1 = MarketNoise(instrument.close, iBarCount1 )
ENDIF
' Delay Calculations until enough records are available
If instrument.bar > iBarCount2 THEN
' Max absolute deviation from the momentum trend line - IPV.
aPlotMktNoise2 = MarketNoise(instrument.close, iBarCount2 )
ENDIF
' Delay Calculations until enough records are available
If instrument.bar > iBarCount3 THEN
' Max absolute deviation from the momentum trend line - IPV.
aPlotMktNoise3 = MarketNoise(instrument.close, iBarCount3 )
ENDIF
' -----
' UPDATE INDICATORS SCRIPT - END
' -----

```

Links:

[Historic Volatility](#), [MedianAbsoluteDeviation](#)

See Also:

[Indicator Pack 1 Series Functions](#)

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:**Description:**

bars

The number of bars over which to find the median absolute deviation

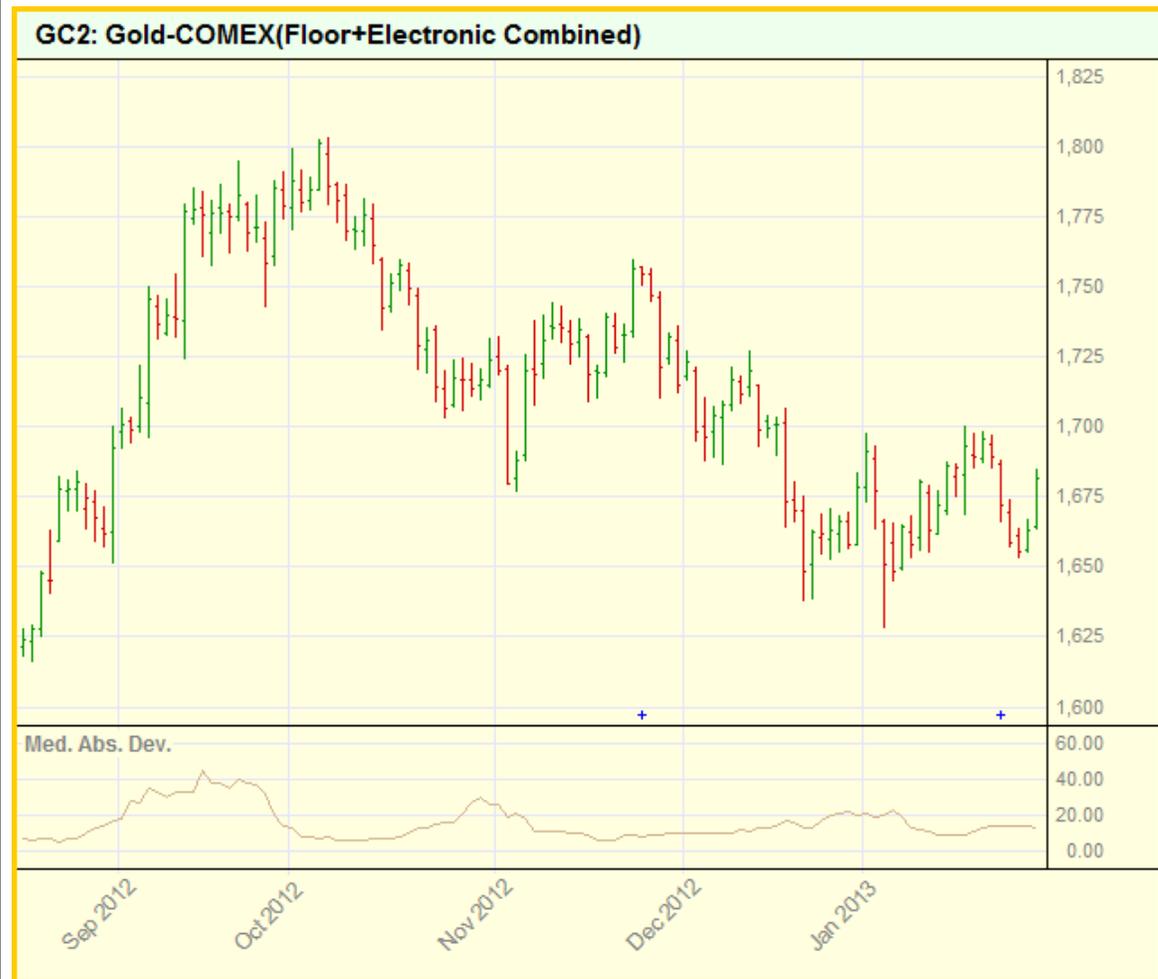
Returns:

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:

Example:

Median Absolute Deviation Indicator

Links:[Historic Volatility](#), [Market Noise](#)**See Also:**[Indicator Pack 1 Series Functions](#)

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:

Edit Indicator

Name for Code:

Type:

Value:

Time Frame:

Lookback:

Not Applicable:

Not Applicable:

Chart Display:

Example:**Links:****See Also:**

MRO

Most Recent Occurrence (**MRO**) returns the numbers of bars back that the condition was true.

If the condition was not true during search back of the previous bars, the function will return a value = **-1** = (**FALSE**).

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.

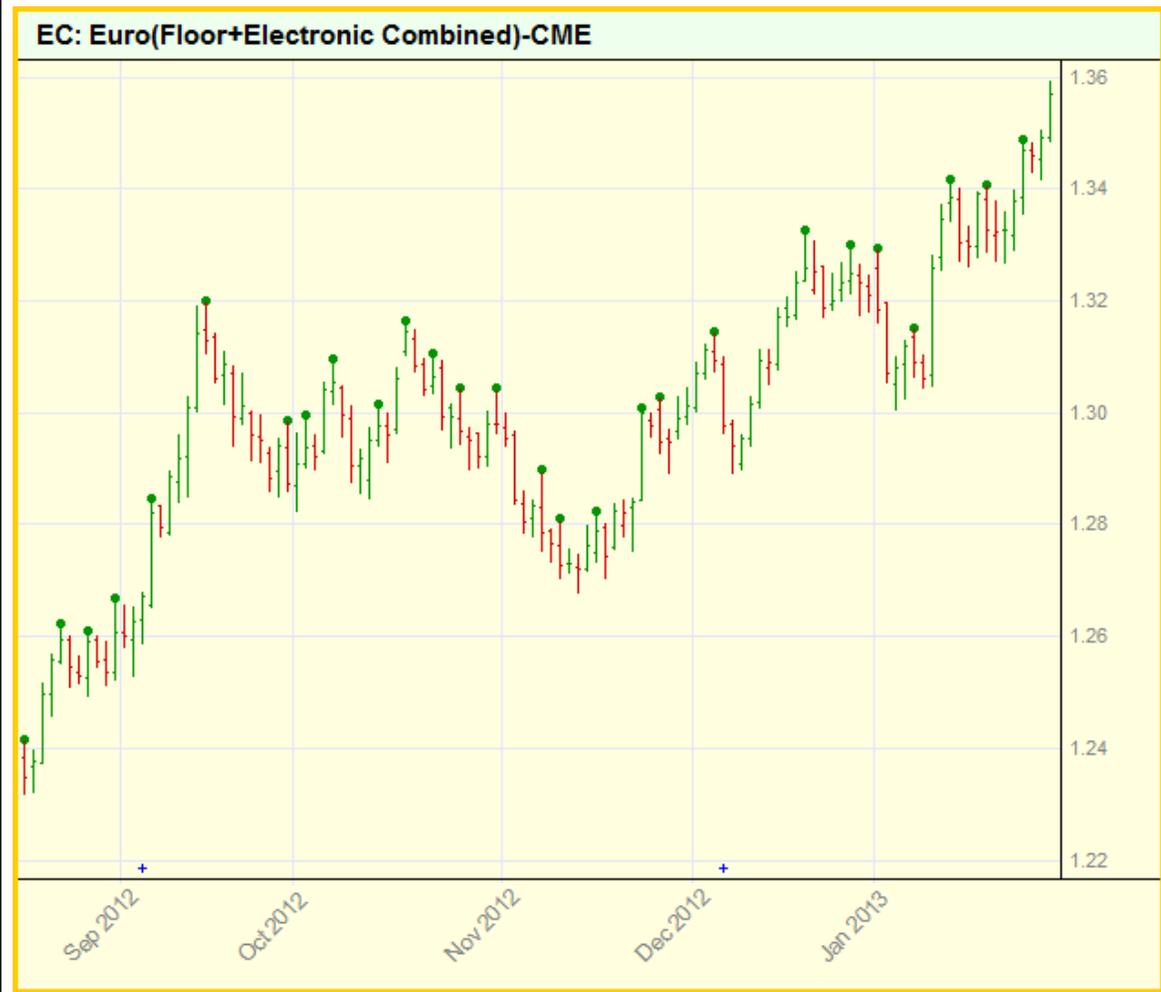
Returns:

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:

Example:

Most Recent Indicator where a condition was True

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
percentile Level	The percentage level used to find the percentile. Specified as a value between 0 and 100, representing 0-100%.

Returns:

Percentile estimate ranked value.

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:

```

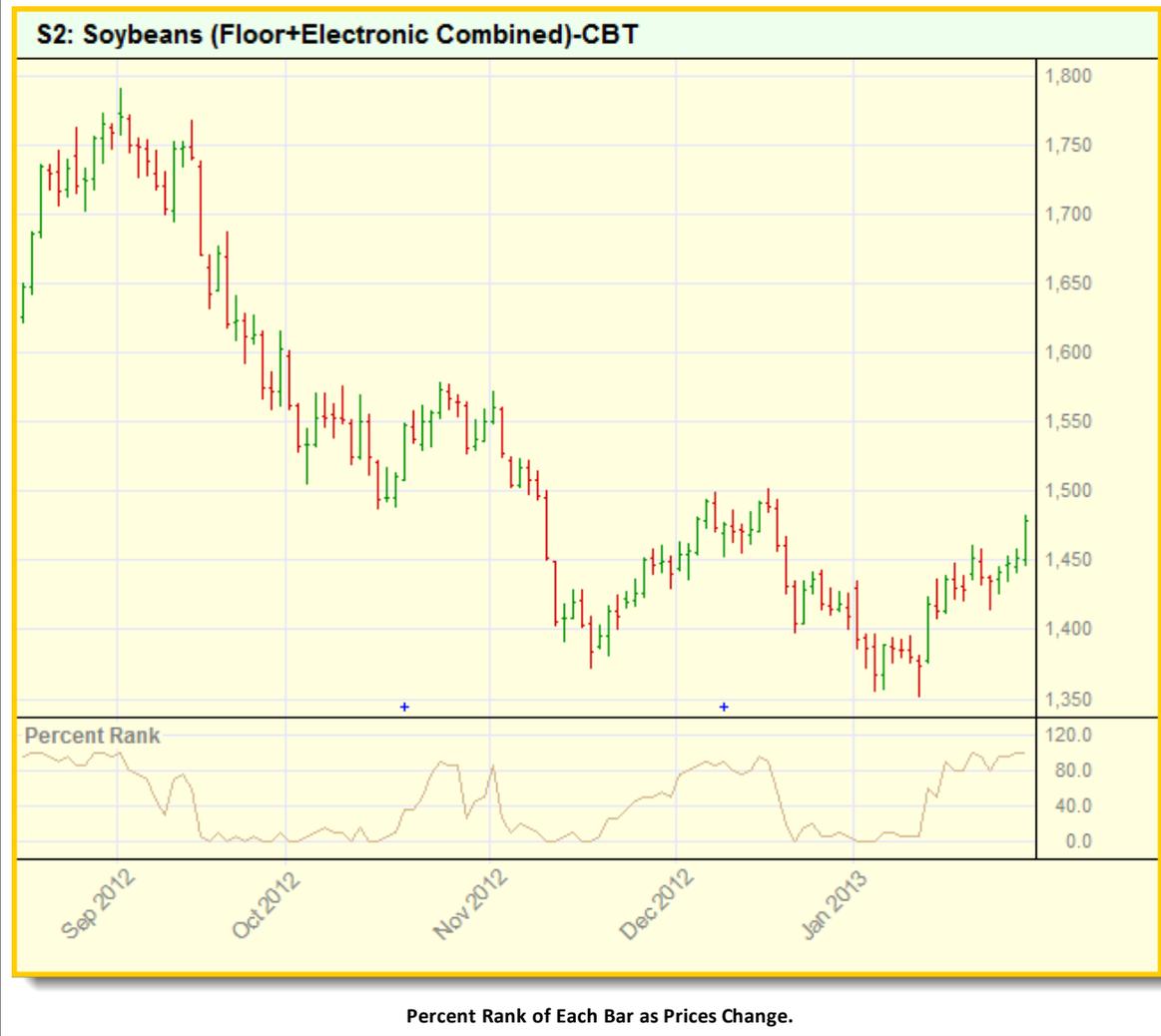
' ~~~~~
' Update Indicators Script
' ~~~~~
' Bar_Count = 21 - Parameter - Lookback enabled
' Calculates percentile ranking of a value in a price series period
Percent_Rank_Close = PercentRank( instrument.close , Bar_Count )
' ~~~~~

```

OR:

Edit Indicator	
Name for Code	Percent_Rank_Close
Type	Percent Rank
Value	Close
Time Frame	Bar
Period	Bar_Count
Not Applicable	Not Applicable
Not Applicable	Not Applicable

Chart Display:

Example:**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:

Edit Indicator

Name for Code:

Type:

Value:

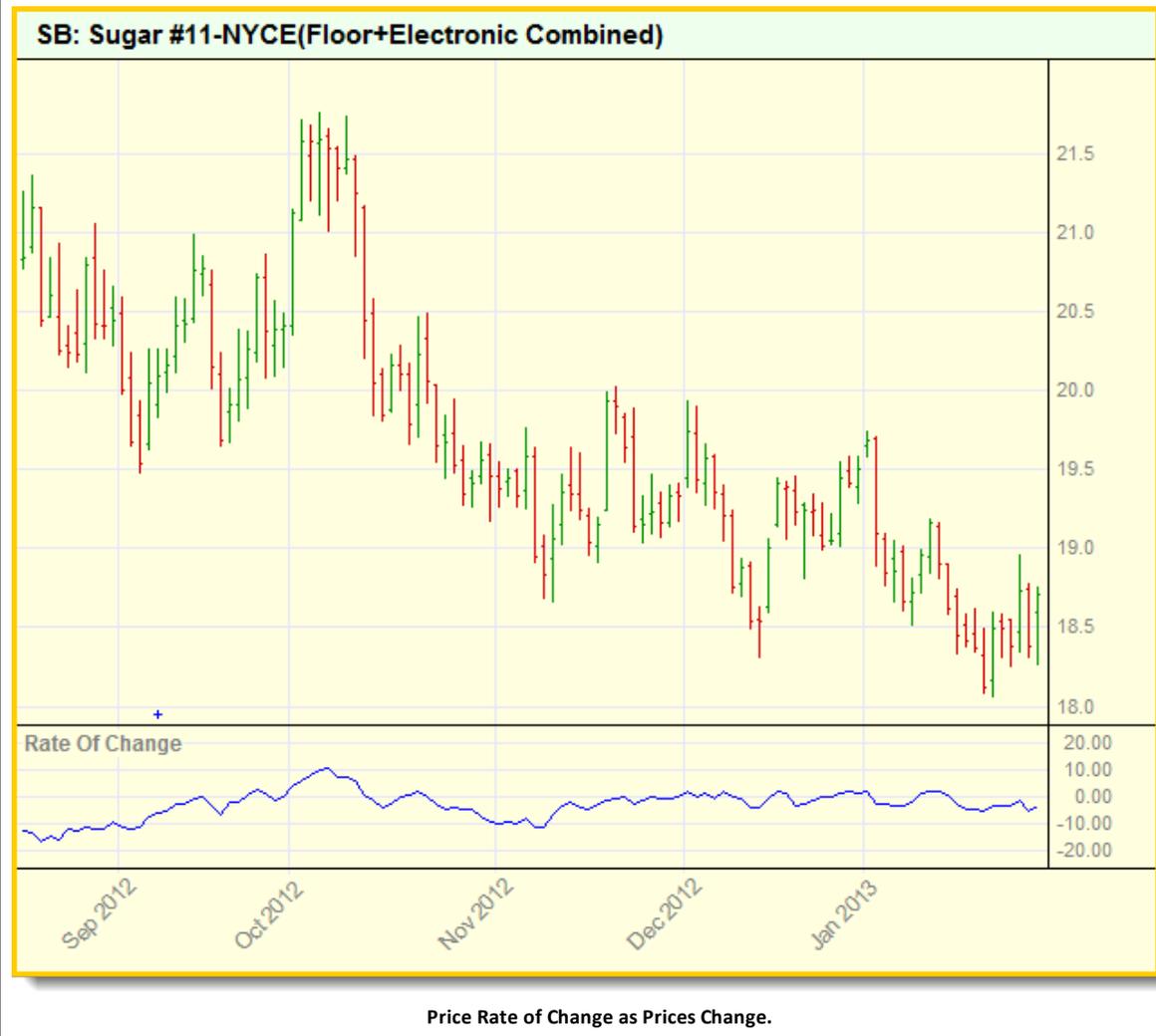
Time Frame:

Lookback:

Not Applicable:

Not Applicable:

Chart Display:

Example:**Links:****See Also:**

SpearmanCorrelation

Spearman correlation is an alternative to the Pearsons correlation and correlates based on the ranking of the series values.

This [Wikipedia link](#) provides a good explanation of how this statical function analyzes information.

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.

This [Wikipedia link](#) provides a good explanation of how this statical function analyzes information.

Syntax:
<code>SpearmanCorrelationSync(series1, series2, bars)</code>

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.

This [Wikipedia link](#) provides a good explanation of how this statical function analyzes information.

Syntax:	
<code>SpearmanLogCorrelation(series1, series2, bars)</code>	

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 .

This [Wikipedia link](#) provides a good explanation of how this statical function analyzes information.

Use this function when an IPV Auto Indexed Series, or Price Series, and the dates in each series can be synchronized before computing series 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:

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.

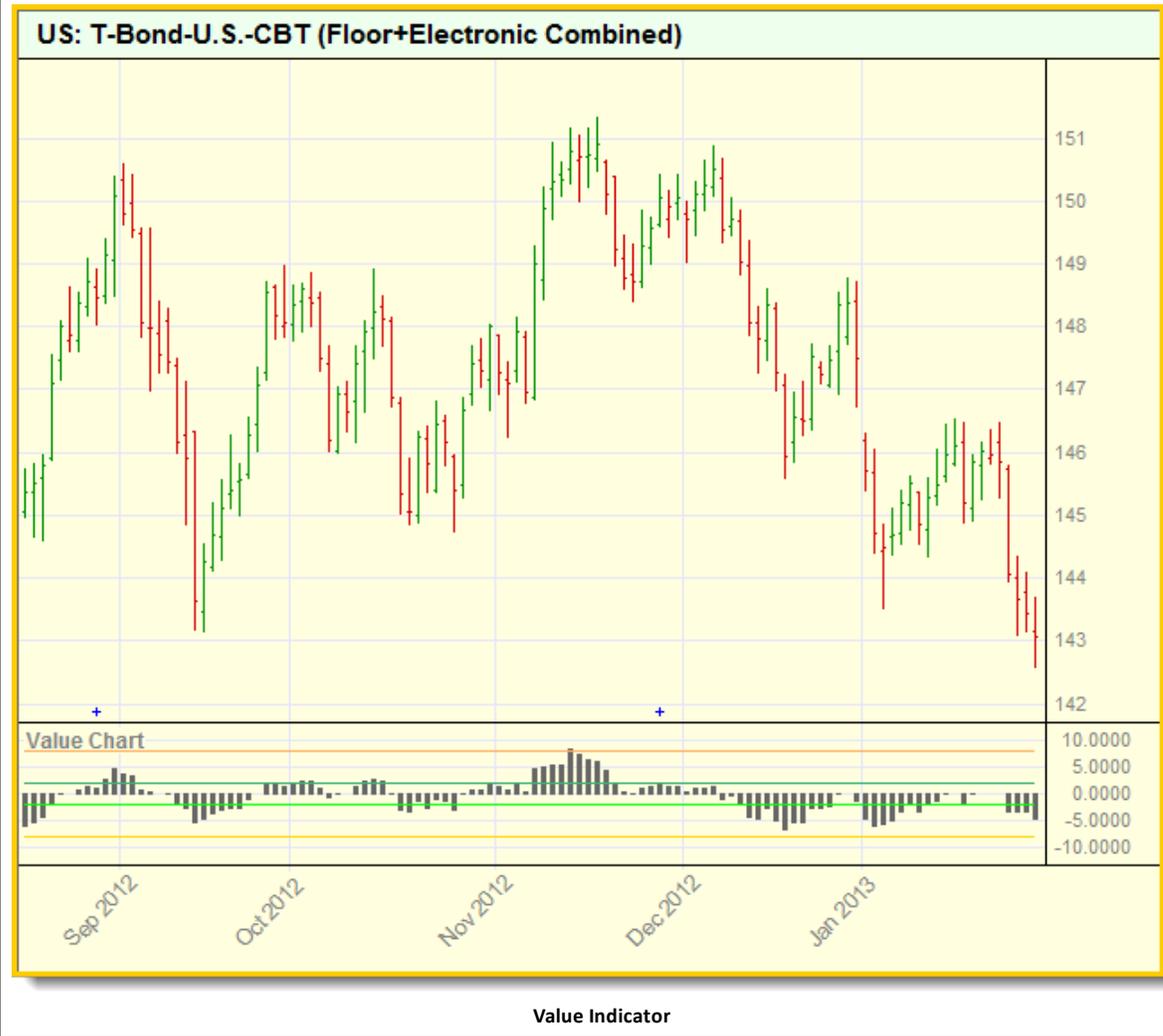
Example:

```
' ~~~~~
' 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, BarsCount)
' ~~~~~
```

Chart Display:

Example:



Links:

[Numeric Series](#)

See Also:

[Data Group and Types](#)

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 Weighed Moving Avg over price period length.
WMA_Close = WMA( instrument.close , Bar_Count )
' ~~~~~

```

OR:

Edit Indicator

Name for Code:

Type:

Value:

Time Frame:

Period:

Not Applicable:

Not Applicable:

Chart Display:

Example:**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:

New Indicator

Name for Code: ZScoreClose

Type: ZScore

Value: Close

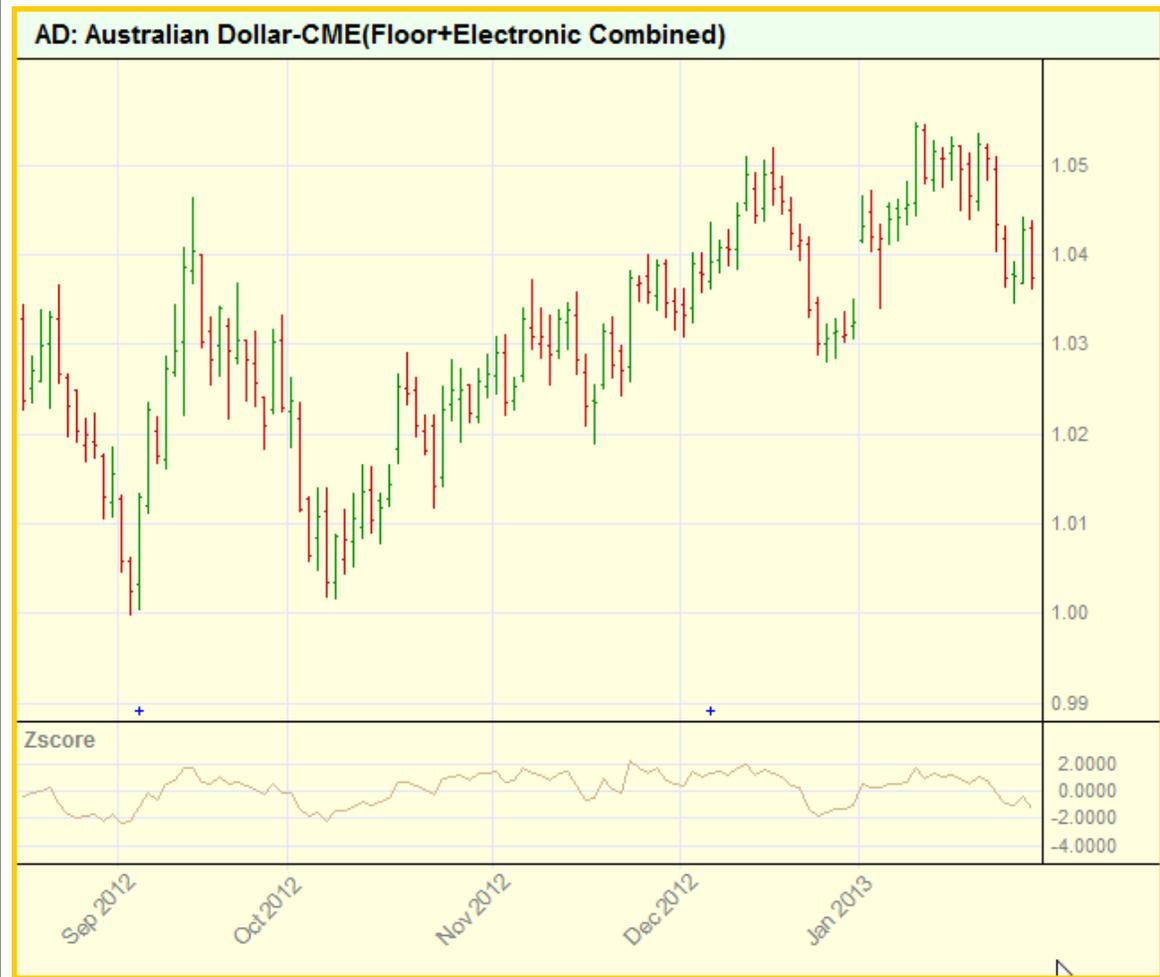
Time Frame: Bar

Period: Bar_Count

Not Applicable

Not Applicable

Chart Display:

Example:

Z-Score indicator values at each price change location.

Links:**See Also:**

Section 5 – Indicator Reference

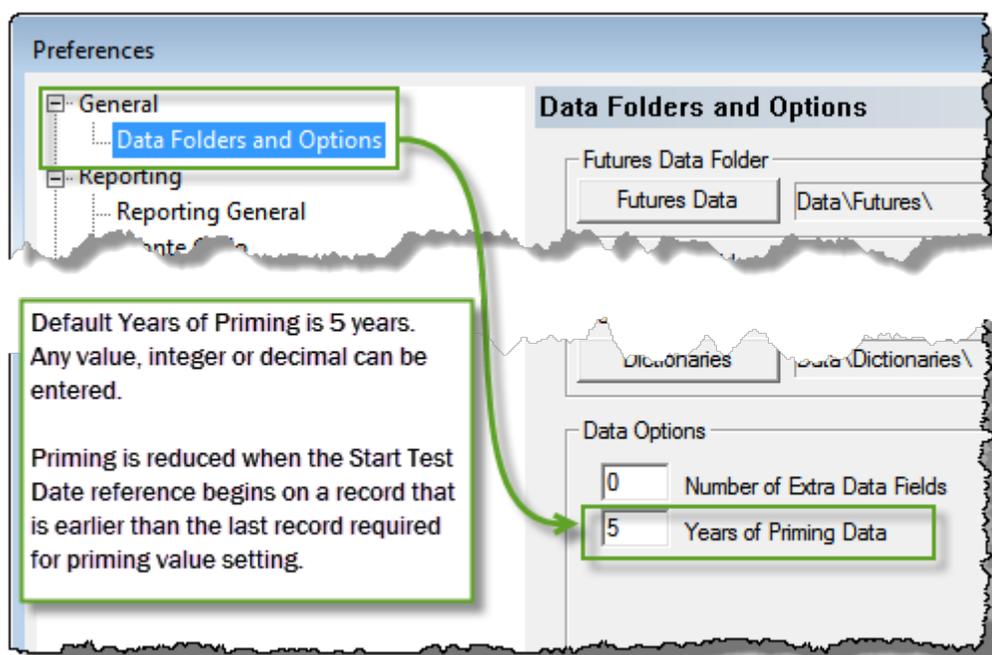
You can use a variety of indicators in a trading system. Trading Blox provides a long list of pre-built calculation methods that can be easily selected and used as either a standard indicator for display or use in a system, or as a value to a calculated or custom indicator you create for your requirements.

Built-in calculation methods used with [Basic Indicators](#) and [Calculated Indicators](#) can be used in scripts, and they can also be enabled to be plotted with the trading data.

Both of these types of indicators are pre-computed before a test simulation begins testing. Calculations performed for these indicators will use the first available record loaded and all the records in an instrument file to create a value for each instrument record so they are available ahead of testing.

Types of indicators:	Description:
Basic Indicators	Use a Selected built-in formula from the Indicator Wizard Dialog.
Calculated Indicators	Take an expression. These can only use pre-test start 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	Indicator Packs are provided as an add-on list of built-in indicator methods. Indicator Pack_1 is included with Trading Blox automatically. Calculation methods contained with an indicator pack will be included in alphabetical order in the same drop down list all the previous built-in calculation methods. Indicator Pack_1 methods can be used in the same way as all other indicators.

First available load record available for pre-calculation processing is dependent upon the simulations established Start Date and the **Preference** setting established for "Years of Priming" field:



Once the available data of each instrument the indicator's selected built-in calculation will create a value for the entire length of the indicator.

1. To create a new Basic or Calculated indicator, click on the **Indicator** 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.

Links:

[Basic Indicators](#), [Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Group and Types](#), [Indicator Pack 1 Indicators](#)

5.1 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.

An addition list of built-in indicator calculations are listed here: [Indicator Pack 1 Indicators](#)

Creating an indicator:

- Examine the fields in the dialog that are colored with a white background.
- Enter, or change the value shown where necessary.
- Decide if the indicator will be appear as a indicator on the price chart.

Indicator Names:	Descriptions:
Accumulation / Distribution	Volume-based momentum indicator
ADX-Avg. 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 "
Bollinger Bands	This indicator uses the last name of John Bollinger. This indicator is a pair of lines that are created by adding or removing the a multiple of the standard deviation of the Close price over a period of prices.
John Bollinger 's description of his indicator is available at this link location:	Bollinger Lower - Is the Lower channel of the Bollinger band
Read the Rules for Trading with Bollinger Bands:	Bollinger Upper - Is Upper channel of the Bollinger band. Uses a Standard Deviation based channel width from an EMA (Exponential Moving Average). 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

Indicator Names:	Descriptions:
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-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

Links:

[Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

Links:

See Also:

[Data Groups and Types](#), [Indicator Pack 1 Indicators](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 160

5.2 Calculated Indicators

Calculated Indicators:

Calculated indicators are calculated in the Before Test script section runs. This is done so all the indicator 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: OK

Type: Cancel

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 B:

Display Value

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.

Links:

[Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

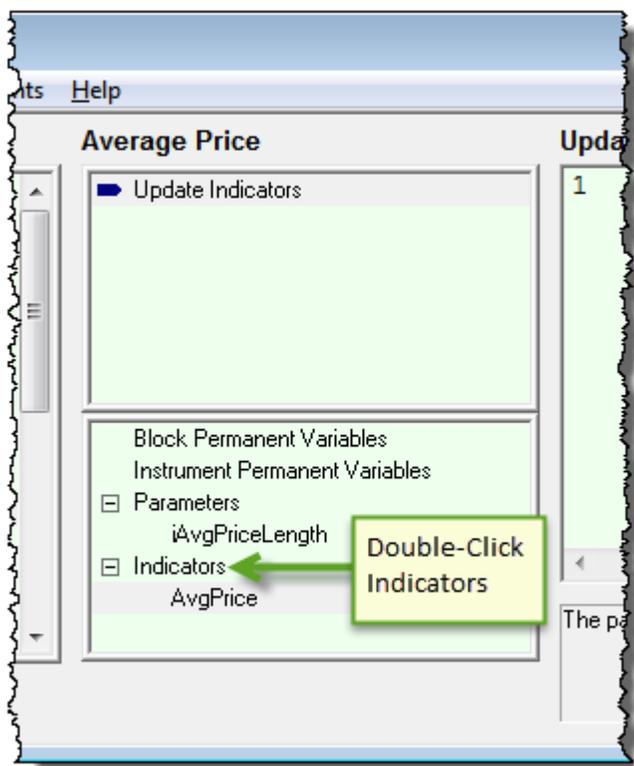
See Also:

[Data Groups and Types](#), [Indicator Pack 1 Indicators](#)

5.3 Creating Indicators

Built-in indicator methods provide a quick means for creating a display or calculated value that can be displayed on a chart or used in the script. Most indicators use parameters, but some don't require any. Before creating an indicator preview your intended indicator to determine if that calculation will require one or more parameters. When the indicator does require a parameter creation details can be reviewed here:

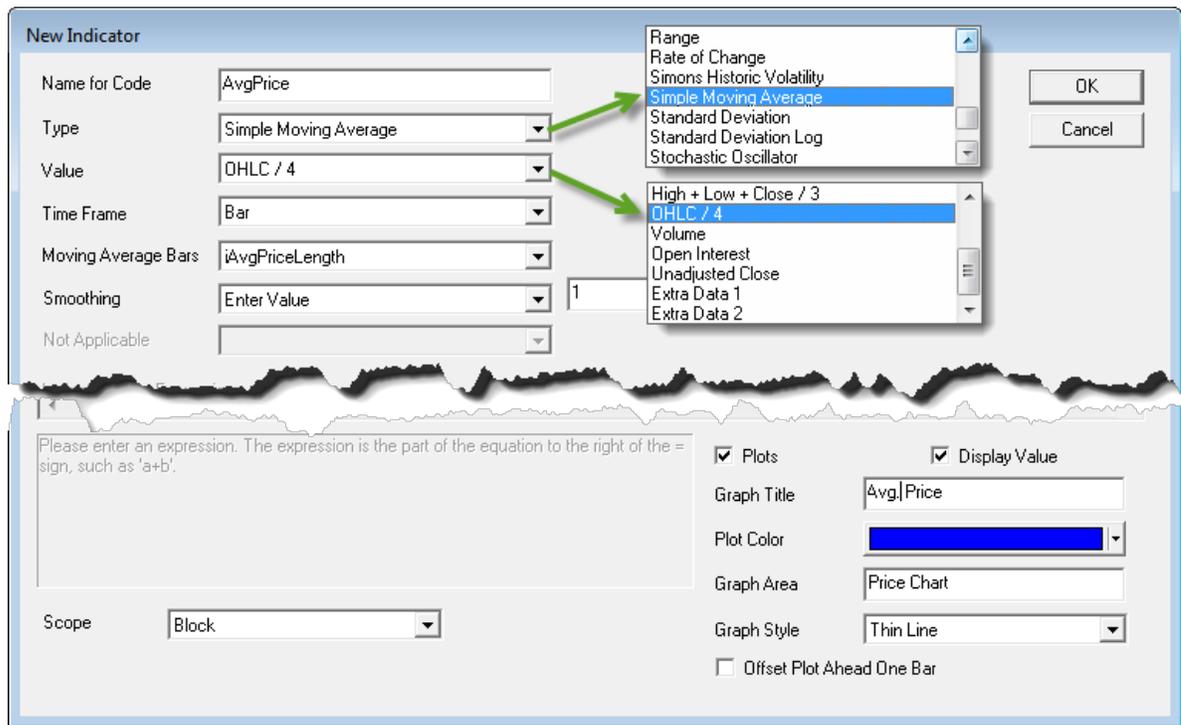
To create an Indicator in the Blox Editor, select the Indicators item and right click or use the Items menu. This will bring up the new indicator dialog:



Built-In Indicator Open Wizard Dialog

When the dialog shown here appears:

- Enter your indicators name
- Select a built-in calculation method
- Select a data record value to use in the built-in calculation
- Select the parameter property setting created to adjust the built-in calculation
- If the indicator is to plot on the chart, give it a name, select a color, and enter a name if the plot is to appear below the price display.



Built-In Indicator Open Wizard Dialog

Naming 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:

Value 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
User Created Indicators	<p>When an indicator is created with Wizard Indicator tool, the names of the other indicators listed in the indicator list will be displayed in the Value section's drop-down list available values. Any of the names listed in the Value's list of names are available for an indicator.</p> 

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 look-back parameter plus one.

Links:

[Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Groups and Types](#), [Indicator Pack 1 Indicators](#)

5.4 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.

Links:

[Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Groups and Types](#), [Indicator Pack 1 Indicators](#)

5.5 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 IPV Data:

One way to access indicators in other blox in your system is to set one of the IPV variables to show a System [Data Scope](#) setting. In the other blocks in the system where this IPV variable will be accessed, create a the IPV variable name and then check Defined Elsewhere. This process enables IPV variables to share information anywhere in the system where the same IPV variable name is declared in a block.

Here is an example:

Instrument Permanent Variable

Name for Code

Name for Humans

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

Now you can use this `averageClose` variable in the block and the value will be consistent across the whole system.

Links:

[Calculated Indicators](#), [Creating Indicators](#), [Custom Indicators](#), [Indicator Access](#), [Indicator Reference](#)

See Also:

[Data Groups and Types](#), [Indicator Pack 1 Indicators](#)

5.6 Indicator Bar to Week Time Frame

It is possible to easily convert indicators from using daily data price records to weekly price records when using the indicator is one of the built-in indicators.

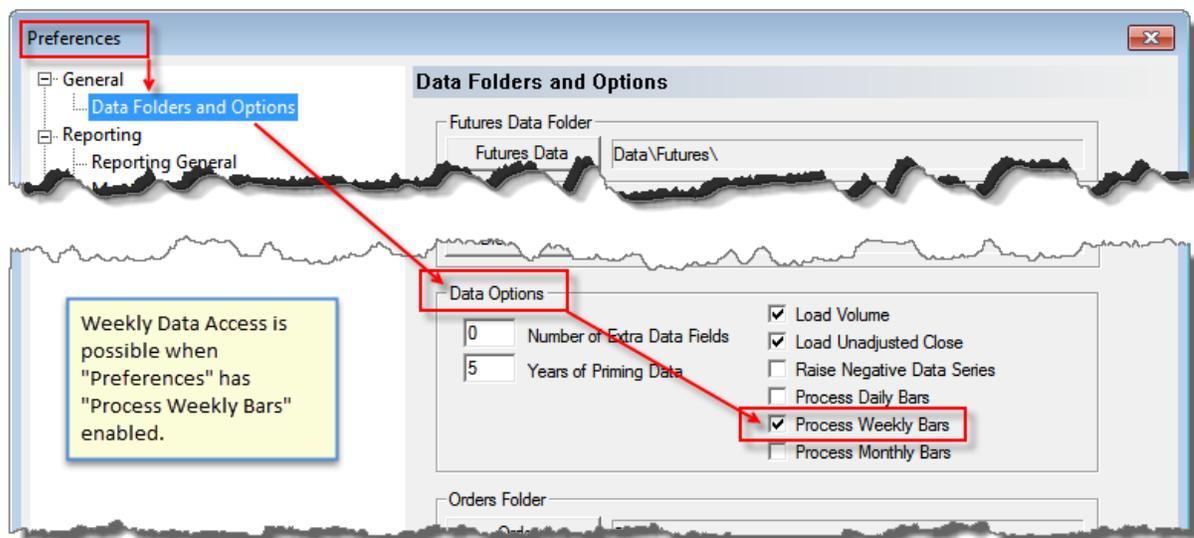
In the Trading Blox default **Dual-Moving Average Entry Exit** Blox use of the built-in Average function provides options for the data's time frame. Average function allows a user to change the time from a smaller to a larger time period (i.e. Daily to Weekly) by clicking on the **Time-Frame** drop down selection option.

By default the **BAR** reference in the built-in **Average** function dialog refers to the data period of the data file. Setting the Time-Frame in the indicator to the Bar option the Average function will use the time frame of records in the data file. In this case the Average function was created to use daily data. By selecting the Bar option the Average indicator will use the file's daily data in its calculations and plot display.

A change in the Time-Frame reference to **Week** when the data file has daily data record allows the Average function to use the optional Weekly records that Trading Blox creates when it **Data Option** section in **Preferences** show **Process Weekly Bars** is enabled.

Changing Daily Price Plots to Week Prices Plots:

Start by ensuring the Preference section shows the data option will "Process Week Bars."



Make a copy of the default blox that needs to be converted:

Group: Entry Exit

- Auxiliary
- Average Price
- Entry
- Entry Exit
 - ADX Entry Exit
 - ATR Channel Breakout
 - Bollinger Breakout
 - Bollinger Counter Trend Entry Exit
 - Donchian Smart Fills
 - Dual Moving Average Entry Exit
 - Dual Weekly Mov. Avg Entry Exit
 - MACD Entry Exit
 - RSI Trend

Dual Weekly Mov. Avg Entry Exit

Before Simulation
Before Test
Before Trading Day
Open
Exit Order Filled
Adjust Stops
After Instrument Day
After Trading Day
Unit Size
After Test

Entry Orders

```

1  VARIABLES:
2
3  ' Get the c
4  currentClos
5
6  ' If we are
7  IF instrum
8  shortMo
9
10 ' If we
11 IF useA
12 ' C

```

FROM

TO

Indicators

- longMovingAverage
- shortMovingAverage
- averageTrueRange

riskPerTrade

```

32 stop
33
34 ' E
35 bro
36
37 ' NOT u
38 ELSE

```

Open each indicator and change the Time Frame setting from Bar to Week:

Edit Indicator

Name for Code: longMovingAverage

Type: Exponential Moving Average

Value: Close

Time Frame: Week

Bar
Day
Week
Month

OK
Cancel

Please enter an expression. The expression is the part of the equation to the right of the = sign, such as 'a+b'.

Plots

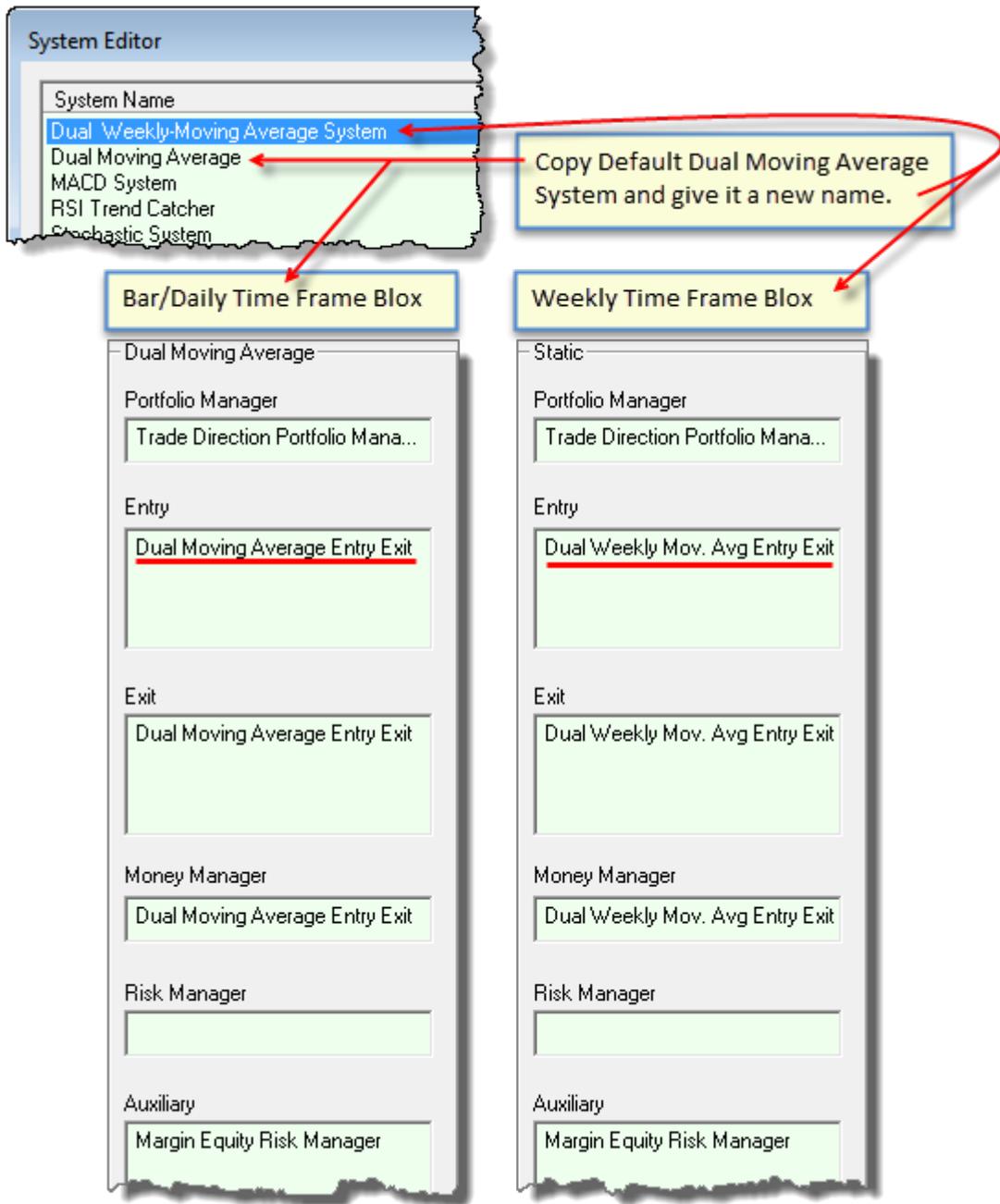
Display Value

Graph Title: Long Moving Average

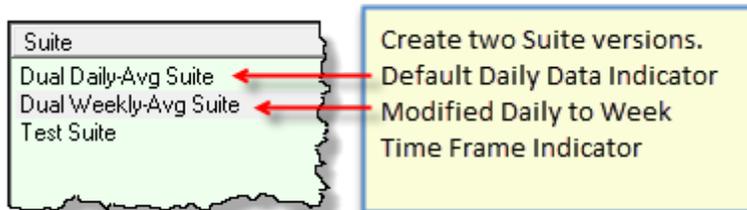
Plot Color: [Blue]

Graph Axis: Price Chart

Create a copy of the default system component list using a different name:



Create a Daily-Avg Suite and a Week Avg Suite:



Both Suites Use Same Daily Data Files

Daily Price Calculated Example:



Weekly Price Calculated Example:**Links:**

[Basic Indicators](#), [Calculated Indicators](#), [Creating Indicators](#), [Indicator Reference](#)

See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 379

Section 6 – 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 %	<p>MOD and % are modulus, or remainder operators. Result returned is a remainder value between 0 and the Absolute(value) of the number being divided. Sign of the resulting value is the same as the sign of the number being divided.</p> <p>The arguments to the modulus operator may be floating-point numbers, so that 5.6 % 0.5 returns 0.1. Both operators will return the same result. Value returned by each is the remainder from the division. The left and right expressions can be floating or integer. The results will be float or int depending on the values used.</p> <p>Examples:</p> <pre>test.currentDate = ,2015-01-01, DateToJulian(test.currentDate) ,42004, iJulianDate % 7 = ,4, 42004 % 7 = ,4, DayOfWeek(test.currentDate) = ,4, DateToJulian(test.currentDate) MOD 7 = ,4, iJulianDate MOD 7 = ,4,</pre> <p>Operator Comparison:</p>

Operator Symbol:	Description:
	<pre>Print 1 % 1 0 Print 1 % 2 1 Print 1 % 3 1 Print 1 MOD 1 0 Print 1 MOD 2 1 Print 1 MOD 3 1</pre>
>	<p>Greater than symbol used in a conditional reference test where a True or a False is needed.</p> <pre>1 > 2 = True, 2 > 1 = False</pre>
<	<p>Less than symbol used in a conditional reference test where a True or a False is needed.</p> <pre>1 < 2 = True, 2 < 1 = False</pre>
<> OR !=	<p>Both symbol pairs are used when the values on either side are Not True.</p> <pre>1 <> 2 = True, 2 <> 2 = False, 4 != 3 = True</pre>

Logical Operators (AND NOT OR):

Operator Names:	Description:
AND	<p>Performs compound result of <u>TRUE</u>, if, and only if, all of its components are true. Result is always an Integer value.</p> <p>' Expression result will return either a TRUE or a FALSE value. (expression1 AND expression2) = [TRUE or FALSE Based on how variables compare]</p> <p><u>TRUE</u> will be returned when each variables are the same value</p> <p><u>FALSE</u> will be returned when each value are NOT the same.</p>
NOT	<p>Performs logical negation of an expression. Result is always an Integer value.</p> <p>Applying this operator to a variable will always returns a negated Integer value.</p> <p>' Example: If Value is equal to 1, return will be: PRINT value value = 1</p> <p>' Example: If Value is equal to 1, return will be: PRINT NOT value NOT value = 0</p>

Operator Names:	Description:
OR	<p>Performs a logical disjunction on two expressions. Result is always an Integer value.</p> <p>' Expression result will return either a TRUE or a FALSE value. (expression1 OR expression2) = [TRUE or FALSE Based on how variable compare]</p> <p><u>TRUE</u> will be returned when either variables is <u>TRUE</u>.</p> <p><u>FALSE</u> will be returned when both values are <u>FALSE</u>.</p>

Example - 0:

```
' ~~~~~
' 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
  PRINT "It is TRUE, Do something"
Else
  ' Do something else
  PRINT "It is FALSE, Do something else"
EndIf
' ~~~~~
```

Example - 1:

```
' ~~~~~
If NOT condition1 THEN
  PRINT "condition1 = FALSE", condition1
ELSE
  PRINT "condition1 = TRUE", condition1
ENDIF
' ~~~~~
```

Returns - 1:

```
condition1 = TRUE 1
```

Example - 2:

```
' ~~~~~
condition2 = FALSE
If NOT condition2 THEN
  PRINT "condition2 = FALSE", condition2
ELSE
  PRINT "condition2 = TRUE", condition2
ENDIF
' ~~~~~
```

Returns - 2:

```
condition2 = FALSE 0
```

Links:

[Constants Reference](#), [Data Groups and Types](#)

See Also:

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 446

6.1 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.

Keyword Syntax:	Description:
=	Equivalent to, or the value the element on each side is exactly the same. <code>IF (var1 = var2) THEN ...</code>
!= <>	Values on each side of the symbol are Not equivalent to, or are not the exactly the same value. Both symbol characters are valid to use.
>	Greater than symbol.
<	Less than symbol.
<=	Less than or equal to.
>=	Greater than or equal to

Example:

```
VARIABLES: i TYPE: Integer, var1 TYPE: String

WHILE ( i != 10 )
    var1 = "Message for you, sir."
    i = ( i + 1 )
ENDWHILE
```

Returns:

Links:

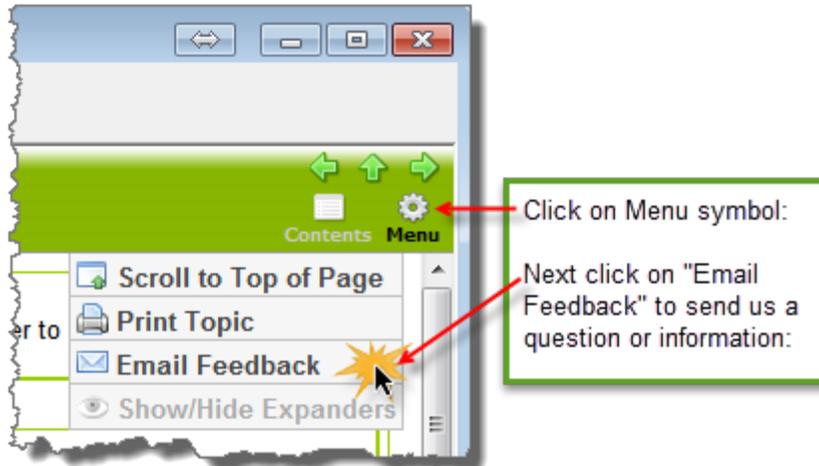
See Also:

Section 7 – Script Problem Information

This section contains information for solving script and help manual problems.

Additions to this section are dependent upon being discovery and reporting by those who find a problem.

If you find a problem that isn't referenced here, please send an email so we will be able to answer the question, or add the information to the Help File:



Script Information Topics:	Description:
Auto-Keyword Changes	Trading Blox version updates have sometimes created a need to change keywords. When that happens the update where that change is implement will automatically attempt to repair the keyword reference so the user doesn't experience a problem. Some changes cannot be handled automatically. When we know about them they will be referenced in this section so you will understand what has to be changed.
Debugger	Information on how to use the Trading Blox Editor's Debugger.
Keyword Changes	Keyword changes that have been recorded are listing in this section. Sometimes we will miss getting all of them listed, but we will list them if you report them.
Restricted Keywords	Keywords that are no longer available or are restricted to only being available to internal Trading Blox resources are referenced in this section. If you find a keyword you cannot use please report it so we can determine if it was an omission or a problem with your version.

7.1 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.

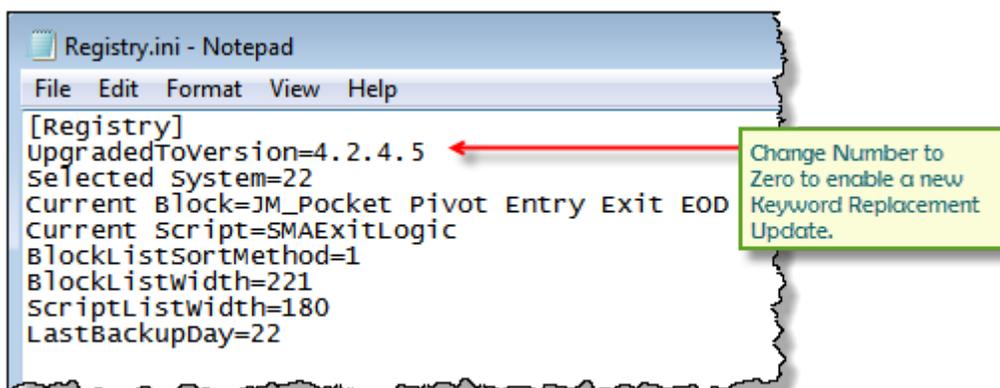
Automatic Keyword Changes:

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
```

Keyword Replacement Example:

Trading Blox will remove the above keyword AND insert this keyword:

`instrument.deliveryMonth`

Links:

[Key Word Changes](#)

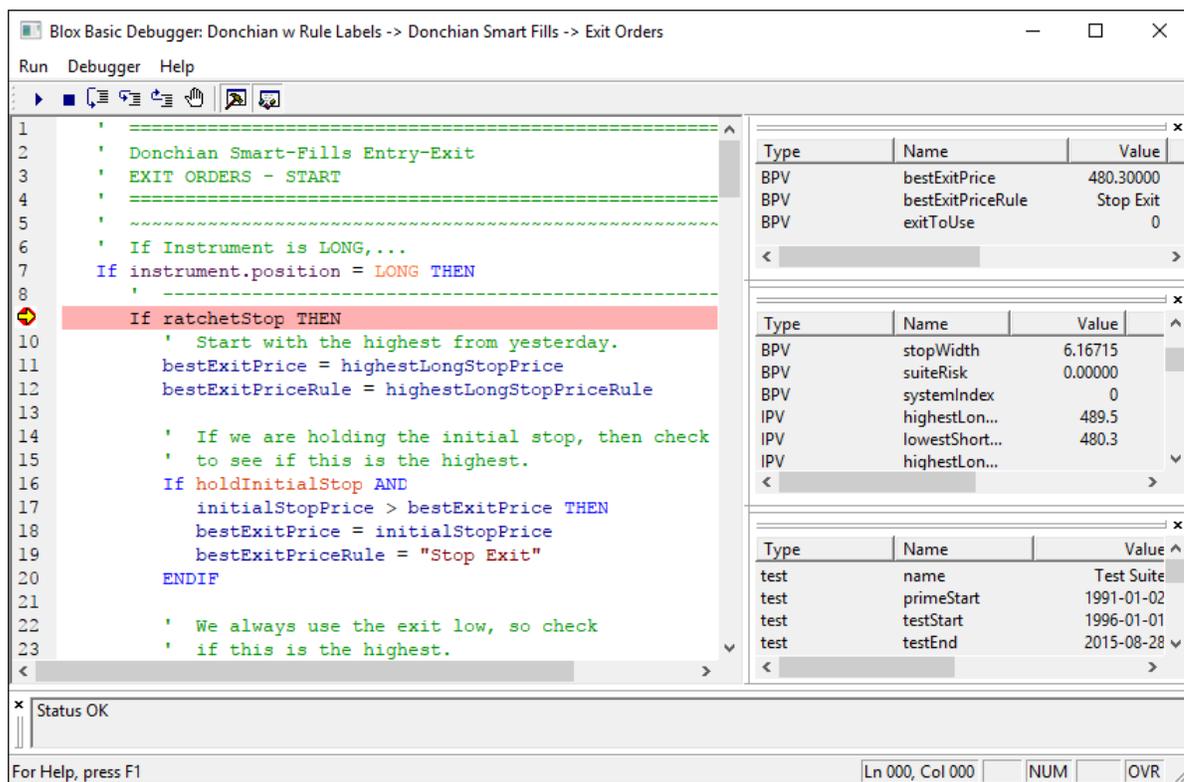
See Also:

7.2 Debugger

The Debugger is a powerful Trading Blox Builder tool for verifying and understanding blox scripts. When it is invoked, the Debugger can check each script line in each blox.

Its checking process can show each script and display the script information values in the data list on the right side of the debugger display.

Debugger's Breakpoint only work when the test is a single step test. Multiple step test disable the Debugger's Break-Point execution pauses.



Blox Basic Debugger

To invoke the debugger, enter a break point on one or more of the location where you wish to see the line by line display of script information.

When a break-point is created, close the Blox Basic Editor and run a test where the blox with the break points are included.

When the single-step test execution reaches a break point, it will stop the execution and display the break point information. To continue to the next line of script, press F11 and the debugging highlighter will move to the next line in the display script section.

F11 increments the execution one line at a time. Pressing F10, will allow execution to proceed until it reaches the bottom of the script.

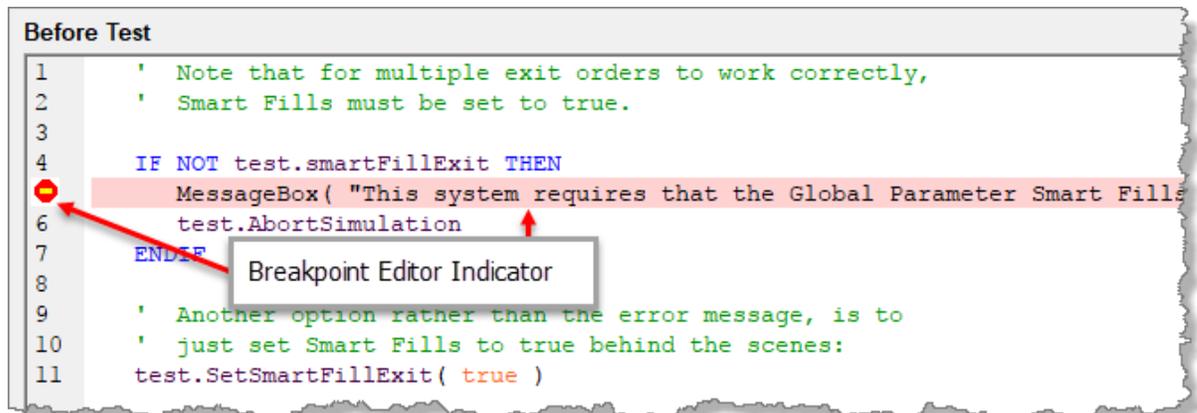
line of the script information is working and changing one line at a time, or only in the sections where a break point is active.

This is possible because the debugger exposes the variables and their values in the blox being stepped. Debugging follows the logic of the system and displays the information for both types of script context. As the script lines step through the blox, the information exposes how each script line analyzed, at the end, it proves an understand of how that blox is designed. With this understanding, you will be able to know if the blox scripts are doing what you need to happen before and after you make changes to the blox.

Break Points:

Break points create location that will halt the test execution of a system. Each break point creates one halting location. There can be many break points in a blox. When a break point halts the processing, it will exposed the values in the various types of variables and parameters in tables that can be scrolled.

Break Points can be created by clicking on the line of a script displayed in the Blox Basic Editor and pressing F9. When a break point is created it will create light-red color background of the line where the cursor is location.



```
Before Test
1      ' Note that for multiple exit orders to work correctly,
2      ' Smart Fills must be set to true.
3
4      IF NOT test.smartFillExit THEN
5      MsgBox( "This system requires that the Global Parameter Smart Fills
6      test.AbortSimulation
7      ENDEF
8
9      ' Another option rather than the error message, is to
10     ' just set Smart Fills to true behind the scenes:
11     test.SetSmartFillExit( true )
```

Block Basic Editor's Breakpoint Indicator

Pressing **F9** on a line where a break point is display will remove a break point during Blox Basic Editing display. When F9 is used to create a break point, the break point type that will be create will be the default break point that will stop execution in the debugger using the [All Instruments - Break on Every Instrument](#) option. To remove a break point, press F9 again when the cursor is on the same line as the displayed break point.

Trading Blox Builder can create other types of break points when they are created using the [Break Point Editor](#). These optional break points can be more selective in how they stop a debugging test allowing for less test-interruptions when you are trying to see what is happening for a specific instrument, or when it appears to be date related.

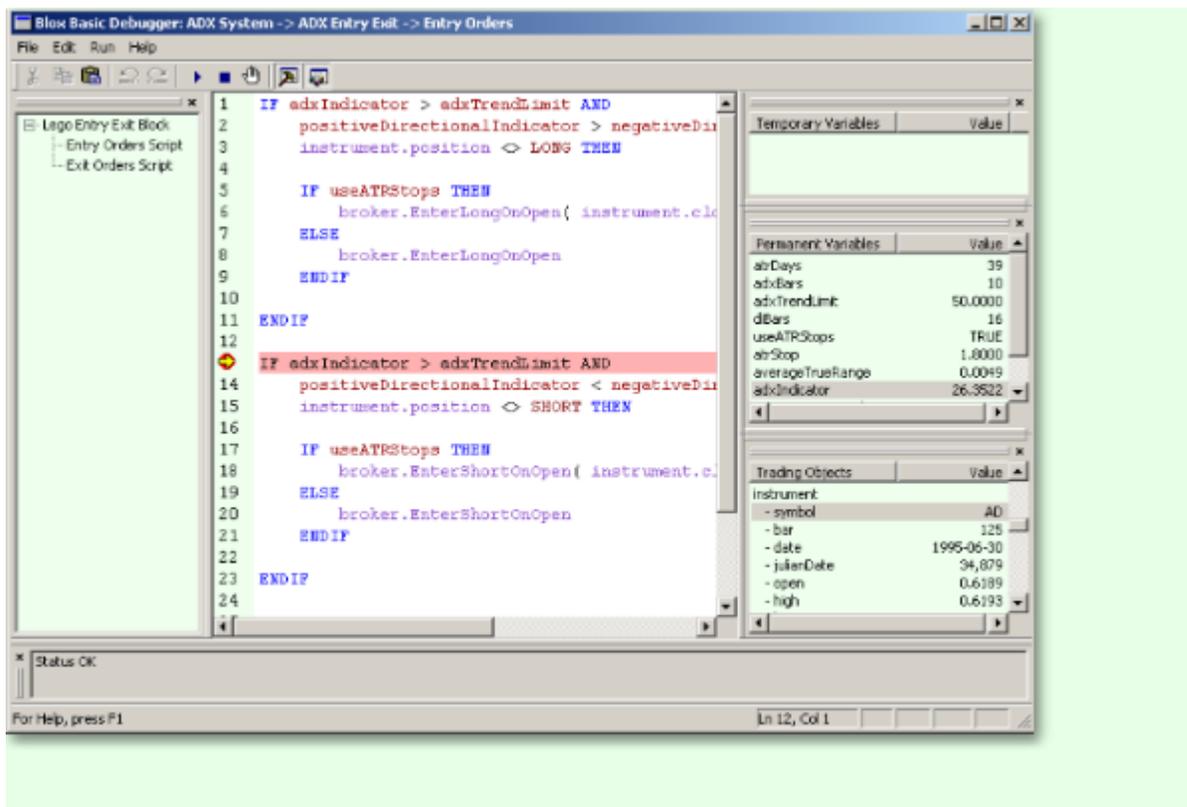
All break points can be cleared using the **Clear Breakpoints** button. When you set a breakpoint in this manner, the default is to break on every instrument on every day.

Note:

Break points are only active during a single step test. All break points are disabled during a multiple- step.

Move Editor Information to Break Point Editor Topic

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

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 400

Break Point Editor

Break Point entries can be created to use the [All Instruments - Break on Every Instrument](#) option by pressing F9. Pressing F9 again when the cursor is on the same line as the display break point line will remove the break point.

To enable one of the other break point options, use the Break Point Editor:

Breakpoint Editor

During a debugging session a break point can be temporarily suspended by pressing F9 when it stops a test at that break point location. Temporary break point suspensions only last as long as the current test is running. At the end of the test, all suspended break points are enabled again.

Click to review the Debugger [Short-Cut Key Table](#).

Instrument Restriction: ^Top	Descriptions:
All Instruments: ^Top	This option will break to automatic script execution all the instrument in the portfolio. This break point can be enabled directly in the Blox Basic Editor by pressing F9. Pressing F9 again will remove it when the cursor is on the same line as the break point to be removed.
Single Instrument: ^Top	Use this option when only one instrument symbol should stop at a breakpoint for the entered symbol. Instrument Type: When this option is used, enable the type of instrument that needs to stop at the breakpoint: <input type="radio"/> Futures

Instrument Restriction: ^Top	Descriptions:
	<ul style="list-style-type: none"> ○ Stock ○ Forex

Date Restriction: ^Top	Descriptions:
All Dates: ^Top	Breakpoints will halt execution for examination whenever this breakpoint is encountered.
Exact Date: ^Top	This open established a date when execution will halt for examination. For example, Break only on 120030415 - Enter the date in: YYYYMMDD format.
Date Range: ^Top	A date range needs two dates. Left most field requires the date with the first halt is execution will begin. The second date requires the date when the halting of execution at this breakpoint will stop. For example, start the first break on: 119000101 and ignore the break on and after 20990101 .

7.3 Key Word Changes

Keyword listed in this section show the old and new names of keywords that have been changed.

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>
3.0 Previous KeyWord	Replacement KeyWord
<code>GetStringField</code> <code>GetNumberField</code>	<code>GetField</code>

2.1 Previous KeyWord	Replacement KeyWord
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>	<code>system.totalPositions</code>
<code>system.tradingBars</code>	<code>system.dataLoadedBars</code>
<code>script.SetStringReturnValue</code>	<code>script.SetReturnValue</code>
Removed Keywords:	
<code>ConvertHtmlToMht</code>	Automatically handled by Trading Blox Builder
<code>today</code>	No longer available. Use bar reference value: 0 or leave brackets empty.
<code>yesterday</code>	No longer available. Use bar reference value: 1

Links:
Data Variable Names
See Also:

7.4 Restricted Keywords

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. There are also 1485 Keywords that are in the Trading Blox Basic Language. That number usually increases a few times a year. If you find a word you entered and the parser complains, it is likely the word is reserved.

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

abs	long
Ab	loop
so	lowerCase
lut	ltrim
eV	max
al	mid
ue	middleCharacters
	min
acos	mod
and	monthNumber
ArcCosine	newPosition
ArcSine	next
ArcTangent	not
ArcTangentXY	or
asc	order
ascii	out
asciiToCharacters	pi
asFloating	print
asin	RadiansToDegrees
asInteger	random
asString	right
atan	rightCharacters
atan2	rtrim
beep	short
block	sin
broker	sine
CanAddUnit	sqr
chr	sqr
clearscreen	squareRoot
cos	step
cosine	stringLength
dayOfMonth	sub
DegreesToRadian	system

s	ta
degtorad	n
do	tangent
else	test
endfunction	then
endif	to
endsub	toJulian
endwhile	totalRisk
entryPrice	trim
exp	trimLeftSpaces
exponent	trimRightCharacters
false	trimRightSpaces
findString	trimSpaces
findString	true
for	type
function	ucase
goto	unitSize
hypot	until
hypot	upperCase
hypotenuse	variables
if	variables
index	while
instrument	xor
isFloating	
isInteger	
isString	
lcase	
left	
leftCharacters	
len	
log	
log10	

Links:[Data Variable Names](#)**See Also:**

Section 8 – 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:
Assignme nt	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

8.1 Assignment

A [variable](#) is assigned a new value with an Assignment statement.

Syntax:
<code>variable = expression</code>

Parameter:	Description:
<code>variable</code>	An expression to be evaluated and assigned to variable.
<code>expression</code>	An expression to be evaluated and assigned to variable. Expression can be a value, the output from a function, or the value contained within a property or another variable.

Example:
<pre>' Variables should be declared before being assigned to in an assignment statement. VARIABLES: variableOne TYPE: Floating, greeting TYPE: String ' Assign the value of 50.60 to the numeric Floating type variable 'variableOne' variableOne = 50.60 ' Assign the value "Have a nice day." to the String text variable 'greeting' greeting = "Have a nice day." PRINT greeting</pre>
Returns:
greeting

Links:
See Also:
VARIABLES

8.2 DO

Repeats a block of statements while a condition is **TRUE** or until a condition becomes **TRUE**

There are few ways to use this keyword in a statement.

Syntax - Example 1:	
<pre>' 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. DO [WHILE UNTIL condition] [statements] LOOP ' 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.</pre>	
Parameter 1:	Description:
<i>condition</i>	Expression that evaluates to True or False .
<i>statements</i>	One or more statements executed while or until condition is True .

Or, you can use this syntax. This second form where the **WHILE** and **UNTIL** follow the **LOOP** keyword is 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.

Syntax - Example 2:	
<pre>' 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. DO [statements] LOOP [WHILE UNTIL condition] ' 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.</pre>	
Parameter 2:	Description:
<i>condition</i>	Expression that evaluates to True or False .
<i>statements</i>	One or more statements executed while or until condition is True .

Example:

```
' -----  
' Example 1:  
VARIABLES: a    Type: Integer  
  
DO  
  ' Print the value in the Integer variable 'a'  
  PRINT a  
  ' Increment the value of a each  
  ' time the loop is executed  
  a = ( a + 1 )  
  ' Keep executing the loop until  
  ' 'a' is equal to '11'  
Loop While ( a < 11 )  
  ' Current value of fibonacci variable  
  PRINT "Current value of variable a: ", a  
  
' -----  
' Example 2:  
VARIABLES: fibonacci, lastFibonacci Type: Integer  
  ' Initialize starting and last step values  
  fibonacci = 1  
  lastFibonacci = 1  
  
  ' Perform this loop until '' is equal  
  ' to or greater than 100  
DO UNTIL fibonacci > 100  
  ' Print out the Fibonacci number.  
  PRINT fibonacci  
  
  ' Compute the next fibonacci number  
  ' and update current 'fibonacci number  
  fibonacci = ( fibonacci + lastFibonacci )  
  ' Update previous fibonacci variable  
  lastFibonacci = fibonacci  
Loop  
  ' Current value of fibonacci variable  
  PRINT "Current value of fibonacci variable: ", fibonacci  
' -----
```

Returns:**Example 1 Output:**

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
Current value of variable a: 11
```

Example:**Example 2 Output:**

```
1
2
4
8
16
32
64
Current value of fibonacci variable: 128
```

Links:[Assignment](#)**See Also:**[VARIABLES](#)

8.3 ERROR

The ERROR statement is useful for detecting unusual conditions that you don't believe should occur. 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 without any parameters displays a blank message.
ERROR [expression, expression,...]
```

Parameter:	Description:
expression	An expression to be printed, separated by commas or semicolons.

Example:

```
' When the stopPrice is less than zero,...
IF stopPrice < 0 THEN
  ' Send an error message to the user
  ERROR "The stop price was negative ", stopPrice
ENDIF
```

Returns:

The stop price was negative: -107.87

Links:

[Debugger](#)

See Also:

[Test.AbortTest](#), [Test.AbortAllTests](#)

8.4 FOR

Repeats a group of statements a specified number of times.

First the expression *start* and *end* are evaluated. When the **NEXT** statement is encountered, *stepSize amount* is added to the variable defined as the *counter*. At this point, if counter is less than or equal to *end value*, 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 *stepSize* 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.

Syntax:
<pre>FOR counter = start TO end [STEP stepSize] [statements] NEXT</pre>

Parameter:	Description:
<i>counter</i>	Variable used as a loop counter. This variable must be declared as an integer before using.
<i>start</i>	Initial value of counter (can be a complex expression)
<i>end</i>	Final value of counter (can be a complex expression)
[STEP <i>stepSize</i>]	Integer amount counter is changed each time through the loop. If not specified, step defaults to one, and the step value will increment. Step value can be a negative that decrements to the value contained in the variable: end.
<i>statements</i>	One or more statements between FOR and NEXT that are executed the specified number of times.

Example:

```

' -----
VARIABLES: index  Type: Integer
PRINT "Example 1:"

' This For Next loop will increment index values
For index = 1 TO 5
    PRINT "index value = ", index
Next

' -----
VARIABLES: index  Type: Integer
PRINT "Example 2:"

' The For Next Loop decrements index values
For index = -1 TO -5 STEP -1
    PRINT "index value = ", index
Next

' -----
VARIABLES: index  Type: Integer
PRINT "Example 3:"

' The For Next Loop Increments index values
For index = -1 TO -5 STEP 1
    PRINT "index value = ", index
Next

' -----
VARIABLES: monthIndex, dayIndex  Type: Integer
PRINT "Example 4:"
' 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 2 STEP 1
    ' Display Month Name
    PRINT "Month: ", MonthName( monthIndex )
    ' Display the first 7 days of a Month
    For dayIndex = 1 TO 7 STEP 1
        PRINT " Day # = ", dayIndex, " Day Name = ", DayOfWeekName( dayIndex
- 1 )
    Next
Next
' -----

```

Returns:**Example 1:**

```

index value = 1
index value = 2
index value = 3
index value = 4
index value = 5

```

Example 2:

Example:

```
index value = -1
index value = -2
index value = -3
index value = -4
index value = -5
```

Example 3:

Month: Jan

```
Day # = 1 Day Name = SUNDAY
Day # = 2 Day Name = MONDAY
Day # = 3 Day Name = TUESDAY
Day # = 4 Day Name = WEDNESDAY
Day # = 5 Day Name = THURSDAY
Day # = 6 Day Name = FRIDAY
Day # = 7 Day Name = SATURDAY
```

Month: Feb

```
Day # = 1 Day Name = SUNDAY
Day # = 2 Day Name = MONDAY
Day # = 3 Day Name = TUESDAY
Day # = 4 Day Name = WEDNESDAY
Day # = 5 Day Name = THURSDAY
Day # = 6 Day Name = FRIDAY
Day # = 7 Day Name = SATURDAY
```

Links:**See Also:**[VARIABLES](#)

8.5 IF

If/THEN condition statements can execute a one or more commands, or scripting statements. Their execution is dependent upon on the conditional value agreement, or disagreement of the condition. A single command can be enabled when an expression resolves to a TRUE or False state. Depending upon how the comparison is condition is create, it can then execute or ignore the command or statement that follows the **If/THEN** conditional area.

When a single **If/THEN ENDIF** syntax is all that is created for handling a condition comparison, that type of condition is a single depends upon what can happen.

If/THEN ELSE ENDIF syntax is use, the conditional comparison will determine where area in the **If/THEN ELSE ENDIF** syntax is used.

Conditional statements can add features, or change how a process selects what happens.

Syntax:

```
' -----
' These are two primary ways an IF-THEN-ELSE process is used:
' This approach only provides changes somethings
' when the condition is TRUE
If SomeCondition = ThisValue THEN
'   Execute this next trip when the
'   condition matches ThisValue
  Make-This-script-Happen
ENDIF
' -----
' This next use will execute a process when the
' condition matches the 'ThisValue' and it will
' do something else when the value does not
' match 'ThisValue'
If SomeCondition = ThisValue THEN
'   This script section will run when the condition
'   is a match for 'ThisValue'
  Make-This-script-Happen
ELSE
'   This script section will run when the condition
'   is NOT a match for 'ThisValue'
  When-NOT-ThisValue-Make-This-Other Stuff
ENDIF
' -----
```

Parameter:	Description:

Returns:

Example:

--

Links:

--

See Also:

--

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 376

8.6 PRINT

Prints values to the **Print Output.csv** file in the **Results** folder where is installed. It will also print the information to the log window when the Log Window is active. The **Log Window** is displayed on the main screen below where a system's parameters are displayed.

Syntax:	
<code>PRINT [expression], [expression], [...]</code>	

Parameter:	Description:
<code>[expression]</code>	Optional: Any expression to be printed. When no expressions are used, the <code>PRINT</code> function will create a new blank line. When more than one expression is included, each expression must be separated by commas or semicolons.
<code>[expression]</code>	Optional: Any expression to be printed.
<code>[...]</code>	Optional: Many more expressions can be included in the statement.

Example:

```

' -----
' Create a list of positive random values
PRINT ( AbsoluteValue ( Random ( 10 ) - 100 ) )

' -----
' Print a blank line to provide space between lines.
PRINT

' -----
' Print the System Name
PRINT system.name

' -----
' Print the Suite Name
PRINT test.name

' -----
' Declare two String variables to hold text information
VARIABLES: variableOne, variableTwo Type: String
' Assign text information to each String variable
variableOne = "Hello I am "
variableTwo = "Bob"

' Print the String variable values.
PRINT variableOne, variableTwo

' Print some text information and a name.
PRINT "Don't blame me, blame ", variableTwo

' -----

```

You can also **PRINT** mathematical expressions which will be evaluated: 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.

Example:

||

Returns:

|

Links:

|

See Also:

|

8.7 WHILE

Executes a series of statements as long as a given condition is **TRUE**.

When the condition is **TRUE**, all statements are executed until the **ENDWHILE** statement is encountered. When **ENDWHILE** is reached, control then returns to the **WHILE** statement where the condition is checked again. If condition is still **TRUE**, the process is repeated. When the condition 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.

Syntax:
<pre>WHILE condition [statements] ENDWHILE</pre>

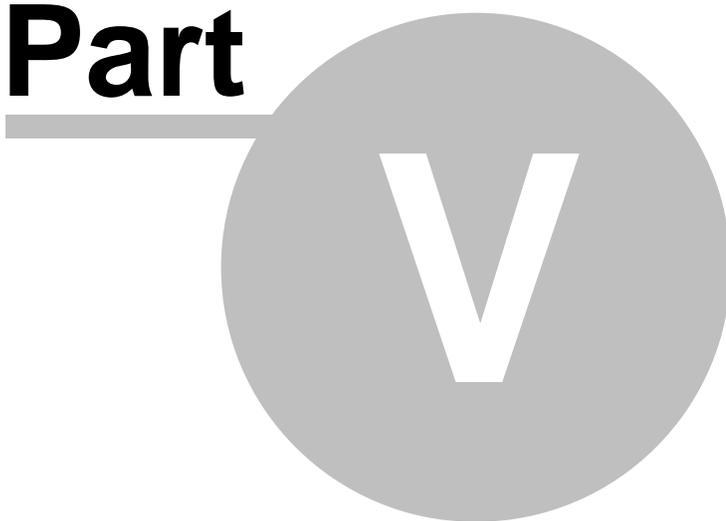
Parameter:	Description:
condition	Expression that evaluates to TRUE or FALSE.
[statements]	One or more statements executed while condition is True.

Example:
<pre>VARIABLES: a TYPE: Integer WHILE (a > 0) print "a = ", a a = (a - 1) ENDWHILE</pre>
Returns:
Example shows how to repeat a process by looping back over the values until a condition is no longer True .

Links:
DO, FOR
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:
Alternate Objects	Alternate Objects are created for accessing data outside of the range of a system's normal context or object range.
Block	represents the current Trading Block and is generally only used for debugging purposes
Broker	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.
Chart	used to create custom charts
Email Manager	used to send emails from scripting
FileManager	used to read and write files
Instrument	Represents a given market, or a tradeable instrument to access pricing, position, and other information that is useful for influencing system orders and positions.
Order	contains information about the order used in the Can Fill Order script
Script	used to access custom user scripts
System	represents the system itself and is used to access system-level information such as the total equity
Test	represents the test and is used to obtain test-level information like the start and end dates

Bar Indexing Reference:

All data instruments with a price record have a for each record and have the values to create a displayed price-bar. Each daily, weekly and monthly price bars have a records that shows the Open, High, Low, Close prices for the time period. This price bar is what is displayed on a

chart as a vertical bar on a chart display.

This approach is also true for Intraday data where the same date and time of day value is used for building a series of price records. The same (Open, High, Low, Close) values show the different transaction times for an Intraday price bar. It is also used for instrument records where less than four price records show the open, high, low and close or last price of a price record.

When looking back through a series of price records, or through a series of date records, the most current record is referenced with a series offset value of zero. This is how the most current price record is indexed (i.e. `instrument.date[0]`).

During a system test, the previous data file's index record can be accessed by adding a value of one to what had been a value of zero when it was the current date record being accessed.

All instrument data records are automatically indexed during a test. In a test of a data file that has one-hundred records, the current date will always be zero. To access the previous record, the index value must be one. To access the record of four days ago, that record is accessed using an index value of three: `instrument.date[3]`.

At the end of the data, that just a test to see an indicator, the index value increase is value. For example, if today's price uses an index of `instrument.date[0]`, the previous price record is identified by adding a "1" to its previous index, `instrument.date[1]`. To go back two-days, the reference would be: `instrument.date[2]`.

is the current instrument record for any symbol. This offset method is also true to reference the property value of a property like Total Equity. For example `test.totalEquity[0]` is the current equity value made available at the end of the current test day. Test information that can be access previous values created during test simulation are contained in an array. An array is a series of values and property and instrument information that allows access to previous information is known as a series because of values that were created earlier in a test.

Users can create a series for storing earlier information in either a **BPV** or an **IPV** data type. Series in both data types can be auto-index or they can be manual. Auto-index series automatically get size with the required number of array elements. Manual series must be sized and indexed by the programmer creating a manual series. Manual series have the require functions to handle sizing, referencing, and sorting in two different ways and directions. Auto-indexed **BPV** series are aligned to `test.currentDay` values. Auto-indexed **IPV** values are aligned to each instrument's price record.

All Trading Blox default properties that are kept in a series are automatically created and kept current by Trading Blox. User created series must be first created and then their values

must be maintained serviced by code the programmer creates to keep each element in the series useful.

To reference other price bars or equity values that are preserved in a series use this reference to help you understand what is being referenced:

Current	1-bar ago	2-bars ago	3-bars ago	4-bars ago, etc...
<code>test.totalEquity[0]</code>	<code>test.totalEquity[1]</code>	<code>test.totalEquity[2]</code>	<code>test.totalEquity[3]</code>	<code>test.totalEquity[4]</code>

Bar Indexing Examples:

Previously Trading Blox provided constants `today` and `yesterday` to use. These are no longer supported.

Properties listed with a '[']' must now be indexed using an offset number:

```
' Test Total Equity from 4-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 ]
```

Syntax Colors:

Trading Blox Basic Editor will color various script element types with the colors set in the Syntax Colors section of the Blox Basic Preference section.

Professional coding editors by default use syntax coloring. Coloring various code elements in various ways helps the person writing the code to understand when the spelling is correct, and in the case of Trading Blox Basic Editor, they can also understand if the object element is in context in the script section where the object's element is being referenced. Coloring variation of different groups helps to make reading easier and faster when type types use different colors.

Trading Blox Basic Editor will also vary the color of object elements that have designated scripts where the that object is not normally in context. This means that the color of a object type will be the color designated in the Preference's Syntax Colors, but it will be a different color when it is in a script section where it is not normally in context in the script section.

Section 1 – Alternate Objects

Alternate Objects provides an ability to inherit the abilities of other objects. This inheritance provides access a system's Properties and Function that won't be restricted data outside of system's normal context or range.

The System Object that works well within a system, is not allowed to reach out past its system scope container. When a user Suite has more than one system to execute during a test, there needs to be a method for accessing a system from one system to the next, or from a Global Suite System. To provide access to other systems in the suite, Trading Blox Builder provides the Alternate Object.

All Trading Blox Builder object have properties and functions. The three Objects that an Alternate Object can use to access those other object properties and functions are:

- [Broker Object](#)
- [Order Object](#)
- [System Object](#)

To access information that would be contained in any of the those three Objects, is to create a process method that inherits the capabilities of those objects. That capability is how the Alternate Object works when any of those three object name appear as part of the Alternate Object name.

- [AlternateBroker Object](#)
- [AlternateOrder Object](#)
- [AlternateSystem Object](#)

Once the Alternate Object inherits the abilities of an object, it will be able to use that object's properties and functions without system limitations. This means it can reach into any other system in a suite and get information that is available for the original object.

Before access to another system can be enabled, permission to access another system must be provided:

```
' Set system data access to System-index 1
test.SetAlternateSystem( 1 ) ( <=Click, for more examples & details.)
```

The above [Test-Object](#) Function, [SetAlternateSystem\(systemIndex\)](#) will provide system context to access information to any of the other systems in the suite while being outside of the system being accessed .

Object:	Description:
AlternateBroker	AlternateBroker object is used to place orders in another system. Set the system of using the <code>test.SetAlternateSystem(systemIndex)</code> . Then pass the symbol into the alternateBroker function.
AlternateOrder	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

Object:	Description:
	SetAlternateOrder function where each order is brought into context using the order's index value.
AlternateSystem	Alternate System Object is used when referencing other systems in the same suite.

Links:[LineNumber](#)**See Also:**

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 144

1.1 AlternateBroker Object

[AlternateBroker](#) is also a Broker object with all the same functions and properties as the standard broker functions and properties. Use this method when the script does not by default have context for instrument access shown in [Blox Script Access](#).

To use the [AlternateBroker](#) process the system where the [AlternateBroker](#) function are to be applied must first be brought into context. Context assignment is performed with the [test.SetAlternateSystem](#) function using the Suite's assigned `system.index` value (see line #4 in Example area).

When the [alternateSystem](#) object is set to the system number referenced, the [AlternateBroker](#) object is automatically sets to the same `system.index` value. This dual context-assignment allows orders to be placed for any system from any system, including from a **Global Suite System (GSS)**.

In a **GSS** system instruments can be looped over and orders placed for any instrument in any or the systems in the suite. From a **GSS**, the [instrument.symbol](#) needs to be the first parameter used in the [LoadSymbol](#) function referenced (see line# 2 in Example area). This additional system index reference requirement after the symbol is necessary because 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.

"Mkt" in the example below is a **BPV Instrument** variable necessary to bring an instrument into context.

Example:

```
' Example loads Gold Future contract into the BPV instrument 'Mkt'
If Mkt.LoadSymbol( "F:GC", 1 ) THEN
  ' Set system data access to System-index 1
  test.SetAlternateSystem( 1 )
  ' When the Mkt/Instrument is primed, and
  ' when the Mkt/instrument's position is Flat,...
  If Mkt.isPrimed AND Mkt.position = OUT THEN
    ' Use the alternate Broker Object function
    ' to create an Long Entry On_Open order for Gold
    alternateBroker.EnterLongOnOpen( Mkt.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 ' Mkt.isPrimed AND Mkt.position = OUT
ELSE
  ' Show the Gold Futures file failed to load.
  PRINT "Unable to load symbol"
ENDIF ' Mkt.LoadSymbol
```

Links:[test.SetAlternateSystem](#)**See Also:**

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 orders 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 function are available to reference or change.

Links:

[SetAlternateOrder](#), [SetSortValue](#), [totalOpenOrders](#),

See Also:

1.3 AlternateSystem Object

Alternate System Object is used when referencing another system that is in the same suite.

Example - Alternate System Access:

```
' 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
```

Return - Alternate System Access:

Out will display the system's suite index, its system-name, and its total-equity amount.

See the topic: [AlternateBroker Object](#) for an example of how it is used.

Links:

[systemCount](#), [AlternateBroker Object](#), [test.SetAlternateSystem](#)

See Also:

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:	Description:
group	Group name of the Blox assigned by the Blox Editor
instrumentExists	<p>Property returns a TRUE value when the script automatically provides instrument context by default.</p> <p>A FALSE value is returned for scripts where automatic instrument context is not provided. Access to instruments is still possible using the LoadSymbol() function.</p> <p>See: Accessing Instruments topic for more details.</p>
name	Block name being executed. When name is printed during debugging operation it shows the current script execution locations.
orderExists	<p>Property returns a TRUE value when an instrument's order is in context and available for access.</p> <p>A FALSE value is returned an order does not exist or is not in context.</p>
scriptName	<p>Script section being executed will be name of the executing script where this property is place when executed.</p> <p>This also means that when a script section, like a custom function, is called from another script, and this property is executed in the custom script, the name script section where this property is placed is returned.</p> <p>If the calling function name is the name needed, the calling function will need to pass its script name to the script section being called.</p> <p>Example:</p> <pre>script.Execute("CalledScriptName" , block.scriptName)</pre> <p>This Block property is important to use when printed output information is being used to Debug a process.</p>
system	System where block is physically assigned during testing.
systemIndex	<p>System index number of the suite.</p> <p>When only one system is located in the Suite, the system index value is always 1.</p>

Properties:	Description:
	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 Section Name:	Execution 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 - 1:
<pre> ' ~~~~~ ' 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 ' ~~~~~ </pre>
Returns - 1:

Common debugging block object properties:

Example - 2:
<pre> ' ~~~~~ ' 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 ' ~~~~~ </pre>

Example - 2:**Returns - 2:**

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](#)

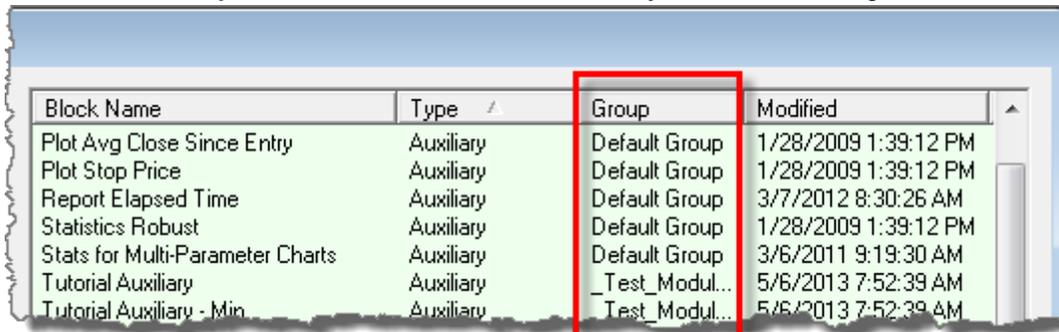
See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 106

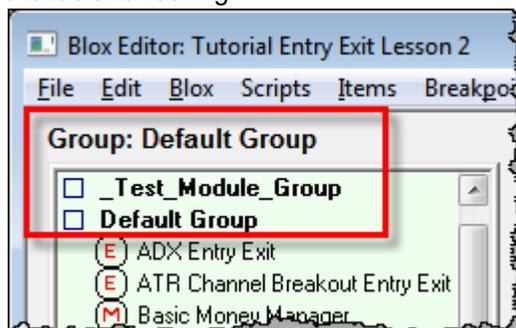
2.1 Group

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:

Description:

<none>

Example:

```
PRINT "Block Group Name", block.group
```

Returns:

Block Group Name: Default Group

Links:

Links:

See Also:

[Block](#)

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 365

2.2 Name

Actual name assigned to the block that is in the system.

Syntax:	
<code>block.name</code>	
Parameter:	Description:
<code><none></code>	
Example:	
<pre> ~~~~~ PRINT "Block Name: ", block.name ~~~~~</pre>	
Returns:	
Block Name: <code>_Max_Position_Limiter 3</code>	
Links:	
See Also:	
Block	

2.3 ScriptName

Name of script section, `block.scriptName`, returns the script name where this property is executed.

This is still true when another script section call a script using the `script.Execute` function. Name returned is always the name where the `block.scriptName` is scripted to return a name.

Syntax:

```
block.scriptName
```

Parameter:	Description:
<none>	

Example:

```
' Property is called using the scripting space within
' the script section:
' ~~~~~
PRINT "Block Script Section Name: ", block.scriptName
' ~~~~~
```

Returns:

```
' If this property was call from within the Before Order Execution
' script section the name returned will be:
Block Script Section Name: Before Order Execution

' If this property was call from a script section called UNIT SIZING
' the script section name returned will be:
Block Script Section Name: Before Order Execution
```

Links:

[Execute](#), [Script](#)

See Also:

[Block](#)

2.4 System

Name of the [System](#) to which the blox module is attached and displayed in the **System Editor** module listing.

Syntax:	
<code>block.system</code>	

Parameter:	Description:
<code><none></code>	

Example:
<pre> ~~~~~ PRINT "Blox applied System Name ", block.system ~~~~~ </pre>

Returns:
Blox applied System Name Bollinger Breakout Plus

Links:
System
See Also:
Block

2.5 SystemIndex

Each Blox object uses the Suite's system Index value where that blox is attached. Each [system.index](#) value assignment in a suite is determined by the order in which the system is added in the Suite's list of systems. This means that when a blox is attached to a specific system, the Blox and all of its script sections will return the [system.index](#) assigned value when `block.systemIndex` property is called. This also means any script called from another system, including a Global System, where the called script is located will return the system number assigned to the called script location.

When only 1 system is in a suite the system index will always be one. When more than one system is 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. Global Systems are always assigned a [system.index](#) value of zero.

Any system with the same name as the Suite is automatically converted to a Global System (GSS) and will return a `block.systemIndex` value of zero. Blox modules attached to a GSS will be reported as having a [system.index](#) and a `block.systemIndex` 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:	Description:
------------	--------------

<none>	
--------	--

Example:

```
' ~~~~~
' When this statement is in a script in the only system in a
' Suite the return value will be 1
PRINT "System Suite Index Number ", block.systemIndex
' ~~~~~
```

Returns:

```
' When this statement is in a script in the only system in a
' Suite the return value will be 1
System Suite Index Number 1
' When this statement is in a script along with a Global System
' the return value will be 1 if there is only 1 system in the
' Suite, or the call is made from the system that was the first
' system to be added to the Suite.
System Suite Index Number 1
' If the statement was in the second system added to a multiple
```

Example:

```
' system suite and the system where the script is located is in  
' the second system added to the suite, the return value will be 2  
System Suite Index Number 2
```

Links:[System](#)**See Also:**[Block](#)

Section 3 – Broker

Broker functions generate orders that can create, remove or adjust a position.

All Broker Object Orders created will be applied to the data on the next available trading date.

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 - 1:

```
' 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.
```

Returns - 1:

Example shows how to create a market on Open and an order to enter on a stop at a specified price.

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 - 2:

```
' Protective Single-Unit Exit Order
broker.ExitAllUnitsOnStop( instrument.unitExitStop )
```

Returns - 2:

When there are multiple units in a position Broker-ExitAllOn... functions will create an exit order for each unit in the position.

When a Broker-ExitUnitOn... function is used only the unit number identified in the function parameter will be removed from the position.

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 - 3:

```
' . . . . .
' 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
' . . . . .
```

Example - 3:**Returns - 3:**

Example calculates an exit price, updates the position with the new price, and generates an order to exit on a stop using the calculated exit price.

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 - 4:

```
' Create an Entry On_Open order using symbol
' when Instruments are out of context. This allows
' Trading Blox to generate an order for ThisSymbol when
' ThatSymbol was the symbol Trading Blox placed in the
' Entry Orders Script section.
broker.EnterLongOnOpen( ThisSymbol )

' When the order needs a protective stop-price,...
broker.EnterLongOnOpen( ThisSymbol, ProtectivePrice )
```

Returns - 4:

Example shows how to specify an instrument when that instrument isn't in context.

[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](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 102

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	Descriptions:
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	Descriptions:
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	Descriptions:
EnterLongOnStop	Buy any time if the market hits specified price
EnterShortOnStop	Short any time if the market hits specified price
Limit Orders	Descriptions:
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	Descriptions:
EnterLongOnClose	Buy on the close

Market On Close Orders	Descriptions:
EnterShortOnClose	Short on the close
Stop Close Only Orders	Descriptions:
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	Descriptions:
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.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 301

EnterLongOnOpen

Enters a long position on the next open. This function is generally used by an [Entry Block](#) to initiate a position.

Syntax:
<code>broker.EnterLongOnOpen([protectStopPrice])</code>

Parameter:	Description:
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.  
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:**Description:**

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.
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.

Important Note: [Limit Order Operation](#)**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:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 283

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
stop
broker.EnterShortOnStopOpen( entryPrice )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 296

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.

Important Note: [Limit Order Operation](#)

Syntax:

```
broker.EnterShortAtLimitOpen( limitPrice [, protectStopPrice] )
```

Parameter:	Description:
limitPrice	Price which the market must be higher in order to trigger this order
protectStop Price	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:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 291

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:	Description:
stopPrice	Stop order price
protectStop Price	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.

Important Note: [Limit Order Operation](#)

Syntax:

```
broker.EnterLongAtLimit( 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.EnterLongAtLimit( priceTarget, protectStopPrice )
OR
broker.EnterLongAtLimit( priceTarget )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 281

Limit Order Operation

[Limit Order Entry with protective stop](#) ([Traders' Roundtable](#))

Tim Arnold » Tue Aug 18, 2020 7:40 am

The protective stop that is placed with an entry order is computed prior to the open, so the entry price and protective stop price are computed with data available before the open of the market. If the market gaps past the stop price, the order will not be filled.

The protective stop price can be adjusted based on the fill price the next day, or just as the order is filled (using the Can Fill script). However if you adjust the stop price as the order is filled, this is only available for back testing, since in real trading you would have to adjust that manually. The Order Generation Report is generated before the open, and would not have information about the entry price or a recomputed protective stop price.

The Entry Risk computed by the Unit Size script is based on the original entry price and entry protective stop. This computes the trade size that is then output on the Order Generation Report.

Links:
EnterLongAtLimitOpen , EnterShortAtLimitOpen , EnterLongAtLimit , EnterShortAtLimit , EnterLongAtLimitClose , EnterShortAtLimitClose
See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 703

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.

Important Note: [Limit Order Operation](#)

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:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 289

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:	Description:
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 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:**Description:**

protectStop
Price

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:	Description:
stopPrice	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:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 287

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:	Description:
stopPrice	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:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 295

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.

Important Note: [Limit Order Operation](#)

Syntax:

```
broker.EnterLongAtLimitClose( 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.EnterLongAtLimitClose( priceTarget )
```

Links:

[Broker](#), [Entry Order Functions](#), [Unit Size Script](#)

See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 282

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.

Important Note: [Limit Order Operation](#)

Syntax:

```
broker.EnterShortAtLimitClose( limitPrice, [ protectStopPrice] )
```

Parameter:	Description:
limitPrice	Entry limit order price
protectStop Price	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:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 290

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 exit orders that reduce the number of units in a position, or reducing the quantity of contracts/shares in a position.

Market On Open Orders	Descriptions:
ExitAllUnitsOnOpen	Exit all units on the open
ExitUnitOnOpen	Exit the specified unit on the open
Stop Open Only Orders	Descriptions:
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	Descriptions:
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	Descriptions:
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	Descriptions:
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	Descriptions:
ExitAllUnitsOnClose	Exit all units on the close
ExitUnitOnClose	Exit the specified unit on the close
Stop Close Only Orders	Descriptions:
ExitAllUnitsOnStopClose	Exit all units on the close if market hits specified price
ExitUnitOnStopClose	Exit the specified unit on the close if market hits specified price
Limit on Close Orders	Descriptions:
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:

Parameter:	Description:
none	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:**Description:**

unitNumber

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:

Parameter:	Description:
stopPrice	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:	Description:
------------	--------------

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 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:**Description:**

unitNumber

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:

Parameter:	Description:
unitNumber	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:**Description:**

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 all units during the next bar if it hits our stop.
broker.ExitAllUnitsOnStop( exitStop )
```

Links:

[Broker, Exit Order Functions](#)

See Also:

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:

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:	Description:
------------	--------------

limitPrice	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

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:**Description:**

Parameter:	Description:
unitNumber	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:

Parameter:	Description:
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 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:**Description:**

unitNumber

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:

Parameter:	Description:
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 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 Broker Order Information

Remove One or More Units from a Position:

When there are multiple units in a position [Broker-ExitAllOn...](#) functions will create an exit order for each unit in the position.

When a [Broker-ExitUnitOn...](#) function is used, and only a specific unit number is to be removed, the unit number given to the function will remove the number unit. When more than a specific, but not all the units are to be removed, each unit number given to this function VIA loop or multiple Broker statements will be removed.

Taking Profit on Bar of Entry:

All orders generated by Trading Blox will be applied to the next available trading date.

To take profit of an active position, use limit orders. Limit orders don't get filled , i.e.

"[.ExitAllUnitsAtLimit](#) ", or a specified number of units i.e. "[.ExitUnitAtLimit](#)" that loops and generates an order for a single unit, or when contained in a loop, it can generate more than one order. A percentage of the quantity in a position can be removed using "[.AdjustPositionAtLimit](#)" where the percentage of the position's quantity is removed.

A question on the Trading Blox Roundtable Forum Asked, "[How to take a profit exit on the bar of entry when Daily Data is being used?](#)" The best answer is given, but it is a test only solution. It won't be possible to use the process explained in live trading because all entry and exit orders are not processed until the next price record becomes available. When Daily data is used, there is only one price record for each date. This means that when a system generates an order to exit, the exit won't be applied to the bar on which the Exit-On-Limit order was created.

When it is necessary to have the system generate an order to exit at a limit to capture a profit, or to reduce size, [use intraday data instead](#). If the calculation need to be on Daily, Weekly or Monthly data, Trading Blox can create those time periods internally using intraday data.

3.4 Entry Day Exit Order

This information requires Trading Blox Builder version: 5.4.2.x or greater.

In the Trading Blox Builder Roundtable forum this question was asked: [Same day exit as entry?](#)

- **Is it possible to exit a trade the same day as entry other than using the protective stop function? Can you just exit at the close on the day of entry?**
 -
-

Yes. You can use the After Instrument Open script to access the trading day [OHLC](#) and create orders to exit on the close based on computations and criteria.

Note:

This is fine for back testing, but these orders will not show on the order generation report for trading because they are computed after the open.

1. The entry needs to be in the Entry Orders script. It needs to **Enter on Open**. It cannot be an "[Entry OnSTOP](#)" / [LIMIT](#) or [Close](#) order.
 2. Then in the After Instrument Open script, you can add a [Broker](#) call to "[EXIT OnClose](#)" [order](#).
 3. If you want to [Enter OnStop/Limit](#) order, and then "[Exit OnClose](#)", you would need to use the [Before Close](#) script and do a [manual instrument loop](#).
 4. Another option, if using [STOP/LIMIT](#) orders, is to place the "[Exit OnClose](#)" order in the [Entry Order Filled](#) script.
-

3.5 Position Adjustment Functions

Position adjustment functions are used to increase or reduce position size.

A new instrument unit number can change when a position's size is changed.

- When size is added to a position, [order.isEntry](#) is **TRUE**.
- When size is added to a position that is **LONG**, [order.isBuy](#) is **TRUE**.
- When size is removed from a position, [order.isEntry](#) is **FALSE**.
- When size is removed from a position that is **LONG**, [order.isBuy](#) is **FALSE**.

When these Position Adjustment 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, the next available unit will be reduced for what was not possible in the previous unit. When all of quantity in a position is reduced to zero, all the units in the position will be 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.

Note:

When this function is used to adjust the size of an order, it is best to place the Position-Adjustment function in either the Exit Orders or the Entry Orders script sections. Placing the size adjusting function in the Update Indicators script executes after the close of the instrument's results for the date and time when the script is accessible. This means the changes you expect to see on the Positions & Orders Report cannot appear on the same day the report is displayed.

Adjust Size On Open Order	Descriptions:
AdjustPositionOnOpen	Adjusts the position on the open
Adjust Size On Stop Order	Descriptions:
AdjustPositionOnStop	Adjusts the position any time if the market hits specified price
Adjust Size At Limit Order	Descriptions:
AdjustPositionAtLimit	Adjusts the position on any time if the market goes beyond specified price
Adjust Size On Close Order	Descriptions:
AdjustPositionOnClose	Adjusts the position on the close
	Descriptions:
AdjustSystemRiskToMax	When this function is called, it will adjust all positions in the system proportionally so that the total system risk is equal to this desired risk percent.

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 used by a [Risk Manager Block](#) to lighten a position to meet certain risk restrictions. It can be used in other script where instruments have context.

Syntax:

```
broker.AdjustPositionOnClose( adjustmentFactor )
```

Parameter:	Description:
adjustmentFactor	Factor by which the current position quantity will be multiplied. Result will determine position size after adjustments have been processed.

Returns:

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:

adjustmentFactor

Description:

Factor by which the current position quantity will be multiplied. Result will determine position size after adjustments have been processed.

Returns:**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 by 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.

Returns:**Example:**

```
' Adjust the position size by our computed adjustment when Stop price is
penetrated.
broker.AdjustPositionOnStop( 0.75, stopPrice )
```

Links:

[Broker](#)

See Also:

[Risk Manager Block](#)

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:

[Risk Manager Block](#), [AdjustPositionOnOpen](#)

AdjustSystemRiskToMax

This function only has one parameter. That parameter is a risk percent entered as a decimal value. When this function is called, it will adjust all positions in the system proportionally so that the total system risk is equal to this desired risk percent.

For example, assume the overall system risk (as computed for each instrument using the close and the current stop price) is 4% for example (.04). Assume the risk should be 5% (.05). To get to the required risk of 5%, this function will adjust/increase every position 25%.

If the total risk for all positions in the system returns an average risk rate of 10%, and the required risk rate of the portfolio is 5%, then every position will be reduced by 50%.

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.AdjustSystemRiskToMax( fixedRiskAdjustRate )
```

Parameter:

`fixedRiskAdjustRate`

Description:

Factor by which the current position quantity will be multiplied. Result will determine position size after adjustments have been processed.

Returns:

System will be adjusted to meet the difference between the current risk and desired risk.

Example:

```
' Reduce the position size by our computed adjustment.
broker.AdjustSystemRiskToMax( fixedRiskAdjustRate )
```

Links:

[Broker](#), [Risk Manager Block](#), [Adjust Instrument Risk Script](#)

See Also:

Edit Time: 9/11/2020 4:48:24 PM

Topic ID#: 706

3.6 Adjusting Size Example

An Adjustment to reduce the position can create multiple exit orders, as the potentially multiple units are exited.

However an Adjustment to increase the position size will always only create one order. You can then modify that order as desired.

In this case I set the stop price for the adjustment order to the same value as the first unit. This assumes the first unit has a stop associated with the original entry. You can also compute any value for this unit stop, or verify that the first unit still has a stop, etc.

Example:

```
' https://www.tradingblox.com/forum/viewtopic.php?f=58&t=11988
'
' Look for scale up orders. If the target number of lots
' is greater than the number of lots you currently have
' on by a factor of "scale percentage", resize the
' position by scaling it up or increasing the number of
' lots to the targetLotSize.
'
' When an Exit Order for the position is not active,...
If instrument.position <> OUT THEN
    ' Use the Parameter "scalePercentage" to determine the current Lot
    Size in the Position
    If targetLotSize / (instrument.currentPositionQuantity /
instrument.roundLot) > 1 + scalePercentage THEN
        ' and then create the order adjustment value for the broker
        function
        broker.AdjustPositionOnClose(targetLotSize /
(instrument.currentPositionQuantity / instrument.roundLot))
    '
    ' When an Adjust Position Order become active,...
    If system.OrderExists() THEN
        ' Update the Current UnitExitStop Price
        order.SetStopPrice( instrument.unitExitStop[1] )
        ' Update the Order Log Action Information
        order.SetRuleLabel( "Vol Scale Up, " )
        ' Update the Order Report Adjust Information
        order.SetOrderReportMessage( "Vol Scale Up" )
    ENDIF ' system.OrderExists()
    ENDIF ' targetLotSize > 1 + scalePercentage
ENDIF ' i.position <> OUT
```

Links:

[Position Adjustment Functions, Broker Order Information](#)

See Also:

[Broker Objects](#)

Edit Time: 3/21/2024 10:41:38 AM

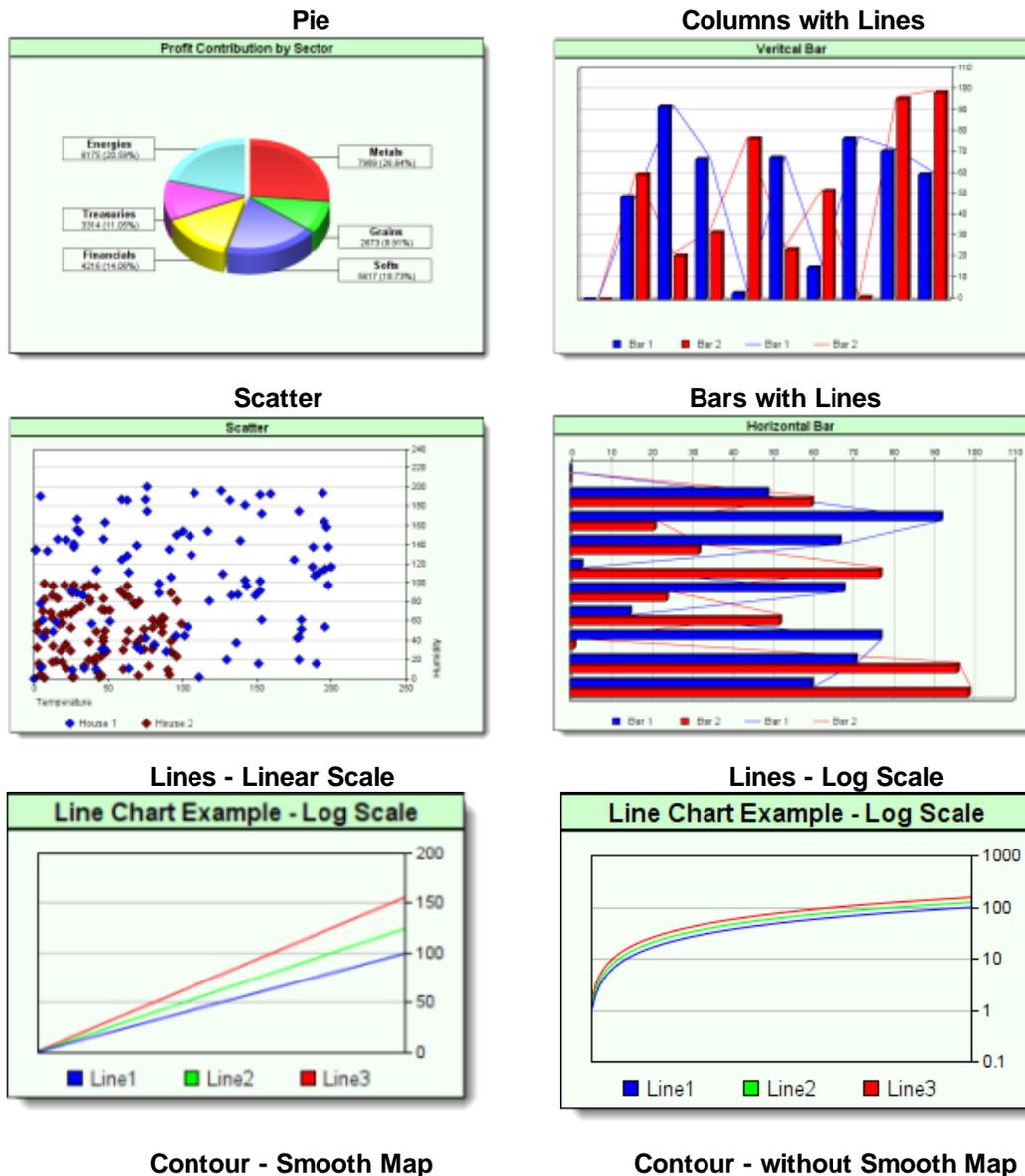
Topic ID#: 671

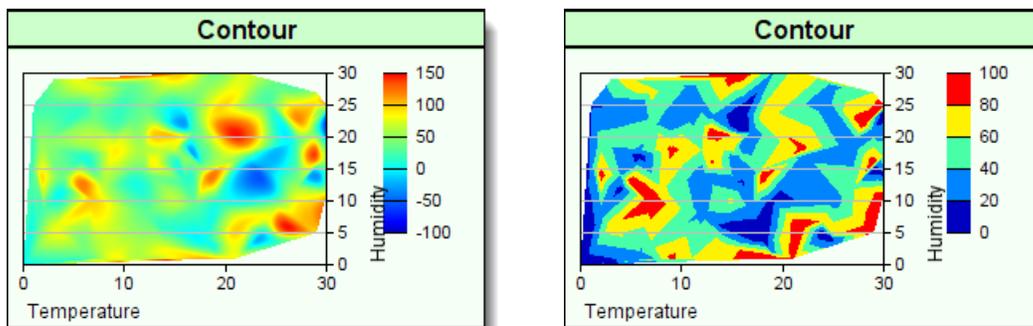
Section 4 – Chart

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 charts are created when Trading Blox Builder Basic Keyword statements are scripted to create one or more custom charts.

Custom chart can be directed appear in the Trading Blox Builder **Summary Performance Reports**, or as individual images in the user's default browser. 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 displaying results in new ways.

These new chart types are intended to expand the ways that Trading Blox Basic can display test results as independent images, or as an expanded level of information at end of simulation test. All the needed new functions and properties needed to create, store and include the custom charts in an expanded report, or in a browser display can be found in the topics of the 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:

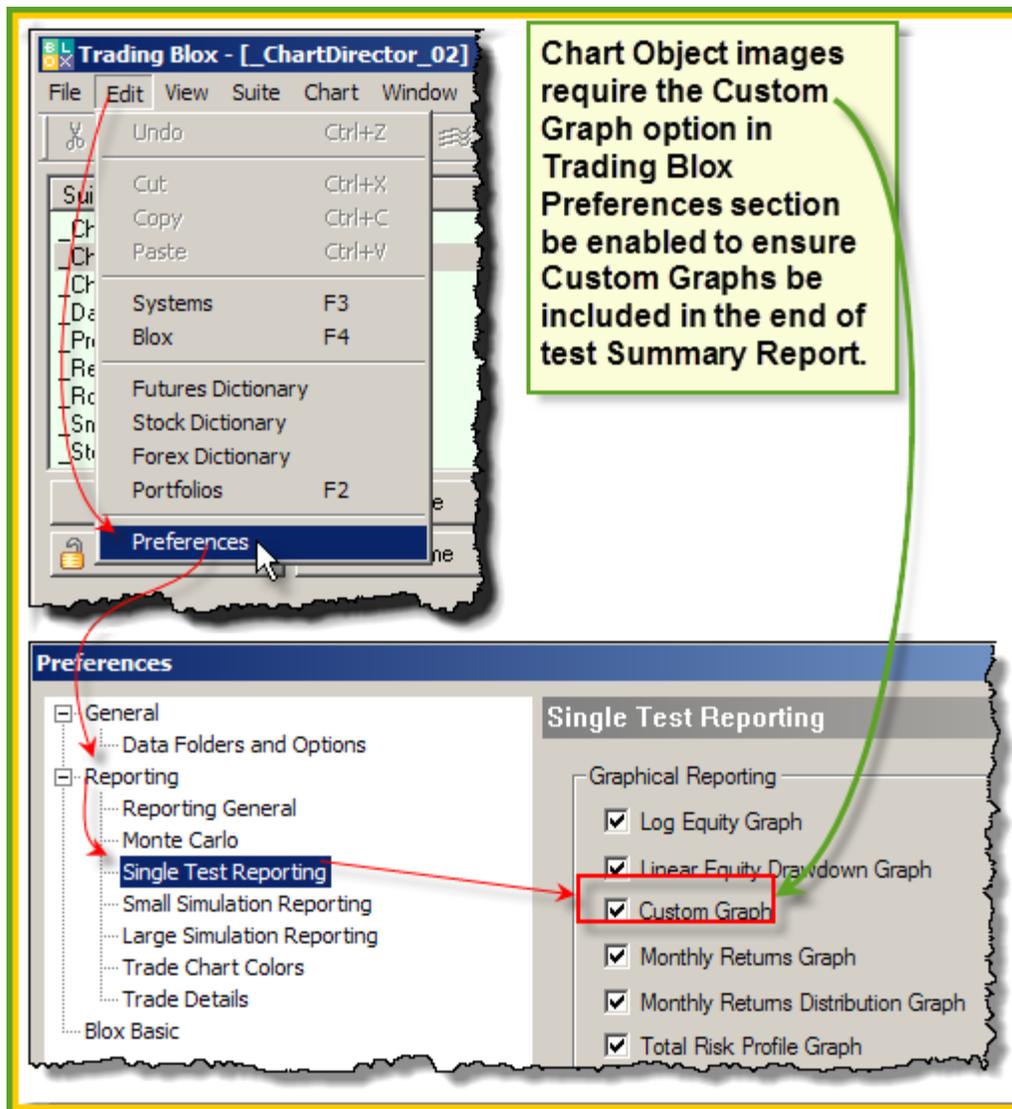
- Collect and analyze the data into BPV numeric series, and BPV String series if labels are needed.
- Determine the size of the chart, and then execute either the [NewXY](#) or [NewPie](#) functions to create chart image space .
- Adjust the plotting area within the chart image so the data plots, scales, 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 Builder 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.SetChartSimulationtHtml( chartHtml1 + "<br>" + chartHtml2 )
' =====

```

Display Custom Charts in default browser:

Usually, the default program is the computer's default browser. Place this code section below the area where the custom chart script creation has saved the chart using the `chart.Make` function

Example - AFTER TEST SCRIPT:

```

' ~~~~~
' 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.html"

' 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 +
sTD_Suffix )

  ' 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:

[CHARTNOVALUE Make](#), [NewPie](#), [NewXY](#), [OpenFile](#), [SetChartSimulationHtml](#), [SetChartTestHtml](#)

See Also:

[Chart Director Help Information](#)

4.1 AddBarLayer

[AddBarLayer](#) can modify the appearance of bars and enhance the 3D effect of an [XYChart](#) created to show bars or columns.

Note - 1:

This method is not required with bar charts, but when it is used the first parameter provides five methods that change the way bars 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.

Note - 2:

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.

Syntax:

```
chart.AddBarLayer( [BarMethodEffect], [3D_Depth])
```

Parameter:	Description:												
[BarMethodEffect]	<table border="1"> <thead> <tr> <th>Bar Method Effect:</th> <th>Use Value:</th> </tr> </thead> <tbody> <tr> <td>Overlay</td> <td>0</td> </tr> <tr> <td>Stack</td> <td>1</td> </tr> <tr> <td>Depth</td> <td>2</td> </tr> <tr> <td>Side (Default)</td> <td>3</td> </tr> <tr> <td>Percentage</td> <td>4</td> </tr> </tbody> </table> <p>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.</p>	Bar Method Effect:	Use Value:	Overlay	0	Stack	1	Depth	2	Side (Default)	3	Percentage	4
Bar Method Effect:	Use Value:												
Overlay	0												
Stack	1												
Depth	2												
Side (Default)	3												
Percentage	4												
[3D_Depth]	<p>Optional: No value is required in this parameter location when the 3D effect doesn't need to be changed.</p> <table border="1"> <thead> <tr> <th>Bar Depth Effect:</th> <th>Use Value:</th> </tr> </thead> <tbody> <tr> <td>Auto Adjust 3D Effect</td> <td>-1</td> </tr> </tbody> </table>	Bar Depth Effect:	Use Value:	Auto Adjust 3D Effect	-1								
Bar Depth Effect:	Use Value:												
Auto Adjust 3D Effect	-1												

Parameter:	Description:	
	Bar Depth Effect:	Use Value:
	Control 3D Depth Size	Pixels depth size

Example - 1:

```

' ~~~~~
' 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( 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 + "\" + hbar.png" )
    
```

Returns - 1:

AddBarLayer Option Image Series Examples
Rotated X-Y Axis Bars
Display Bars in a Left to Right Horizontal Layout

Image Group 1 of 2

**Horizontal Bar Layout Options
Group 1**

AddBarLayer Option Image Series Examples
Rotated X-Y Axis Bars
Display Bars in a Left to Right Horizontal Layout

Image Group 2 of 2

Example - 2:

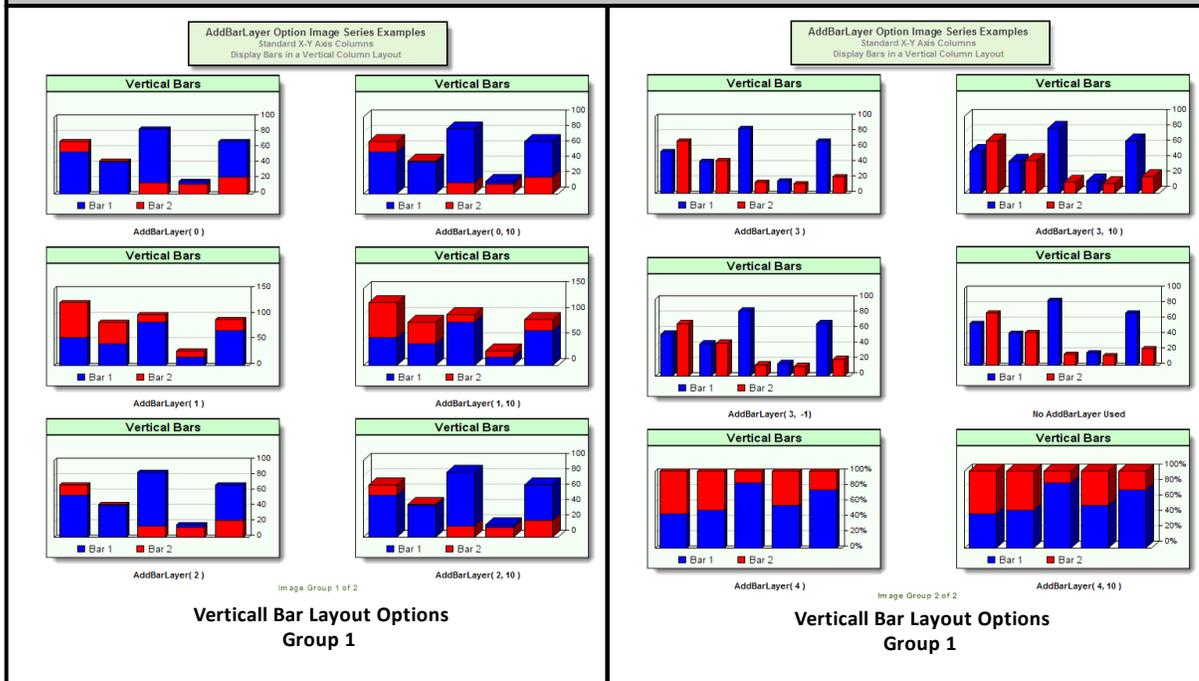
```

' ~~~~~
' 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" )

```

Returns - 2:



Links:

[AddBarSeries](#), [AsSeries](#), [Make](#), [NewXY](#), [resultsReportPath](#), [SetPlotArea](#)

See Also:

[Chart Director Help Information](#)

Edit Time: 9/11/2020 4:48:24 PM

Topic ID#: 115

4.2 AddBarSeries

[AddBarSeries](#) adds a different series of data on a chart display.

Note:

To add an addition bar group to a chart, call [AddBarSeries](#) again with a different data series.

Syntax:

```
chart.AddBarSeries( AsSeries( BarSeries ), Elements )
```

Parameter:	Description:
BarSeries	<p>The BPV numeric series intended to represent a group of bars.</p> <p>Note: Use with all BPV Numeric or String series that are passed to any Chart parameter.</p> <p>AsSeries(. . .) function conditions the series so the chart function will be able to use the information contained within each of the series elements.</p>
Elements	<p>The count of the numeric elements in the series.</p> <p>Note:</p> <p>Manually Sized Series: GetSeriesSize function provides the element count.</p> <p>Auto-Index series: BPV: test.currentDay property reports last series element index. IPV: instrument.bar property reports last series element index.</p>

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 )

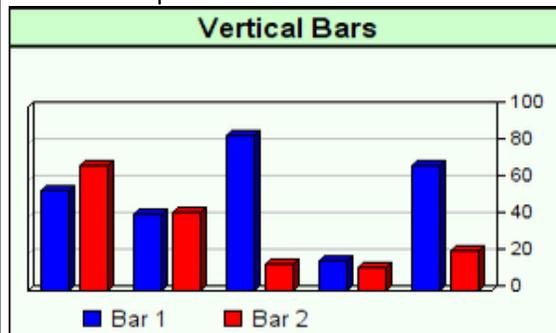
' Add 5 element values to represent "bar2"
chart.AddBarSeries( AsSeries( bar2 ), 5 )

' Create & Save an image of the chart with
' this file name. BackSlash Character is Required
' when using ResultsReportPath
chart.Make( test.resultsReportPath + "\" + "vbar.png" )

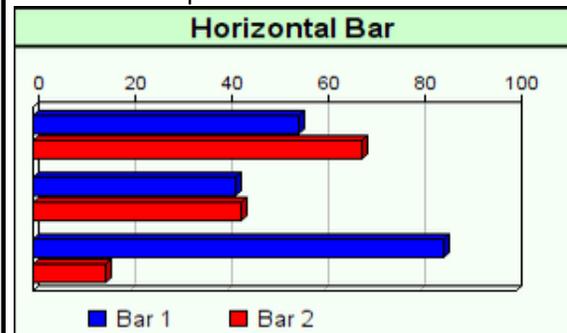
```

Returns:

NewXY Option "Vertical" Not Enabled.



NewXY Option "Vertical" Enabled.

**Links:**

[AddBarLayer](#), [AsSeries](#), [Make](#), [NewXY](#), [ResultsReportPath](#), [SetPlotArea](#)

See Also:

[Chart Director Help Information](#)

4.3 AddContourLayer

Applies a 3-Dimensional contour layer color map onto a newly created [NewXY](#) chart.

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.

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.

Syntax:	
<pre>chart.AddContourLayer(AsSeries(XAxisSeries), _ AsSeries(YAxisSeries), _ AsSeries(ZAxisSeries), _ SeriesCount, _ [Smooth])</pre>	

Parameter	Description:
XAxisSeries	X-Axis Series data.
YAxisSeries	Y-Axis Series data.
ZAxisSeries	Z-Axis Series data.
SeriesCount	The integer value specifies the number of data element values in of all three of the data series specified. Note:

Parameter :	Description:
	<p>Manually Sized Series: GetSeriesSize function provides the element count.</p> <p>Auto-Index series: BPV: test.currentDay property reports last series element index. IPV: instrument.bar property reports last series element index.</p>
Smooth	<p>Optional:</p> <p>When left out, chart will draw contours with clear line definitions between the contour colors levels creating levels that change with clear level lines.</p> <p>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.</p>

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 )

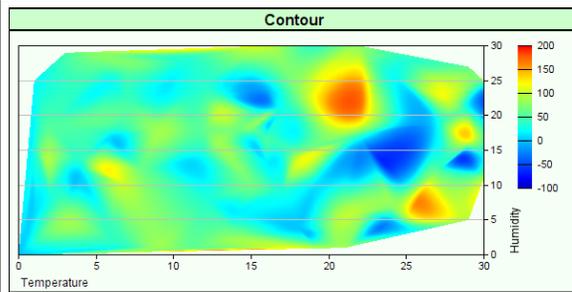
' Size the Bar Plotting area within the boundaries of the graphing
' image area
'           (x-Left, x-Right,  y-Top,   y-Bottom )
chart.SetPlotArea( 10, 100, 40, 40 )

' Create a contour map with BPV numeric series for x, y, & z axis
chart.AddContourLayer( AsSeries( randomx ), _
                      AsSeries( randomy ), _
                      AsSeries( house2x ), iContourLevels, sSurface )

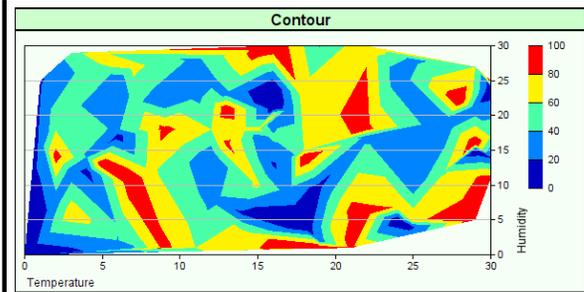
' Add Titles for X & Y Axis Scales
' Function places: "Temperature" near the x-Axis bottom left-side
' and it places "Humidity" at the bottom right as vertical text
chart.SetAxisTitles( "Temperature", "Humidity" )

' 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" )
' ~~~~~

```

Returns:

Contour XY & Z Axis with Smooth Enabled



Contour Smooth option omitted AddContourLayer

Links:

[AsSeries](#), [Make](#), [NewXY](#), [resultsReportPath](#), [SetAxisTitles](#), [SetPlotArea](#)

See Also:

[Chart](#), [Chart Director Help Information](#)

4.4 AddHLOCLayer

Syntax:

Parameter:	Description:

Example:

Returns:

Links:

See Also:

4.5 AddLineLayer

Function allows the chart's Linear scale to be converted to a Log scale.

Syntax:

```
chart.AddLineLayer( , )
```

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

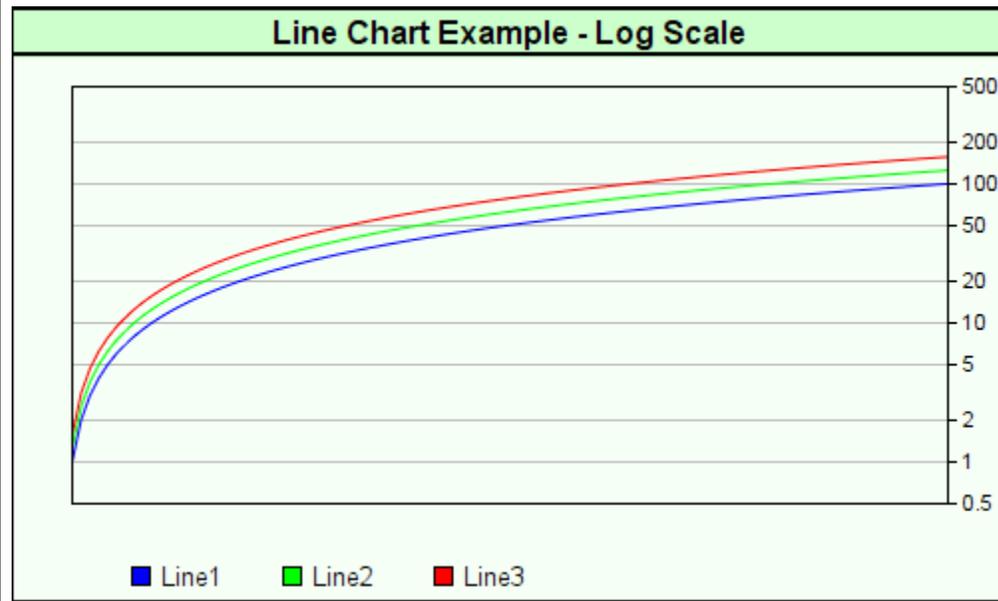
' Create a image
chart.NewXY( "Log Scale Line Chart Example", iChartWidth, iChartHeight )
' Size the Scatter Dot Plotting area
chart.SetPlotArea( 30, 30, 40, 50 )

' 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" )
```

Returns:

Scatter Line Chart Example

Links:

[AsSeries](#), [AddLineSeries](#), [ColorRGB](#), [Make](#), [NewXY](#), [resultsReportPath](#),
[SetPlotArea](#)

See Also:

[Chart](#), [Chart Director Help Information](#)

4.6 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:

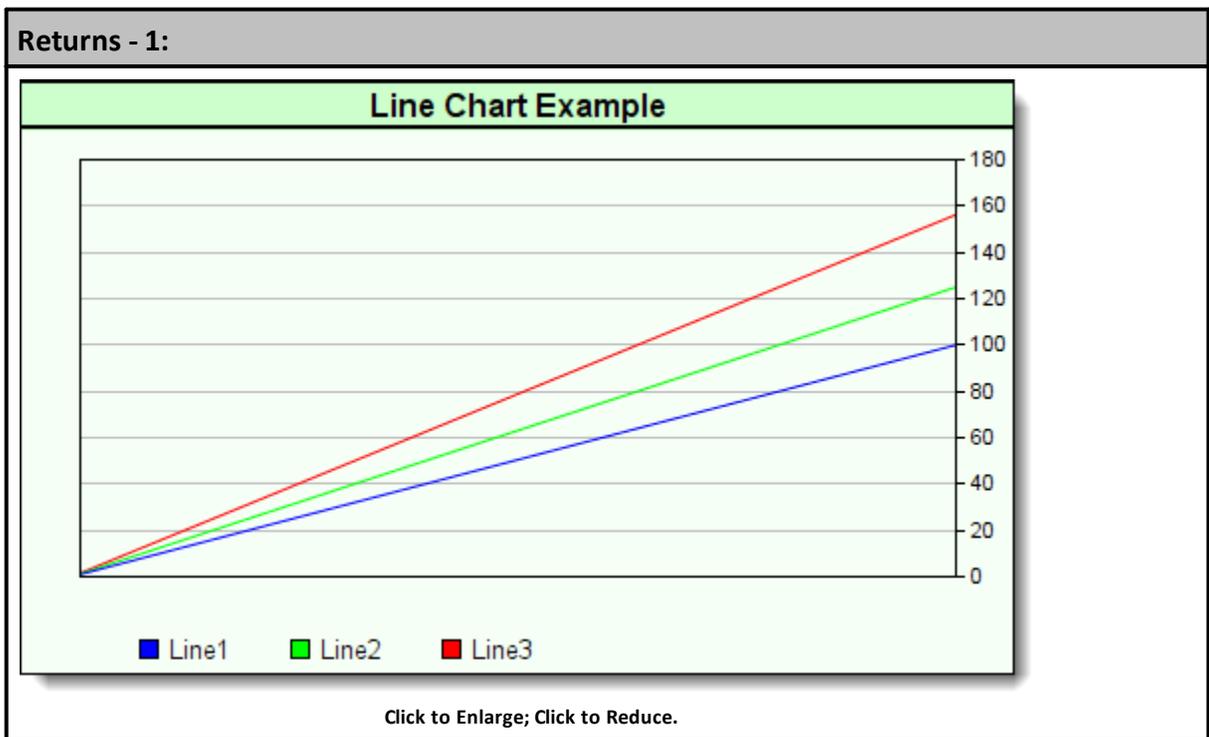
```
chart.AddLineSeries( AsSeries( LineSeries ), SeriesElements, [Title],
[Color] )
```

Parameter:	Description:
LineSeries	Name of BPV Series containing data to be plotted. 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.
SeriesElements	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.
[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 - 1:

```
' ~~~~~  
' 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( 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 + "\" + "LineChartExample.png" )  
' ~~~~~
```

Returns - 1:**3-Line Chart Example:**



Example - 2:

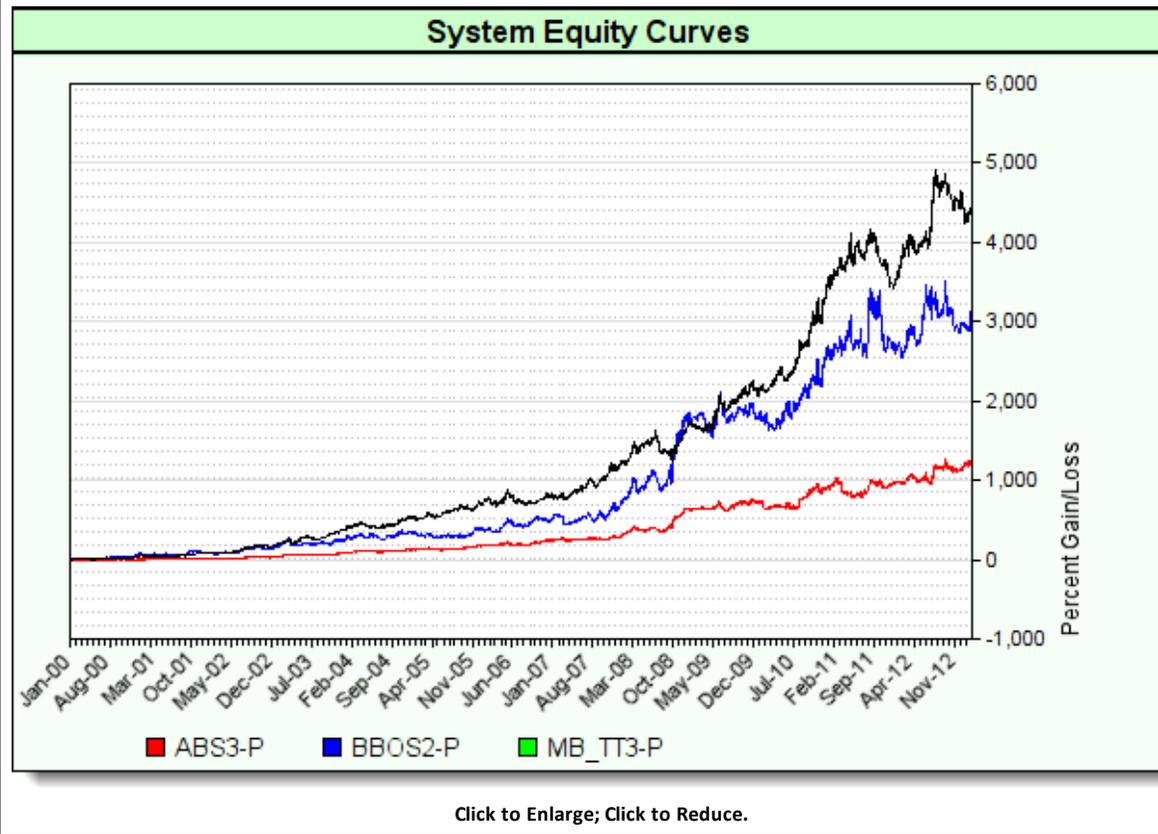
```

' ~~~~~
' 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
series.
    systemEquity[i] = ( alternateSystem.totalEquity[ elementCount - i ]
-
    / alternateSystem.totalEquity[ elementCount - 1 ] - 1 ) *
100
  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 )
' ~~~~~

```

Returns - 2:**Multi-Line Chart Example:**

Returns - 2:



Links:

[AsSeries](#), [ColorRGB](#), [Make](#), [NewXY](#), [resultsReportPath](#), [SetPlotArea](#)

See Also:

[Chart](#), [Chart Director Help Information](#)

4.7 AddMultiChart

Syntax:

Parameter:	Description:

Example:

Returns:

Links:

See Also:

4.8 AddScatter

Scatter charts show unconnected shapes that reflect the two-dimensional values of each data point intersection on supplied data series.

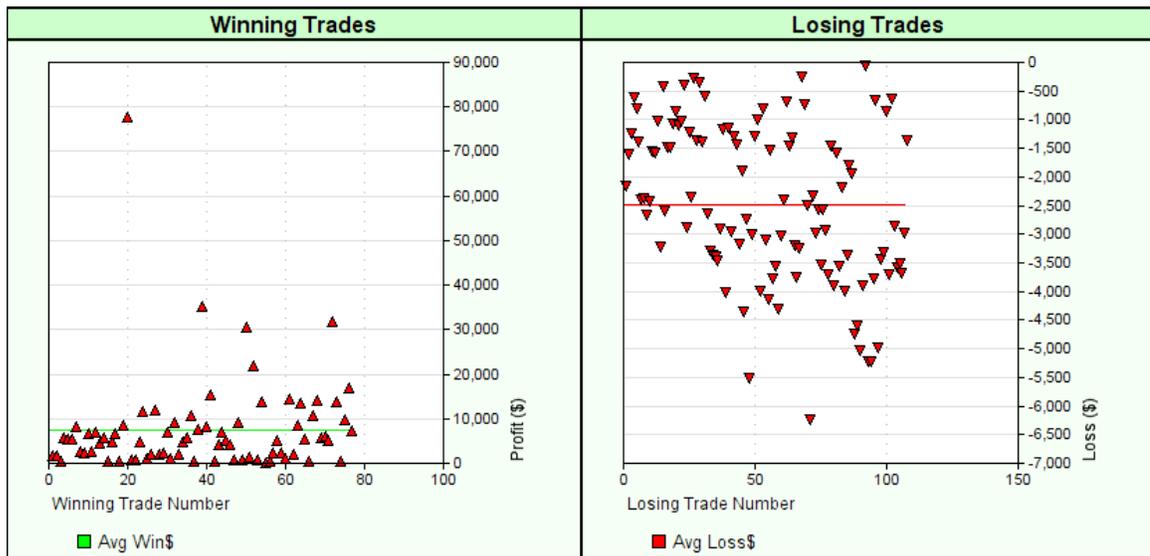
Syntax:																	
<pre>chart.AddScatter(AsSeries(xSeries), AsSeries(ySeries), ElementCount, [Symbol], [Size])</pre>																	
Parameter:	Description:																
xSeries	An array of numbers representing the x values of the data points. If no explicit x coordinates are used in the chart (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, 3,
8 )
' Display the average of the sum of all winning trades as a line
chart.AddLineSeries( AsSeries( dataAverage ), count, "Avg Win$",
ColorRGB(0,255,0) )
' Display the Chart's labels for X & Y Axis scales
chart.SetAxisTitles( "Winning Trade Number", "Profit ($)" )
' Create the chart as an image file in the current test folder.
chart.Make( test.resultsReportPath _
+ "\Winning Trades" _
+ AsString( test.currentParameterTest ) _
+ ".gif" )
' ~~~~~

```

Returns:

Trade Charts Blox - Win Loss Report Image Example

Links:

[AddLineSeries](#), [AsSeries](#), [Make](#), [NewXY](#), [resultsReportPath](#), [SetxAxisLabels](#), [SetAxisTitles](#), [SetPlotArea](#)

See Also:

[Chart](#), [Chart Director Help Information](#)

4.9 Custom Chart Definitions

Definitions and explanations in this section are information taken from the ChartDirector's Help file.

Listed Items:

AutoScale

Zero Infinity

4.10 Make

Saves scripted charts as image files in the names of the location and file name specified.

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.

Syntax:
<code>chart.Make(FileSavingDetails)</code>

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:</p> <pre><Drive>:\<Full path and folder name>\<filename>.<image-format></pre> <p>Supported Image Formats: PNG, JPG, GIF, BMP</p>

Example:
<pre>' Create & Save this new chart as a file. ' Always add a backSlash Character after "resultsReportPath" chart.Make(test.resultsReportPath + "\" + "Scatter.png")</pre> <p>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.</p> <p>Click on any of the these links to review chart creation code examples to see how all the chart methods use this Make function:</p> <ul style="list-style-type: none"> • AddBarSeries

Example:

- [AddContourLayer](#)
- [AddLineLayer](#)
- [AddLineSeries](#)
- [AddScatter](#)
- [NewPie](#)

Returns:

Each links in the above Example will display a chart type.

Links:

[General Properties](#), [resultsReportPath](#)

See Also:

[Chart](#), [Chart Director Help Information](#)

4.11 NewAngularMeter

Syntax:

Parameter:	Description:

Example:

Returns:

Links:

See Also:

4.12 NewMultiChart

Syntax:

Parameter:	Description:

Example:

Returns:

Links:

See Also:

4.13 NewPie

Creates a sectioned Pie chart that can show legends, and pie section values connected to each segment of the pie chart.

Notes:

Do not use the [SetPlotArea](#) function with PIE charts because they don't support adjustable overlays.

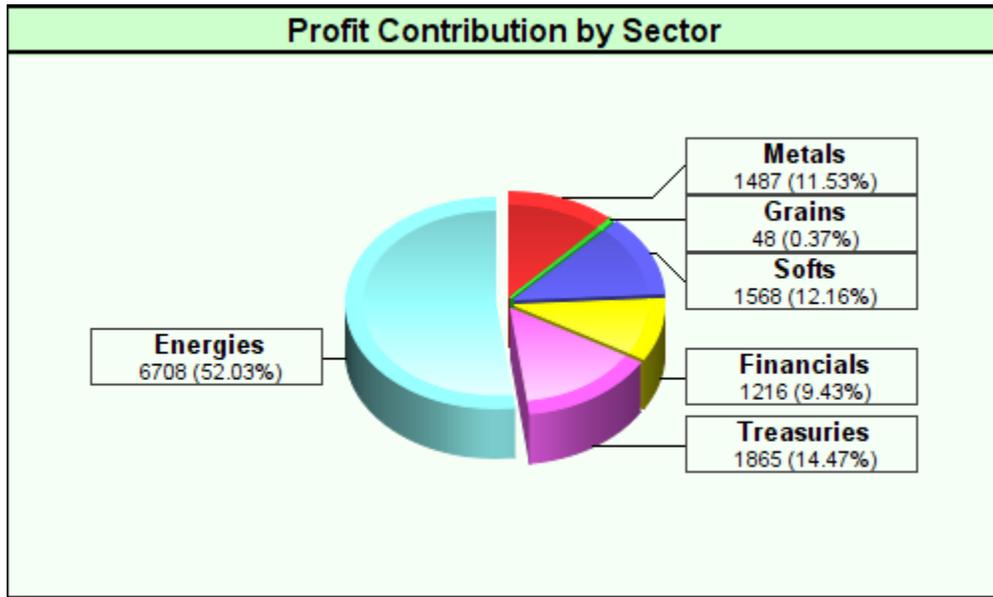
Syntax:

```
Chart.NewPie( ChartTitle, xAxisWidth, yAxisHeight, _
              AsSeries(pieChartValues), AsSeries(pieLabels),
              PieSections, [Option] )
```

Parameter:	Description:
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.

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" )
```

Returns:

Click to Enlarge; Click to Reduce.

Shadow styling around image was added later by graphics editing program

Links:

[AsSeries](#), [Make](#), [NewPie](#), [Random](#), [resultsReportPath](#)

See Also:

[Chart](#), [Chart Director Help Information](#)

4.14 NewXY

NewXY is the starting method used to create Bar, Scatter, Line, linear and, or Logarithmic charts.

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.

Syntax:
<code>Chart.NewXY(ChartTextName, ChartPixelWidth, ChartPixelHeight, [ChartOptions])</code>

Parameter:	Description:								
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]	<p>There are three optional words that can change a chart:</p> <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> <p>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.</p> <p>Color used for the shadow will be different from the normal white color of the background to make its appearance noticeable.</p> </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</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	<p>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.</p> <p>Color used for the shadow will be different from the normal white color of the background to make its appearance noticeable.</p>	Vertical	Bar, Contour, Dot, and Line charts are oriented according the placement of the X & Y axis locations. In most charts the
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	<p>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.</p> <p>Color used for the shadow will be different from the normal white color of the background to make its appearance noticeable.</p>								
Vertical	Bar, Contour, Dot, and Line charts are oriented according the placement of the X & Y axis locations. In most charts the								

Parameter:	Description:	
	Methods:	Descriptions:
		<p>orientation of the X-Axis is placed at the bottom of the chart, and the Y-Axis is placed along the side, or vertical axis.</p> <p>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.</p> <p>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.</p> <p>Contour, dot and line information will also be rotated when those type of displays are added to the chart image space.</p>
	Combined Methods	<p>All three methods can be combined as an additional instruction to the NewXY chart method.</p> <p>Example:</p> <p>"Vertical Shadow" will create a rotated axis chart with a drop shadow effect</p> <p>"Log Shadow" will create a rotated axis with a drop shadow effect.</p> <p>"Log Vertical Shadow" will create a rotated axis Log chart with a drop shadow effect.</p>

Example:

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](#)

Returns:

Link the above example area will display chart information and examples.

Links:[General Properties](#)**See Also :**[Chart, Chart Director Help Information](#)

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 434

4.15 SetAutoScale

Syntax:**Parameter:****Description:****Example:****Returns:****Links:****See Also:**

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 721

4.16 SetAxisTitle

`SetAxisTitles` function adds two text axis titles to X & Y scale locations.

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.

Syntax:

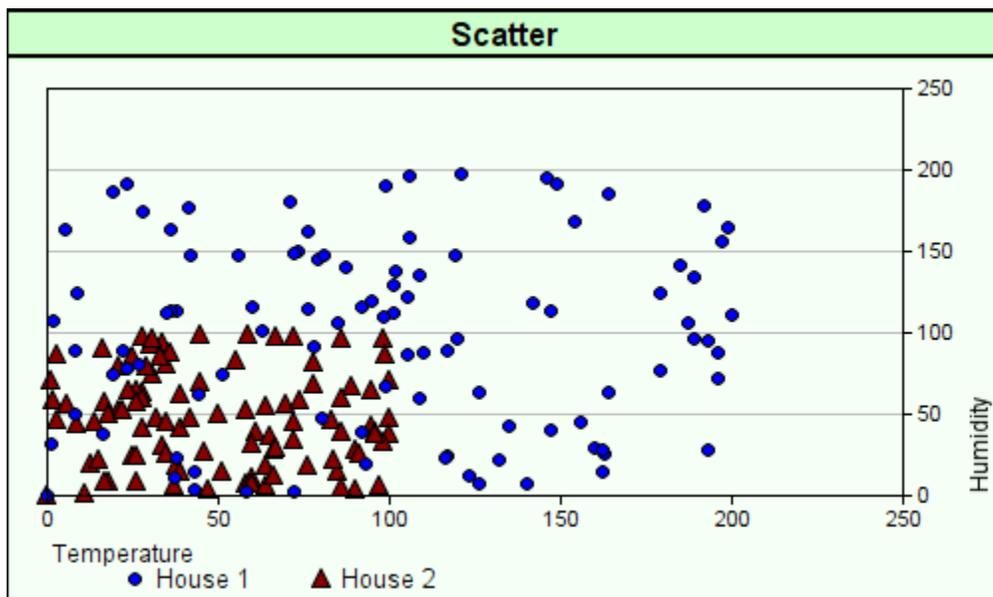
```
chart.SetAxisTitles( xAxisLabel, yAxisLabel )
```

Parameter:	Description:
xAxisLabel	The string - Name of x-Axis title.
yAxisLabel	The string - Name of y-Axis title.

Example - 1:

```
' ~~~~~
' 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,
6 )
' 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,
10 )
' Create & Save this new chart as a file.
' Always add a backSlash Character after "resultsReportPath"
chart.Make( test.resultsReportPath + "\" + "Scatter.png" )
```

Returns - 1:



Click to Enlarge; Click to Reduce.

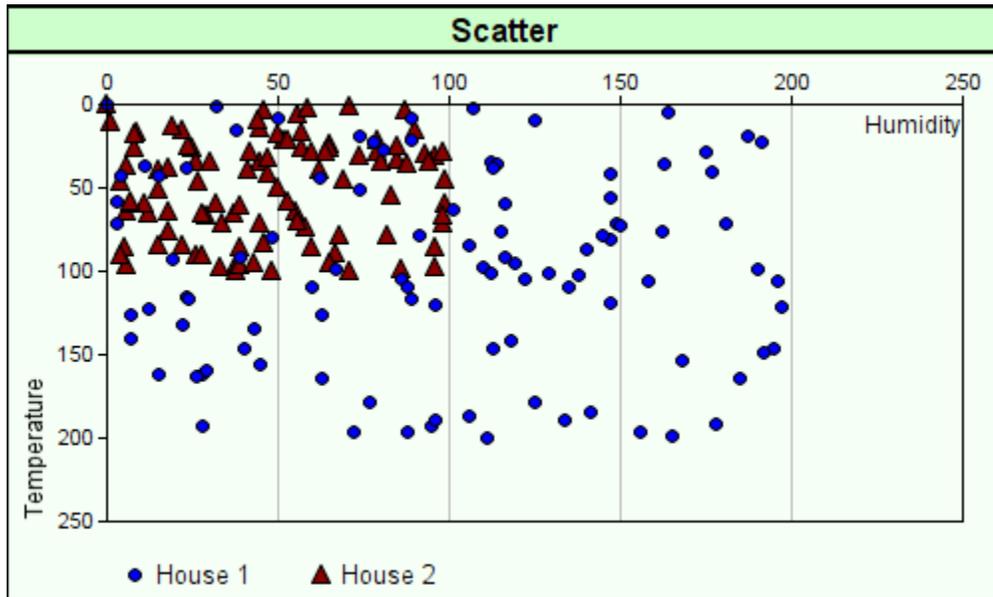
Example - 2:

```

' ~~~~~
' 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,
6 )
' 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,
10 )
' Create & Save this new chart as a file.
' Always add a backSlash Character after "resultsReportPath"
chart.Make( test.resultsReportPath + "\" + "Scatter.png" )

```

Returns - 2:



Click to Enlarge; Click to Reduce.

Links:

[AddScatter](#), [AsSeries](#), [Make](#), [NewXY](#), [resultsReportPath](#), [SetPlotArea](#)

See Also:

[Chart](#), [Chart Director Help Information](#)

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 533

4.17 SetBarGapShape

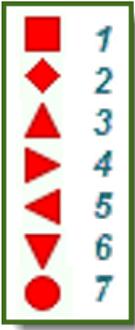
Function must be used with the [AddBarLayer](#) function and it must only be executed after the [AddBarLayer](#) has executed.

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.

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.

Syntax:
<code>chart.SetBarGapShape([Shape], [Gap], [Subgap])</code>

Parameter:	Description:																
<p>[Shape]</p> 	<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>																

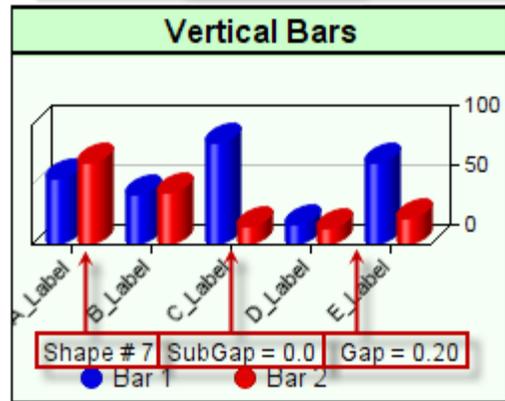
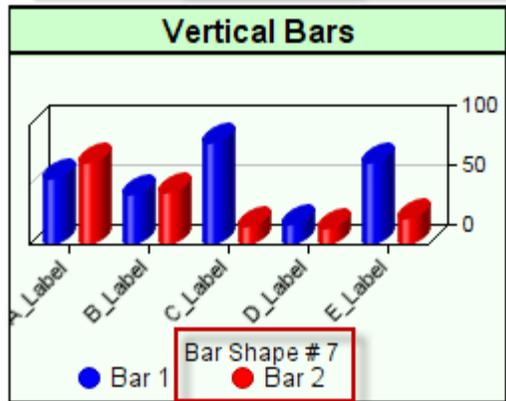
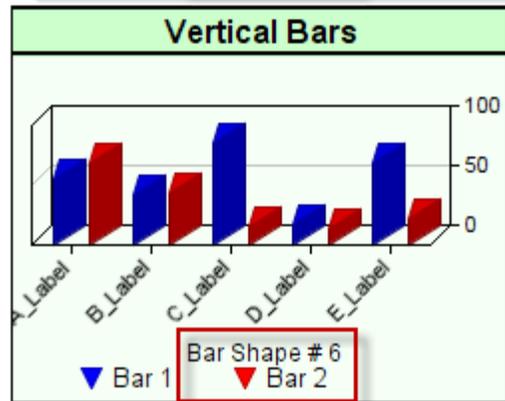
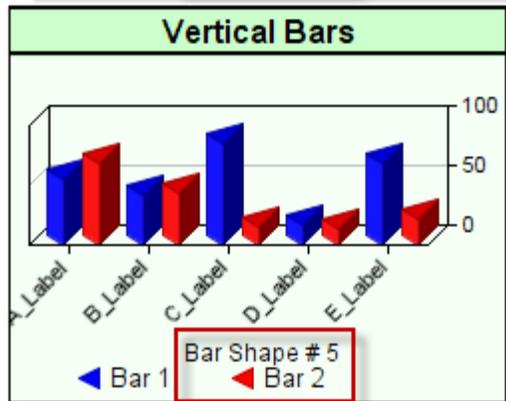
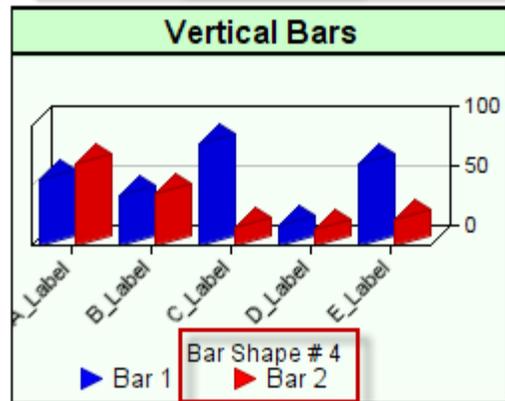
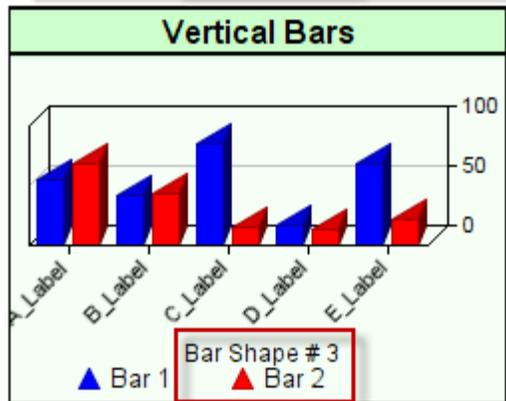
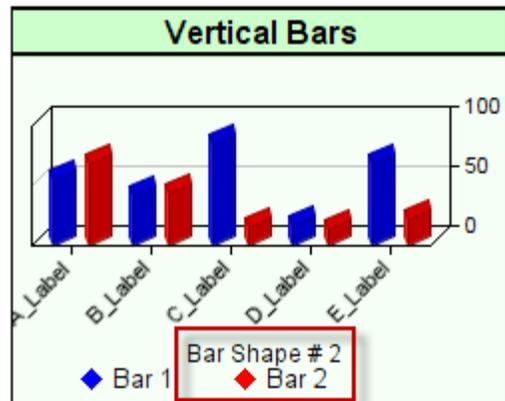
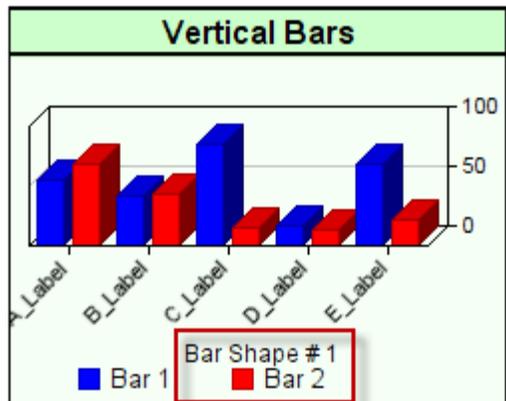
Example:

```
' ~~~~~  
' 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" )
```

Returns:

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:

Returns:



Bar Shapes & Gap Space References

Links:

[AddBarLayer](#), [AddBarSeries](#), [AsSeries](#), [Make](#), [NewXY](#), [ResultsReportPath](#), [SetPlotArea](#)

See Also:

[Chart](#), [Chart Director Help Information](#)

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 534

4.18 SetChartColors

Syntax:

Parameter:	Description:

Example:

--

Returns:

--

Links:

--

See Also:

--

4.19 SetLegendPosition

Syntax:
<code>chart.SetLegendPosition</code>

Parameter:	Description:

Example:
Returns:

Links:
See Also:

4.20 SetPieChartLabelColor

Syntax:

Parameter:	Description:

Example:

Returns:

Links:

See Also:

4.21 SetPlotArea

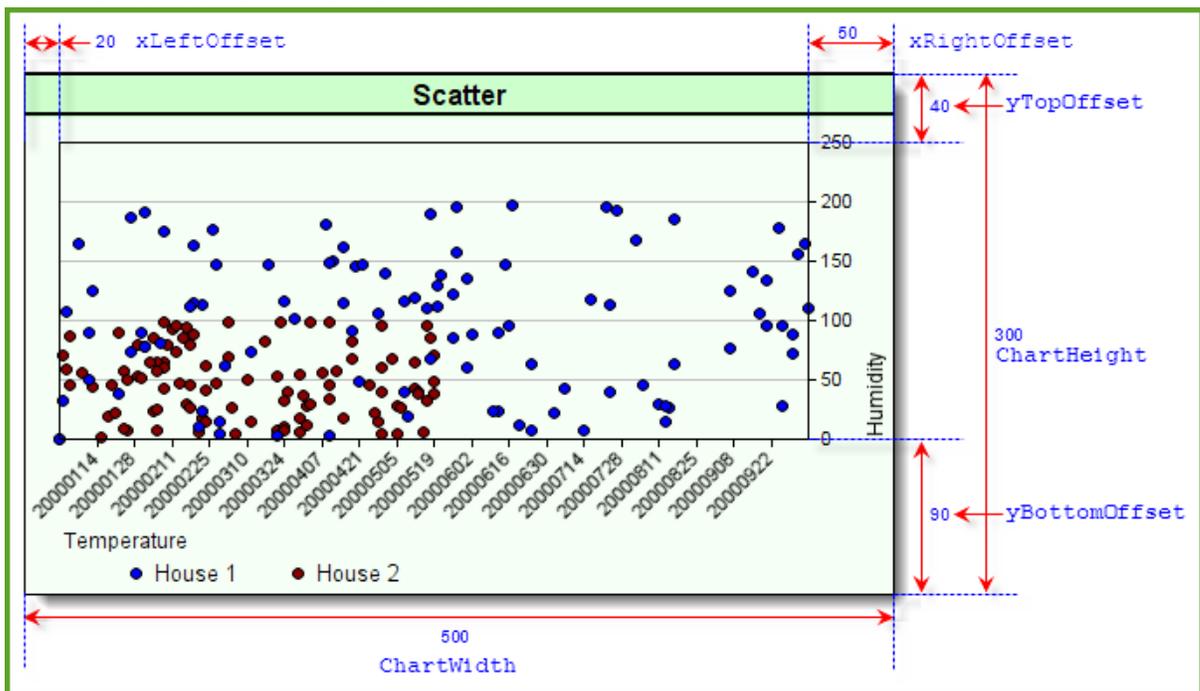
`SetPlotArea` is used to control the positioning and size of the plotting area within the boundaries of the image. Each of the four parameters provides the offset distance in pixels from the image edge to graph's plotting area around which space for the axis scales, labels, and legends is provided.

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 display 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 using a 32-bit color setting. Screen density is 96 pixels per inch set in Landscape mode to fit a 21-inch monitor.

NewXY & SetPlotArea Dimension Locations



Charting SetPlot Area Details

Syntax:

```
chart.SetPlotArea( xLeftOffset , xRightOffset , yTopOffset ,
yBottomOffset )
```

Parameter:	Description:
xLeftOffset	The size of space to offset the graph's left side boundary plotting area from the image's left side edge.
xRightOffset	The 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	The pixel size positioning of the graph's top edge graphing boundary border area from the top edge of the graphing image boundary.
yBottomOffset	The pixel size positioning of the graph's bottom edge graphing boundary border area from the bottom edge of the graphing image boundary.

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 Ploting area
chart.SetPlotArea( 20, 50, 40, 90 ) <-- Reference chart Diagram Details
Above

' 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,
6 )
' 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,
6 )
' Create & Save this new chart as a file.
' Always add a backSlash Character after "resultsReportPath"
chart.Make( test.resultsReportPath + "\" + "Scatter.png" )

```

Returns:

See image at top of this topic.

Links:

[AddScatter](#), [AsSeries](#), [Make](#), [NewXY](#), [SetxAxisLabels](#), [SetAxisTitles](#)

Links:**See Also:**[Chart, Chart Director Help Information](#)

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 549

4.22 Setting AutoScale

This Autoscale ability is used with some of the Chart Object scaling methods. When the methods where Autoscale is used, that feature will set the margin area at both ends of the selected axis identified in the function being used. The value of the axis doesn't matter. This feature will automatically set the scale of the axis based upon the series value assigned to the custom chart elements being used for display on the chart.

Values assigned when an Autoscale method is use can create an upper and lower boundary range where the plotted items will be placed. By establishing the area boundary above and below the plotting area, the minimum and maximum plotted items will not be placed up against the boundaries of the chart. In simple terms, the custom chart areas of the axis boundaries will show some space between the plotted items and the edge of the chart. Consider this example: Suppose a bar chart where the longest bar is 10 units. When the chart's Y-Axis is scale the result show a 0 - 10 as the range of value. In this example, the longest bar will touch the top edge of the plot area. In most cases, the chart will look better with some margin at the top so that the longest bar does not touch the top edge of the chart.

The values use with an AutoScale method determines margin space reserved by the `topExtension` and `bottomExtension` arguments. These arguments determine the portion of the axis where no data point can reach. i.e. a `topExtension` of 0.2 will ensure no data point can fall within the top 20% of the axis.

For a purely positive axis, the bottom end has a "Zero Affinity". That means, ChartDirector will tend to choose 0 as the bottom end because zero is a natural starting point for the axis. However, if the data range is too extreme (e.g. the data is in the range 10000 - 10005), it may be "unreasonable" to choose 0 as the axis starting point. In this case, ChartDirector will not use 0 as the axis starting point.

ChartDirector will determine that it is "unreasonable" to use 0 as the axis starting point if the data fluctuation (the difference between the maximum and minimum data values) is too small compare with the data value. ChartDirector test the "too small" condition using the formula:

```
maxDataValue * zeroAffinity < minDataValue  
where zeroAffinity by default is 0.8.
```

Links:**See Also:**[Chart Object](#), [Chart Director Help Information](#)

4.23 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.

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.

Syntax:
<code>chart.SetxAxisDates(AsSeries(DateSeries), [ElementCount], [Filter], [LabelStep], [Format])</code>

Parameter:	Description:														
DateSeries	<p>The numeric date series, converted using ChartTime. Or if 0 is passed in, the default internal test date/time will be used.</p> <p>Note: Use with all BPV Numeric or String series that are passed to any Chart parameter.</p> <p>AsSeries(...) function conditions the series so the chart function will be able to use the information contained within each of the series elements.</p>														
[ElementCount]	The number of elements in the date series.														
[Filter]	<p>Option is dependent upon date data in the supplied series, and chart axis display area.</p> <table border="1"> <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														

Parameter:	Description:
[LabelStep]	The 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} "

Example:

```

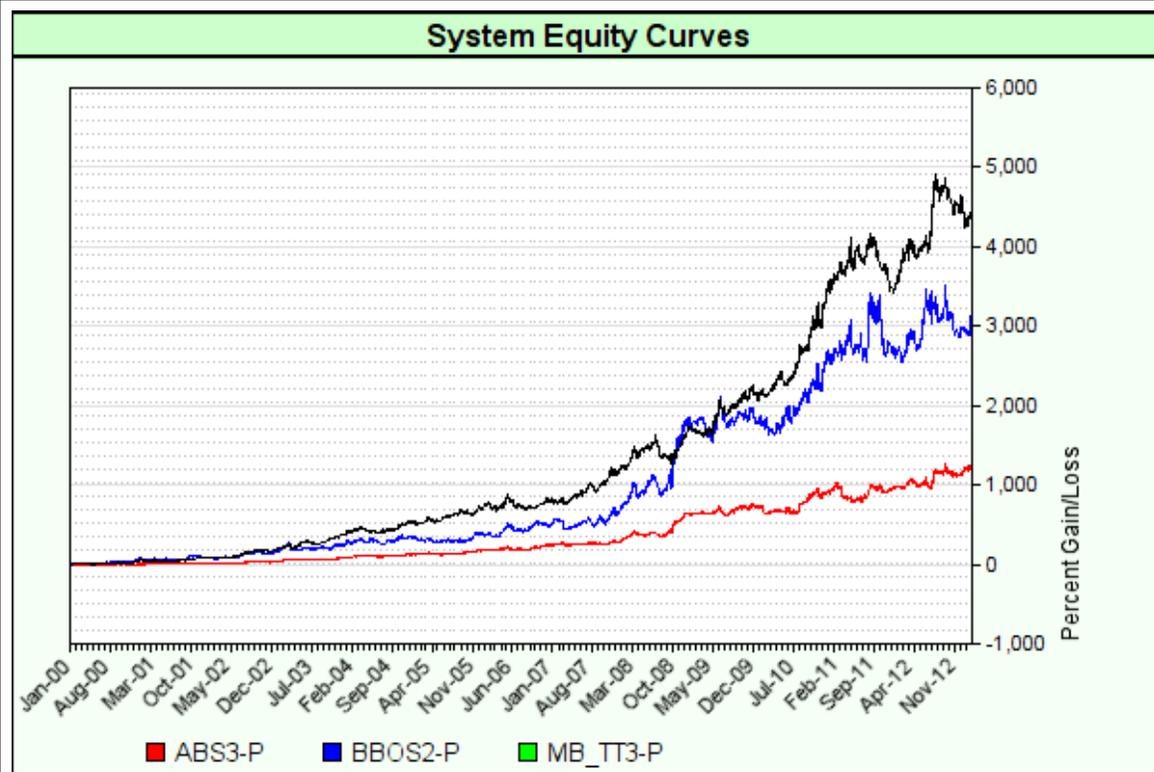
Multi-Line Chart Example in AddLineSeries Examples:

```

```

Set X-Axis Dates to use beginning of month
chart.SetxAxisDates(AsSeries(DateSeries), elementCount, 4)

```

Returns:

Charting SetPlot Area System Equity Curve Details

Links:

[AddLineSeries, AsSeries](#)

See Also:

[Chart](#), [ChartTime](#), [Colors](#), [Chart Director Help Information](#)

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 565

4.24 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.

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 tje 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.

Syntax:

```
chart.SetxAxisLabels( AsSeries(LabelSeries), LabelCount )
```

Parameter	Description:
:	
LabelSeries	A 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	The number of labels contained in the BPV String series.

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 )

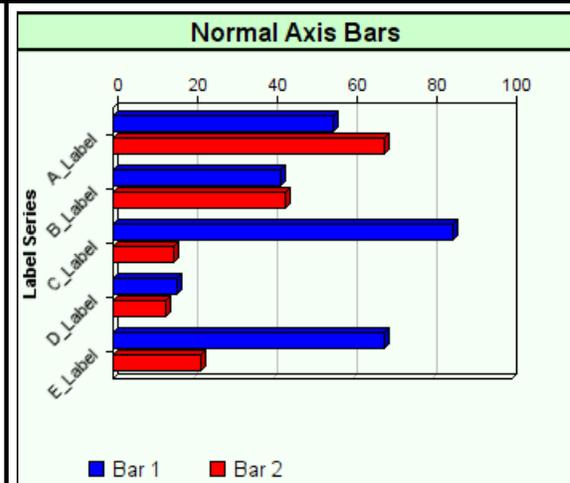
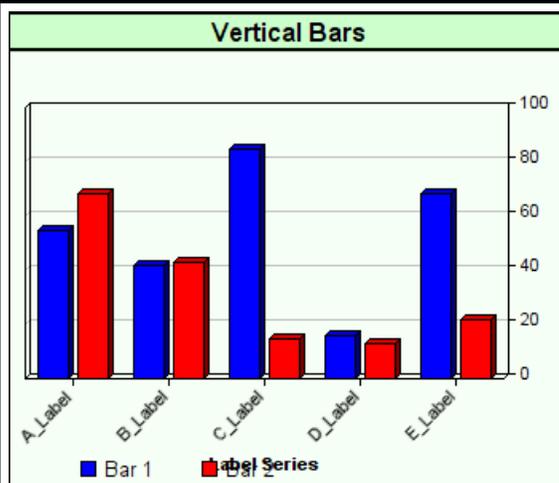
' Add x-Axis Bar Labels
chart.SetxAxisLabels( AsSeries(LabelSeries), iBarCount)

' 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" )

```

Returns:

Links:

[AddBarLayer](#), [AddBarSeries](#), [AsSeries](#), [Make](#), [NewXY](#), [resultsReportPath](#)

See Also:

[Chart](#), [Chart Director Help Information](#)

4.25 SetxAxisLinearScale

Sets the min and max of the x axis scale. This subroutine sets the X-Axis-Scale for a linear range and it requires a Minimum as a starting value and a Maximum value for the ending value.

Syntax:
<pre>' Set the minimum and maximum value for ' the X-Linear Custom Chart Scaling. chart.SetxAxisLinearScale(iMin_X, iMax_X)</pre>

Parameter:	Description:
iMin_X	Integer value of the minimum X-Scale range.
iMax_X	Integer value of the maximum X-Scale Range.

Example:
See Syntax above.
Returns:
No return value. I will set the start and end value of the custom chart's X-scale.

Links:
See Also:
Chart Object , Chart Director Help Information

4.26 Sety2AxisLabelStyle

Syntax:

```
chart.Sety2AxisLabelStyle
```

Parameter:**Description:****Example:****Returns:****Links:****See Also:**

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 92

4.27 Sety2AxisLinearScale

Syntax:
<code>chart.Sety2AxisLinearScale</code>

Parameter:	Description:

Example:
Returns:

Links:
See Also:

4.28 Sety2AxisTitle

Syntax:

Parameter:	Description:

Example:

Returns:

Links:

See Also:

4.29 SetyAxisAutoScale

Syntax:

```
chart.SetyAxisAutoScale(TopMargin, BottomMargin, [Zero Affinity])
```

Parameter:	Description:
TopMargin	
BottomMargin	
[Zero Affinity]	

Example:**Returns:**

No Return

Links:**See Also:**

[Chart Object](#), [Chart Director Help Information](#)

4.30 SetyAxisLabelStyle

Syntax:

```
chart.SetyAxisLabelStyle( Alignment, [Width], [Format] )
```

Parameter:	Description:
Alignment	
[Width]	
[Format]	

Example:

Returns:

Links:

See Also:

[Chart Object](#), [Chart Director Help Information](#)

4.31 SetyAxisLinearScale

This method requires a minimum and a maximum parameter to create the Y-Axis Scale Range values.

Syntax:
<code>chart.SetyAxisLinearScale(iMinY, iMaxY)</code>

Parameter:	Description:
iMinY	Integer value of the minimum Y-Scale range.
iMaxY	Integer value of the maximum Y-Scale range.

Example:
Returns:
No Return

Links:
See Also:
Chart Object , Chart Director Help Information

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.

Parameter :	Description:
EmailAddImage	Function adds an email image to the email being constructed. See examples that show how to include the image tag in the email body.
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

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 108

5.1 EmailAddImage

Function adds an email image to the email being constructed. See examples that show how to include the image tag in the email body.

Syntax:
<In Process>

Parameter:	Description:

Example:
Returns:

Links:
See Also:

5.2 EmailConnect

The EmailConnect function connects to the email server.

Syntax:
<pre>connected = EmailConnect(serverName, [returnEmail], [replyEmail], [userName], [password], [port])</pre>

Parameter:	Description:
serverName	The name of the email server.
[returnEmail]	The return address.
[replyEmail]	The reply address.
[userName]	The username for the email account.
[password]	The password for the email account.
[port]	The connection port.

Example:
Returns:
connected = True, when a connection is successful, and a False when it fails.

Links:
See Also:
Email Manager

5.3 EmailConnectSSL

The EmailConnectSSL function connects to the email server using SSL over Stunnel. Required for sending mail through

Most often the only differences in settings between each mail provider are the server names and the SMTP ports. The SSL port numbers are values that like 465, 587 and other port numbers.

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	The name of the email server.
[returnEmail]	Optional: The return address.
[replyEmail]	Optional: The reply address.
[userName]	Optional: The user's "username-ID" for their email account.
[password]	Optional: The password for the email account.
[port]	Optional: The Out-Going SMTP connection 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 = True, when a connection is successful, and a False when it fails.

Links:**See Also:**

5.4 EmailDisconnect

This function disconnects from the server setup with [EmailConnect](#), and [EmailConnectSSL](#).

This function is required to prevent the email connection from remaining open. To ensure that happens, be sure to execute this command function before the Suite test execution ends.

Syntax:

```
EmailDisconnect()
```

Parameter:	Description:
------------	--------------

<none>	
--------	--

Example:

```
' ~~~~~  
' Disconnect from out-going SMTP mail server.  
EmailDisconnect()  
' ~~~~~
```

Returns:

No information is returned.

Links:

See Also:

5.5 EmailSend

This function sends email using the connection to the email server setup with [EmailConnect](#).

Syntax:
<pre>EmailSend(toList, subject, message, [ccList], [bcclist], [attachments])</pre>

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	The subject of the email.
message	The message body of the email.
[ccList]	Optional: The email cc list.
[bcclist]	Optional: The email bcc list.
[attachments]	Optional: The list of file names to attach. Full path name required. Separate multiple files by a semicolon only -- no spaces.

Example:
<pre>' ~~~~~ 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") ' ~~~~~</pre>
Returns:
No information is returned.

Links:
See Also:
Email Manager

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 277

5.6 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	The subject of the email.
message	The message body of the email.
[ccList]	Optional: The email cc list.
[bccList]	Optional: The email bcc list.
[attachmen ts]	Optional: The list of file names to attach. Full path name required. Separate multiple files by a semicolon only -- no spaces.
[images]	Optional: The image files to send.
[alternate text]	Optional: The alternate text for plain text viewers.

Example:

Returns:

--

Links:

--

See Also:

[Email Manager](#)

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.

- When parameter stepping threads are active, the [FileManager](#) has thread safe protections.
- The number of open files is increased from 100 to 400. However, watch out for Windows limit on concurrent open files.

Functions:	Description:
Close()	Closes an open file.
CountLines()	Returns the number of lines in the file. Takes the file number as a parameter
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
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.
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

Notes:

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.

- 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.
-

Notes:

- 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:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 107

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:

Description:

Parameter	Description
fileNumber	File number of each file opened

Returns:

```
20140531,1.1174000
```

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 & Disk Functions](#)

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:	
<code>iRecordsInFile = fileManager.CountLines(iFileNumber)</code>	

Parameter:	Description:
iFileNumber	The opened file-number integer value returned by the FileManager's Open methods.

Example:
<pre> ' ~~~~~ VARIABLES: sFullPathFileName, sPathName, sFileName Type: String VARIABLES: iFileNumber, iRecordsInFile 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 iFileNumber = fileManager.OpenRead(sFullPathFileName) ' If a File number is assigned, . . . If iFileNumber THEN ' Count the Records contained in the file iRecordsInFile = fileManager.CountLines(iFileNumber) ENDIF ' iFileNumber ENDIF ' FileExists ' ~~~~~ </pre>

Returns:
Number of records contained in the file.

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:

Description:

<none>

Example:

```
' ~~~~~
VARIABLES: sTBPath      Type: String
' ~~~~~
' Get Trading Blox Installation Path
sTBPath = fileManager.DefaultFolder
```

Returns:

Assigns the text containing the Full Path of the installed Trading Blox location.

i.e. C:\Trading Blox\

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:

Description:

fileNumber

The opened file-number identified by one of the Open file methods.

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
' ~~~~~
```

Returns:

endReached is set to **TRUE** when the **End-of-File** marker is reached.

Links:

[OpenRead](#), [ReadLine](#)

See Also:

[File Manager](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 279

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( aFullFileName )
```

Parameter:

Description:

aFullFileName

The full Path & File-name of the file to open.

Example:

```
' ~~~~~
VARIABLES: fileName      Type: String
VARIABLES: fileNumber    Type: Integer
' ~~~~~
' Open file named so additional information can be appended.
fileNumber = fileManager.OpenAppend( fileName )
```

Returns:

fileNumber = Trading Blox assigned file identification number of the file opened so additional information can be added.

Links:

[OpenRead](#), [OpenWrite](#), [WriteLine](#), [WriteString](#)

See Also:

[File Manager](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 438

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 Builder 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:	
<code>fileNumber = fileManager.OpenRead(aFullFileName)</code>	

Parameter:	Description:
aFullFileName	The file and path names for the file to open.

Example:	
<pre> ' ~~~~~ 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 ' ~~~~~ </pre>	
Returns:	
<code>fileNumber</code> = File identification number when file is successfully opened for Read only access.	

Links:

[OpenAppend](#), [OpenWrite](#), [WriteLine](#), [WriteString](#)

See Also:

[File Manager](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 444

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( aFullFileName )
```

Parameter

Description:

:

aFullFile
Name

The file and path names for the file to open.

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
' ~~~~~
```

Returns:

`fileNumber` = File identification number required to access the file when opened successful;
Zero if not successful.

Links:**See Also:**

[File Manager](#)

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:	Description:
------------	--------------

<none>	
--------	--

Example:

```
' Read indexed record as a string
sRecord = fileManager.ReadLine( iFilNum, x )
' Did ReadLine Fail to get the entire record
iNotAllRecordData = fileManager.PartialLine
```

Returns:

iNotAllRecordData = **TRUE** when record could **NOT** read the entire record's contents

Links:

[ReadLine, String](#)

See Also:

[File Manager](#)

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 [,recordToRead] )
```

Parameter:	Description:
<code>fileNumber</code>	The open file identification number to read information.
<code>[,recordToRead]</code>	Optional: The <code>lineString</code> returns the text record at the file's index location number in the <code>recordToRead</code> parameter.

Example:

```
' ~~~~~  
VARIABLES: lineString Type: String  
VARIABLES: fileNumber, recordToRead 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 of text from the current file at the location value in  
recordToRead.  
    lineString = fileManager.ReadLine( fileNumber, recordToRead )  
  
    ' Print the text line  
    PRINT lineString  
  LOOP  
  
  ' Close the file.  
  fileManager.Close( fileNumber )  
ENDIF  
' ~~~~~
```

Returns:

lineString = Text contents of the record read from the file.

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 [, expression2] [, expression2] [, etc... ] )
```

Parameter:	Description:
fileNumber	The file number, when more than one file has been opened.
[, expression]	Optional: The first string expression to write.
[, expression2]	Optional: The second string expression to write.
[, etc...]	Optional: Any other string expression to write.

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  
' ~~~~~
```

Returns:

No information is returned.

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( fileNumber [, expression] [,expression2]
[,etc...] )
```

Parameter:	Description:
fileNumber	The file number, when more than one file has been opened.
[expression]	Optional: The first string expression to write.
[,expression2]	Optional: The second string expression to write.
[,etc...]	Optional: Any other string expression to write.

Example:

```
~~~~~
VARIABLES: lineString Type: String
VARIABLES: fileNumber Type: Integer
' ~~~~~
' Open the file.
fileNumber = fileManager.OpenWrite( "C:\FileToOpen.txt" )

If fileNumber > 0 THEN
' Write out the date to the current file.
fileManager.WriteString( fileNumber, instrument.date, "," )

' Construct another line.
lineString = instrument.symbol + "," + instrument.close

' Write out the line to the current file.
fileManager.WriteString( fileNumber, lineString )

' Write out the end-of-line character sequence.
fileManager.WriteLine( fileNumber )

' Close the file.
fileManager.Close( fileNumber )
ENDIF ' fileNumber > 0
' ~~~~~
```

Returns:

No information is returned.

Links:[Close, OpenWrite](#)**See Also:**[File Manager](#)

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 655

Section 7 – Instrument

All instrument data is specific to each system's portfolio, and the instrument specific data generated in a test.

Instrument Group Section:	Description:
Accessing Instruments	Instrument access is only possible when an instrument is brought into the context of the a script sections. Only script sections that run for each instrument in the portfolio have automatic instrument context.
Correlation Functions	Instrument correlation functions allow you to dynamically control instruments correlations during a test.
Correlation Properties	Instrument correlation properties allow you to access instrument information during a test.
Data Access Properties	Instrument object provides access to all the built-in properties provided to all instruments, and to all the added new property names added by then user.
Data Functions	Data function can be used to round, format, and find instruments during a test.
Group Properties	Group information is one of the primary methods used to manage or identify an instrument during a test.
Historical Trade Properties	Closed trade information is available by accessing any of the historical trade information.
Instrument Loading Functions	These functions can be used in conjunction with an added Instrument Block Permanent Variable Type.
Position Functions	These functions are used to assign or change the value of position property values.
Position Properties	Active trade information is made available by accessing these Instrument Position Properties.
Ranking Functions	The ranking functions are usually used in the Portfolio Manager to set ranking values and to filter the instrument from the portfolio. They can also be used in other script section when a custom portfolio process is needed, or when a ranking property needs to be changed.
Ranking Properties	The ranking properties can be accessed from anywhere in the system where an instrument is accessible.
Trade Control Functions	Trade control functions are designed to allow a portfolio process to control which instruments are allowed to create an order.
Trade Control Properties	These control properties provide a True or False value of the most recent trade control function used.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 101

7.1 Accessing Instruments

Instrument Context:

Instrument Context is a process that provides automatic direct access to instrument information. Script sections where instrument context is automatic by default is shown in the [Blox Script Timing](#) table, and in the table [here](#). During the initialization of test, each instrument's data is automatically loaded into an instrument object. Instrument context makes access to the instrument data easy.

In script sections where instrument context information is not automatically provided, instrument data can still be accessed using the Instrument Object's [LoadSymbol](#) function. The [LoadSymbol](#) function uses the [BPV Instrument Type](#) as a container to access an instrument in a script section where instrument context is not automatically provided. Examples showing how to create the scripts statements is available in the [LoadSymbol](#) topic.

The [LoadSymbol](#) function can also be used to bring additional instruments into a script location where instrument context is automatic. This means the active instrument Trading Blox Builder made available can have a companion instrument brought into the script section at the same time. This ability to access more than one instrument provides information that might influence how the script logic considers. An example of how this can be used is with a spread value difference between two instruments, or a ranking comparison of group of instruments.

Instrument Data:

[Instrument](#) data loaded from a symbol's data file cannot be changed by scripting, but it is easy to retrieve data in script sections where Trading Blox Builder provides [Automatic Instrument Context](#).

Adding other information to an instrument is done by adding an [IPV](#) variable. There is no practical limit to how many variables can be added to an instrument. [IPV](#) additional variables listed in the [IPV](#) list section are available when the instrument is being accessed. Instruments it can be used in testing or reporting. User added information can be changed during a test using script statements.

Instrument Objects have many functions for different purposes. For example, when updating an instrument's protective stop, the [instrument.SetExitStop](#) will update a position's protective exit price after a new price has been calculated so that it is active for the next price bar.

Script sections where [Automatic Instrument Context](#) is not provided, scripting can use the Instrument Object's [LoadSymbol](#) function to gain access to a specific instrument's information. The Instrument Object's [LoadSymbol](#) function can be used in a looping structure that accesses many or all of the instruments in the portfolio.

These script sections get get automatic instrument context.

Automatic Instrument Context Script Sections:	
Rank Instrument	Can Fill Order
Filter Portfolio	Exit Order Filled
Before Instrument Day	After Instrument Open

Automatic Instrument Context Script Sections:	
Exit Orders	Adjust Stops
Entry Orders	Compute Instrument Risk
Unit Size	Adjust Instrument Risk
Can Add Unit	After Instrument Day
Update Indicators	
Note: Click Blox Script Timing to get more information about when these scripts sections are available to access.	

How to know when a script section has instrument context?

Use the Trading Blox Builder Basic property [block.instrumentExists](#). When this property returns a `True` (or a 1) the script section where this property is accessed will have automatic instrument-context. If the return is a `False`, that script section will need the [LoadSymbol\(\)](#) function to access an instrument's information.

Script sections that do not automatically provide instrument context will always need the [LoadSymbol\(\)](#) function. When multiple instruments are looped into the same BPV Instrument variable name, the information in the BPV instrument name will change to last instrument that was accessed.

The [LoadSymbol\(\)](#) function loads symbol's information as a bi-directional connection. This means that information changed in the BPV instrument name that is the load symbol's information container, that change will be represented in the instrument object's information for that symbol. When the instrument is changed in the instrument object in later processing, the information will also change in the BPV instrument information if the same instrument is still in the BPV instrument.

Does [Automatic Instrument Context](#) allow access to IPV variables?

Yes. Any data specific to a symbol is available once the instrument-context has been established. This happens when a script section is automatic-context, and it is available when an instrument is assigned to a BPV instrument type using the [LoadSymbol\(\)](#) function.

Click [LoadSymbol](#) for more information.

BPV Instrument-Context Variable Example:

Block Permanent Variable [Block: Instrument Data Access | Group: _Help]

Script Name: →

Display Name:

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:

Scope:

Reset Before Test

OK

Cancel

Name can be Mkt, Corn, Bonds, LoadedSymbol or many other ideas.

Name replaces the prefix `instrument` object prefix for symbol access .

i.e. `"Mkt.close"`

BPV Instrument Variable Type

This BPV Instrument Type and [LoadSymbol\(\)](#) function can be used with any symbol that is available. It can also be used to access information in another system that is in the same test suite. Access to instruments in other systems in the same suite requires the [test.SetAlternateSystem\(systemIndex\)](#) example.

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.

When an instrument doesn't get automatic context, or an additional instrument is needed in script where it is provided (think spreads), this example can be used to make an additional symbol accessible.

Example:

```
' In a script with Automatic-Context, this is how
' the date of the instrument record is obtained.
variable1 = instrument.date

' In a script section where Automatic Instrument-Context is Not
' provided, this is how an instrument's record date is obtained

' Create a BPV Instrument Type name that will accept an instrument's
' information (See BPV Dialog Above).

' Call LoadSymbol with a instrument's reference.
' Next example is using the instrument's symbol.
' There are more ways for LoadSymbol to bring an
' instrument into context and so the instrument's
' information can be assigned.

' Set the portfolio instrument.
aBpvName.LoadSymbol( "GC" )

' Pass the instrument's date to another variable
anyDate = aBpvName.date
```

Scripts with automatic instrument context will execute the same script name for each of the instruments in the portfolio. This repetitive loop of each instrument is necessary so that all the symbols in the portfolio have a chance to be processed on each test date. Click this topic [Comprehensive Simulation Loop](#) for more information about the testing process actions.

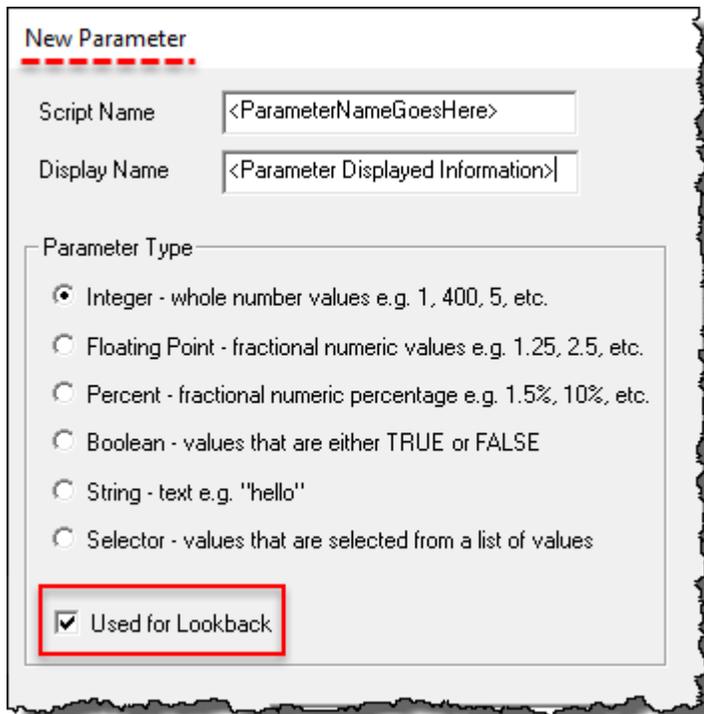
When an instrument doesn't have a data current date the next market date, the instrument record that will be processed with the previous record in the data file. When an instrument does not have a current date for the next market date, the `instrument.tradesOnTradeBar` property will return a `False` value so scripting can prevent a calculation from counting a previous record more than once. When there is a current date for the next market date, the return is `True`.

Instrument Priming:

When indicators are contained in a system, the parameter with the largest calculation look-back value will determine when the system can be allowed to generate orders. This means a system that is expected to generate orders by the Suite Start-Date will need information before that Start-Date so there no Start-Date trading delay.

How much data is needed can be calculated, but the value must be the enough for all the calculations in the system to have begun their calculations prior to the planned Start-Date.

Trading Blox helps to make this possible by examining the parameters in the system that have enabled the parameter's **Look-Back option**:



New Parameter

Script Name: <ParameterNameGoesHere>

Display Name: <Parameter Displayed Information>

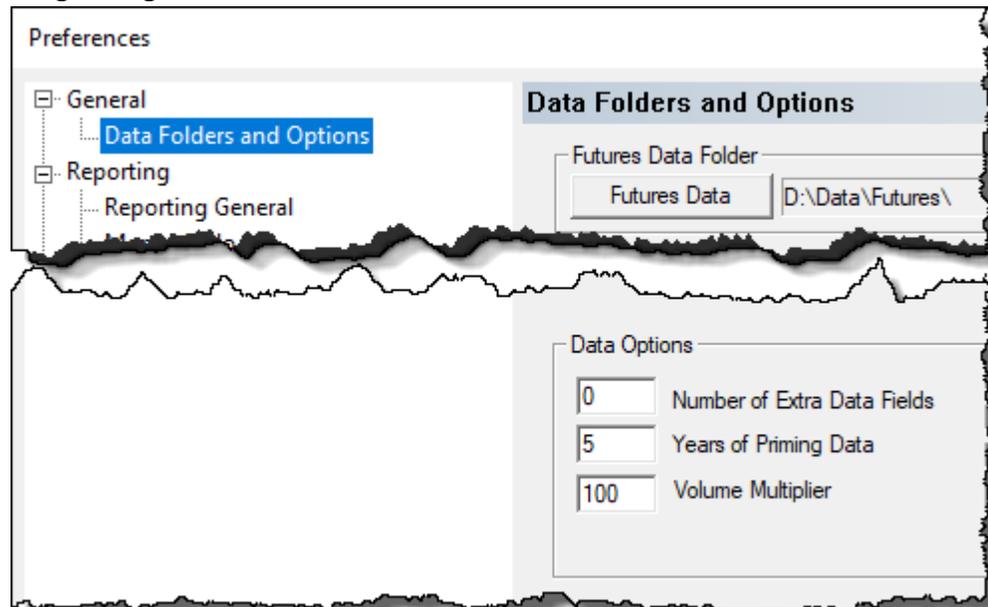
Parameter Type:

- 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

Used for Lookback

When this option is enabled, Trading Blox Builder will use all the available data ahead of the Start-Date so that the calculations will be primed and ready for the system.

Providing data to be available for calculation is controlled by the user's "Years of Priming" setting in the Data Folders and Options section of the Trading Blox Builder Preference setting dialog:



Preferences

- General
- Data Folders and Options**
- Reporting
- Reporting General

Data Folders and Options

Futures Data Folder: Futures Data | D:\Data\Futures\

Data Options

- 0 | Number of Extra Data Fields
- 5 | Years of Priming Data
- 100 | Volume Multiplier

By default, this setting is set at "5" years of information. This means that Trading Blox Builder will attempt to load up to five-years of available records for each of the instruments in system's portfolio. This extra-data value allows the system to begin calculation ahead or on the Start-Date

set for testing to begin. When there isn't enough data to start on the Start-Date, Trading Blox Builder will begin the system's calculation at the earliest possible time.

Some instruments in the portfolio may not have enough trading information ahead of the Start-Date. When the system encounters this condition, any instrument that hasn't primed its calculation will not be allowed to trade until its data is primed. However, once the look-back requirements are met, the Instrument Priming.

To know when an instrument has been primed for the current look-back requirements, use the `instrument.isPrimed` property. When this property returns a `TRUE` (or 1) value, the instrument is ready to test. When a `FALSE` (or a 0) is the returned, the instrument has not been primed.

```
' Is this instrument primed?
If instrument.isPrimed = TRUE THEN
  ' Instrument is ready for testing
ENDIF ' i.isPrimed = TRUE
```

System Testing Sequencing:

When a test is started, Trading Blox Builder creates a list of all the trade day dates that from the start, usually this means just the weekday dates.

The dates used in a test day numbers in the `test.currentDay` property.

Edit Time: 9/11/2020 4:48:24 PM

Topic ID#: 112

Instrument Context

Instrument Context is a process that provides automatic direct access to instrument information. Script sections where instrument context in automatic by default is shown in the [Blox Script Timing](#) table, and in the table [here](#). During the initialization of test, each instrument's data is automatically loaded into an instrument object. Instrument context makes access to the instrument data easy.

In script sections where instrument context information is not automatically provided, instrument data can still be accessed using the Instrument Object's [LoadSymbol](#) function. The [LoadSymbol](#) function uses the [BPV Instrument Type](#) as a container to access an instrument in a script section where instrument context is not automatically provided. Examples showing how to create the scripts statements is available in the [LoadSymbol](#) topic.

The [LoadSymbol](#) function can also be used to bring additional instruments into a script location where instrument context is automatic. This means the active instrument Trading Blox Builder made available can have a companion instrument brought into the script section at the same time. This ability to access more than one instrument provides information that might influence how the script logic considers. An example of how this can be used is with a spread value difference between two instruments, or a ranking comparison of group of instruments.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 57

Instrument Priming

The Years of Priming value in the preferences area shown below is used to determine how much data to load before trading is planned to begin. Data loading prior to the Test Start Date is needed enables the calculations of indicators and other calculations to have enough records available to generate results that will enable trading to begin on the Test Start Date.

Data Options	
<input type="text" value="0"/>	Number of Extra Data Fields
<input type="text" value="5"/>	Years of Priming Data
<input type="text" value="100"/>	Stock Volume Multiplier
<input checked="" type="checkbox"/>	Load Volume
<input checked="" type="checkbox"/>	Load Unadjusted Close
<input type="checkbox"/>	Process Daily Bars
<input checked="" type="checkbox"/>	Process Weekly Bars
<input type="checkbox"/>	Process Monthly Bars

Trading Blox Preference Section

Loading data ahead of when the system is to start trading is called Data-Priming. This means that data ahead of the Test Start Date will be made available to the system. To understand this better, consider an Indicator[lookback] length requires 100 data records be available for calculating the indicator's results so that trading can begin on the Test Start Date. In our example data file, its record information shows it has twenty-years of records ahead of its last record date of today. That file has more than enough data records to load the five years of data and to also enable an indicator that requires a 100-data records to produce results before the Test Start Date. For example, the years shown in the image above will instruct Trading Blox Builder to load five-years of data records ahead of when the Test Start Date is expected to begin trading.

In this example where an indicator needs 100-records, and the priming data will make available 5-years of records. That is more records than this indicator requires. In this case, it will begin generating results 1-record later than the first data date in the primed data file.

Instrument Priming is based on the maximum lookback parameter value, plus the maximum indicator length value. With those number in mind, add one bar to each of those lengths, and one bar to the final tally for the number of data records required for internal Trading Blox Builder reasons. That value is how many records will be required before trading can begin.

So if the Test Start Date is 2017-01-02 and the years of priming data is 0, then the first bar of data will be 2017-01-02. If the Years of priming data is 1 then the first bar of data will be 2016-01-02 – if the data is available. The first bar of data is always the first bar of available data, or the test start date minus the years of priming, whichever is later.

Every block and system is evaluated for its required lookback priming. The priming for a block is the max lookback parameters plus the max indicator lookback. The max of this value over all blox in all systems, is the test priming bars required. The first bar that the Entry Orders script and others will start running is on the Test Start Date, or the first primed bar, whichever is later. So if the Test Start Date is 2017-01-02 but the first primed date is 2017-02-01, then the first evaluated by the Entry Orders script will be 2017-02-01.

The Update Indicators script by default is set to start running on the first primed bar. So this might be before the Test Start Date. This is designed so that custom indicators can be computed and ready before the Test Start Date, and before the Entry Orders script starts running. Enough data is required for this: the number of bars is equal to what is required by the custom indicators, plus what is required for the test priming. So the user should set the Test Start Date to account for this.

The Update Indicators script runs for every instrument on bar of data, so that excludes holidays. The reason is so that this can be used to compute custom indicators correctly.

All non-instrument scripts like Before Instrument Day and After Instrument Day will run on all test days. Test days are typically weekdays, unless process holidays .ini setting is true in which case Sundays are included.

Note that if you set the test start date back in time, and then set the test start date ahead in time, the data is still cached and available. So this may change the priming and start of the update indicators script. Something to keep in mind.

Edit Time: 9/11/2020 4:48:26 PM

Topic ID#: 56

7.2 Correlation Functions

Instrument correlation functions allow you to dynamically control instruments correlations during a test.

Correlation Functions:	Descriptions:
AddCloselyCorrelated	adds an instrument to the close correlation matrix for this instrument
AddLooselyCorrelated	adds and instrument to the loose correlation matrix for this instrument
MaxSynchBars	This function returns the maximum available date synched bars between two series. It was designed to be used with the CorrelationSynch and CorrelationLogSync functions to determine a minimum threshold value, return 0 or some other action.
ResetCloselyCorrelated	removes all instruments from the close correlation matrix for this instrument
ResetLooselyCorrelated	removes all instruments from 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](#)

Parameter:

r:

<none>

Description:**Example:**

```
' Remove all the close correlations.  
instrument.ResetCloselyCorrelated
```

Returns:

Removes all the Closely correlated settings.

Links:

[Correlation Properties](#)

See Also:

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
```

Parameter:**Description:**

```
<none>
```

Example:

```
' Remove all the loose correlations.  
instrument.ResetLooselyCorrelated
```

Returns:

```
Removes all the Loosely correlated settings.
```

Links:

[Correlation Properties](#)

See Also:

AddCloselyCorrelated

Adds the specified market to 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 after a call has been made to the [ResetCloselyCorrelated](#) function.

NOTE:

There is a maximum of 250 loose and 250 close correlations for each instrument. This function is enabled for futures, forex, and stocks.

Syntax:
<code>AddCloselyCorrelated(symbol)</code>

Parameter:	Description:
<code>symbol</code>	The symbol to add to the Closely correlation matrix.

Example:
<pre>' Add gold to the correlation matrix. instrument.AddCloselyCorrelated("GC")</pre>
Returns:
Gold is added as a Closely Correlated symbol.

Links:
Correlation Properties
See Also:

AddLooselyCorrelated

Adds the specified market to 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 after a call has been made to the [ResetLooselyCorrelated](#) function..

NOTE:

There is a maximum of 250 loose and 250 close correlations for each instrument. This function is enabled for futures, forex, and stocks.

Syntax:

```
AddLooselyCorrelated( symbol )
```

Parameter:	Description:
<code>symbol</code>	The symbol to add to the loose correlation matrix.

Example:

```
' Add gold to the correlation matrix.  
instrument.AddLooselyCorrelated( "GC" )
```

Returns:

Gold is added as a Loosely Correlated symbol.

Links:

[Correlation Properties](#)

See Also:

7.3 Correlation Properties

Instrument correlation properties allow you to access instrument information during a test.

Note:

These do not include zero sized trades.

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

Correlation Property Names:	Description:
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
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

Links:

[Correlation Functions](#)

See Also:

7.4 Data Properties

Instrument object provides access to all the built-in properties provided to all instruments, and to all the new property names added by the user.

Bar Indexing

Properties listed with a '[']' following them are series properties and may be indexed using a number to access a previous data record value.

Properties with [] will support access to earlier values:

```
currentInstrumentDate = instrument.date (date of current bar )
currentInstrumentTime = instrument.time (time of current bar if using
intraday data)
priorInstrumentDate = instrument.date[1] (date of prior bar )
( special case: instrument.date[-1] is allowed to see the date for the
next record. Not available on the last record of data. )
```

Lack of [] defaults to 0. So these are the same:

```
todaysClose = instrument.close
todaysClose = instrument.close[0]
```

Using [1] will return the prior bar to the current. This could be the prior day for daily data, or prior minute for minute data.

```
priorClose = instrument.close[1]
```

When a property does not support '[']' like the instrument examples above, those properties are not series and contain values of the current bar location, or their values are static for an instrument. For example, `instrument.bigPointValue` is a static value for each instrument, and `instrument.bar` is reference value that provides the current record number of that instrument.

Data Access Properties:

Property:	Description:
<code>activeStatus</code>	optional value for stocks, A for active and I for inactive
<code>averageVolume</code>	<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>

Property:	Description:
<code>bar</code>	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.
<code>barSize</code>	When used with daily data, the <code>instrument.barSize</code> property will return D1
<code>bigPointValue</code>	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 (see <code>.conversionRate</code>) For Futures it is the value set in the Futures Dictionary in the Big Point column. When it is applied it is adjusted by the currency conversion rate, if there is one. Does not change for futures. For Forex, the current value in dollars for the pair its value can change each day.
<code>brokerBigPointValue</code>	Value is entered into the Futures Dictionary in the Futures Dictionary column Broker BPV . This column is used by the brokerage when their value for their symbol is different than what is used by the data service being used (See <code>bigPointValue</code> for how it is applied.)
<code>brokerSymbol</code>	Broker symbol as defined in the dictionary
<code>close[]</code>	Close price of a specific instrument data record
<code>closeAsk[]</code>	
<code>country</code>	
<code>conversionRate</code>	The conversion rate of foreign denominated futures and stocks. This property respects the "reverse conversion" check-box option in the Forex Dictionary . <code>instrument.conversionRate</code> is not a series. It is a property that is computed on the fly during the test based on the instrument date and the value it calculates from the instrument's date matching Forex currency data. If the Forex conversion file is not current to the latest date the instrument, this property will use the last available record in the Forex file to calculate the conversion rate in all instances after the Forex file's data ends. In most cases the last date won't give an accurate local price so be sure to keep the Forex files current. Note: <code>instrument.conversionRate</code> replaces <code>instrument.forexRate</code> and will be removed in a later version of Trading Blox.
<code>currency</code>	Currency in which the future is denominated. Set in the Futures or Stock Dictionary.
<code>currencyBorrowRate</code>	Borrow rate of the currency.
<code>currencyDate</code>	Current date of the currency converter, if present.

Property:	Description:
<code>currencyLendRate</code>	Lending rate of the currency.
<code>currencyTime</code>	Current time of the currency converter, if present.
<code>currentBar</code>	Bar number minus the startBar plus one.
<code>currentWeek</code>	instrument's weekIndex minus the startWeek plus one.
<code>custom1</code>	Blank column cell for each symbol.
<code>custom2</code>	Blank column cell for each symbol.
<code>customSortValue</code>	Property contains the value assigned by the <code>instrument.SetCustomSortValue()</code> .
<code>dataLoadedBars</code>	Total number of bars of data loaded and cached.
<code>dataVendorID</code>	Optional value for stocks, the data vendor id
<code>date[]</code>	Date value for the specified bar. In YYYYMMDD format.
<code>dayClose[]</code>	Close of the current day as of the current time
<code>dayHigh[]</code>	High of the current day as of the current time.
<code>dayIndex</code>	Current day index for use with the day series which is derived from intraday data
<code>dayLow[]</code>	Low of the current day as of the current time
<code>dayOpen[]</code>	Open of the current day
<code>dayVolume[]</code>	Volume of the current day as of the current time
<code>defaultAverageTrueRange</code>	<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>
<code>deliveryMonth</code>	Delivery month of the contract represented by the data -- format: YYYYMM (futures only)
<code>deliveryMonthLetter</code>	Delivery month letter (Z for December, etc)
<code>description</code>	Description from the appropriate dictionary for this symbol

Property:	Description:
<code>displayDigits</code>	Number of digits to the right of the decimal, as set in the dictionary.
<code>dividend</code>	Dividend for the current bar
<code>endBar</code>	Bar number for the last day of testing for this instrument. Not system specific.
<code>endDate</code>	End date of testing for this instrument. Not system specific. Can be different than the <code>lastDataLoadedDate</code> if the end testing date changes to an earlier date after the data is loaded and cached.
<code>exchange</code>	Exchange where this instrument signal is traded.
<code>exchangeBroker</code>	
<code>exchangeName</code>	Instrument's Exchange Name listed in the Futures Dictionary.
<code>exchangeRegion</code>	
<code>extraData1[]</code> to <code>extraData8[]</code>	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 added column format.
<code>fileName</code>	Filename of the instrument
<code>firstDataLoadedDate</code>	First date of the data loaded and cached for this instrument. Not system specific.
<code>folder</code>	Folder location of the file
<code>forexBaseBorrowRate</code>	Borrow interest rate of the Base side of the Forex pair
<code>forexBaseCode</code>	Provides transparency as to what the interest rate date is used for the base and quote currencies.
<code>forexBaseDate</code>	Provides transparency as to what the interest rate date is used for the base and quote currencies.
<code>forexBaseLendRate</code>	Lending interest rate of the Base side of the Forex pair
<code>forexPipSize</code>	Pip size of the Forex market as set in the Forex dictionary with 7-decimal maximum size
<code>forexPipSpread</code>	Pip spread as set in the Forex dictionary
<code>forexQuoteBorrowRate</code>	Borrow interest rate of the Quote side of the Forex pair
<code>forexQuoteCode</code>	Provides transparency as to what the interest rate date is used for the base and quote currencies.
<code>forexQuoteDate</code>	Provides transparency as to what the interest rate date is used for the base and quote currencies.
<code>forexQuoteLendRate</code>	Lend interest rate of the Quote side of the Forex pair

Property:	Description:
<code>high[]</code>	High for the specified bar
<code>highAsk[]</code>	
<code>industry</code>	
<code>inPortfolio</code>	Returns TRUE if the instrument is in the system's portfolio. When the instrument is not in the portfolio, or the instrument.DisableTrading function has been applied to a symbol in the selected portfolio, this property will return FALSE.
<code>intradayData</code>	Returns TRUE if the instrument is using intraday data
<code>isForex</code>	Returns TRUE if the instrument is a Forex - See Data Class Properties
<code>isFuture</code>	Returns TRUE if the instrument is a Future - See Data Class Properties
<code>isPrimed</code>	Returns TRUE if the instrument is primed. Each instrument is considered primed when the instrument bar count is greater than the longest look-back parameter value that has been enabled in the parameter declaration dialog.
<code>ISIN</code>	
<code>isStock</code>	Returns TRUE if the instrument is a Stock - See Data Class Properties
<code>julianDate[]</code>	Number of days since 1900 for the current bar
<code>lastBarOfDay</code>	Returns TRUE if the bar is the last bar in the day
<code>lastDataLoadedDate</code>	Last date of the data loaded and cached for this instrument. Not system specific.
<code>lastDayOfMonth</code>	Returns TRUE if the bar is the last bar in the month
<code>lastDayOfWeek</code>	Returns TRUE if the bar is the last day in the week
<code>lastDayOfYear</code>	Returns TRUE if the bar is the last bar in the year
<code>lastTradingInstrument</code>	Returns TRUE if the instrument is the last trading instrument for the trading day.
<code>low[]</code>	Low for the specified bar
<code>lowAsk[]</code>	
<code>margin</code>	Margin requirement for a futures instrument as set in the Futures Dictionary. Not used for stocks or Forex. See instrument.usedMargin for each symbol's margin amount in stock or futures systems.
<code>minimumTick</code>	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

Property:	Description:
	calculated as the unadjusted close divided by the adjusted close. In this way, the actual minimum tick for the time period can be determined.
<code>minimumVolume</code>	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.
<code>monthClose[]</code>	Close of the current calendar month, as of the current day
<code>monthHigh[]</code>	High of the current calendar month, as of the current day
<code>monthIndex</code>	Current month index for use with the month series
<code>monthLow[]</code>	Low of the current calendar month, as of the current day
<code>monthOpen[]</code>	Open of the current calendar month
<code>monthVolume[]</code>	Available Monthly volume known for the specified bar. Note: Future contract volume reported on the last bar of the month might not be the exchanges settled volume for that month. Historically, Future contract settled volume values are reported one trade-day after the price record date.
<code>nativeBPV</code>	Native currency big point value, as set in the dictionary's currency valuation.
<code>negativeAdjustment</code>	Use with 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.
<code>open[]</code>	Open for the specified bar
<code>openAsk[]</code>	
<code>openInterest[]</code>	Open interest for the specified bar (when available).
<code>orderSortValue</code>	Default Instrument Order Sort Value at the beginning of a test:

Property:	Description:																																																																																	
	T y p e:	Default Priority Index Order:																																																																																
	F u t u r e s	<p>Futures in a portfolio are sorted and assigned a Priority Index value in ascending order.</p> <p>First sort is by the value in the Futures Dictionary's 'Order Sort Value' field. Symbol records for Futures will be sorted by the value in this field.</p> <p>When more than one symbol in the Dictionary has the same 'Order Sort Value,' the records with the same 'Order Sort Value' will be grouped.</p> <p>Each will have the symbol records within the same-value group sorted in ascending alphabetical order.</p> <p>If all the symbol records in a group have the same value in the 'Order Sort Value' field, the symbols selected for the portfolio will appear in ascending alphabetical order.</p> <table border="1" data-bbox="873 478 1425 1180" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th></th> <th>A</th> <th>B</th> <th>C</th> <th>D</th> </tr> </thead> <tbody> <tr> <td>1</td> <td></td> <td>Test Date:</td> <td>1/2/2017</td> <td></td> </tr> <tr> <td>2</td> <td>Mkt.Date</td> <td>Mkt. symbol</td> <td>Mkt. order Sort Value</td> <td>Mkt. priority Index</td> </tr> <tr> <td>3</td> <td>12/30/2016</td> <td>AD</td> <td>0</td> <td>1</td> </tr> <tr> <td>4</td> <td>12/30/2016</td> <td>EBL</td> <td>820</td> <td>2</td> </tr> <tr> <td>5</td> <td>12/30/2016</td> <td>ED</td> <td>820</td> <td>3</td> </tr> <tr> <td>6</td> <td>12/30/2016</td> <td>TY</td> <td>820</td> <td>4</td> </tr> <tr> <td>7</td> <td>12/30/2016</td> <td>PA2</td> <td>830</td> <td>5</td> </tr> <tr> <td>8</td> <td>12/30/2016</td> <td>LGO</td> <td>1005</td> <td>6</td> </tr> <tr> <td>9</td> <td>12/30/2016</td> <td>FC</td> <td>1010</td> <td>7</td> </tr> <tr> <td>10</td> <td>12/30/2016</td> <td>LC</td> <td>1010</td> <td>8</td> </tr> <tr> <td>11</td> <td>12/30/2016</td> <td>LB</td> <td>1030</td> <td>9</td> </tr> <tr> <td>12</td> <td>12/30/2016</td> <td>BP</td> <td>1250</td> <td>10</td> </tr> <tr> <td>13</td> <td>12/30/2016</td> <td>CD</td> <td>1250</td> <td>11</td> </tr> <tr> <td>14</td> <td>12/30/2016</td> <td>ES</td> <td>2000</td> <td>12</td> </tr> <tr> <td>15</td> <td>12/30/2016</td> <td>NQ</td> <td>2000</td> <td>13</td> </tr> </tbody> </table> <p style="text-align: center; margin-top: 5px;">Default Futures Order Sort Value Priority Index Example.</p>		A	B	C	D	1		Test Date:	1/2/2017		2	Mkt.Date	Mkt. symbol	Mkt. order Sort Value	Mkt. priority Index	3	12/30/2016	AD	0	1	4	12/30/2016	EBL	820	2	5	12/30/2016	ED	820	3	6	12/30/2016	TY	820	4	7	12/30/2016	PA2	830	5	8	12/30/2016	LGO	1005	6	9	12/30/2016	FC	1010	7	10	12/30/2016	LC	1010	8	11	12/30/2016	LB	1030	9	12	12/30/2016	BP	1250	10	13	12/30/2016	CD	1250	11	14	12/30/2016	ES	2000	12	15	12/30/2016	NQ	2000	13
	A	B	C	D																																																																														
1		Test Date:	1/2/2017																																																																															
2	Mkt.Date	Mkt. symbol	Mkt. order Sort Value	Mkt. priority Index																																																																														
3	12/30/2016	AD	0	1																																																																														
4	12/30/2016	EBL	820	2																																																																														
5	12/30/2016	ED	820	3																																																																														
6	12/30/2016	TY	820	4																																																																														
7	12/30/2016	PA2	830	5																																																																														
8	12/30/2016	LGO	1005	6																																																																														
9	12/30/2016	FC	1010	7																																																																														
10	12/30/2016	LC	1010	8																																																																														
11	12/30/2016	LB	1030	9																																																																														
12	12/30/2016	BP	1250	10																																																																														
13	12/30/2016	CD	1250	11																																																																														
14	12/30/2016	ES	2000	12																																																																														
15	12/30/2016	NQ	2000	13																																																																														

Property:	Description:								
	<table border="1"> <tr> <td data-bbox="518 291 565 464">Type:</td> <td data-bbox="565 291 1461 464">Default Priority Index Order:</td> </tr> <tr> <td data-bbox="518 464 565 520"></td> <td data-bbox="565 464 1461 520"></td> </tr> <tr> <td data-bbox="518 520 565 646">Forecast</td> <td data-bbox="565 520 1461 646">All portfolio symbols are sorted in ascending alphabetical order.</td> </tr> <tr> <td data-bbox="518 646 565 772">Stocks</td> <td data-bbox="565 646 1461 772">All portfolio symbol are sorted in ascending alphabetical order.</td> </tr> </table>	Type:	Default Priority Index Order:			Forecast	All portfolio symbols are sorted in ascending alphabetical order.	Stocks	All portfolio symbol are sorted in ascending alphabetical order.
Type:	Default Priority Index Order:								
Forecast	All portfolio symbols are sorted in ascending alphabetical order.								
Stocks	All portfolio symbol are sorted in ascending alphabetical order.								
<p><code>priorityIndex</code></p>	<p>At the start of testing, the default symbol order in which the instruments are listed is explained by the description in the '<code>.orderSortValue</code>' property above.</p> <p>Trading Blox Builder provides instrument Ranking Functions and Ranking Properties. It also has System functions Rank Instruments and SortInstrumentList function that process the instrument ranking assignment changes and the sort the portfolio list of symbols based up the SortInstrumentList option used for the sorting. Some of the better rules for ranking instruments are those created by the user so they meet their system's needs.</p> <p>As part of the changes created by the ranking and sorting of instruments in the portfolio, the <code>instrument.priorityIndex</code> assignments are updated so the sorted list of symbols has an ascending list of index values.</p> <p>Instruments are processed for each test date record using the '<code>.priorityIndex</code>' property index. An instrument with an '<code>.priorityIndex</code>' value of one is processed first, a index value of two is processed next. Each remaining instrument in a portfolio is then process using the next ascending index number,</p> <p>Ranking and sorting that changes the priority index assignments can be performed for each test date.</p> <p>Each instrument in a blox, are automatically assigned instrument context. Those script section will process the portfolio instruments in the same way. Accessing an instrument in a script section where instrument context isn't automatically assigned, will need the LoadSymbol function to access an instrument.</p>								

Property:	Description:
<code>referenceID</code>	This property is a special object pointer that is used by custom DLL extension functions to access instrument object information.
<code>roundLot</code>	Returns the round lot of the instrument, as set in the dictionary
<code>savedWFPProfit</code>	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.
<code>sector</code>	
<code>startBar</code>	Bar number of the first day of testing for this instrument taking into consideration the priming required for this instrument for this system.
<code>startDate</code>	Start date of testing for this instrument and system taking into consideration priming. Is usually different than the firstDataLoadedDate.
<code>startWeek</code>	The weekIndex of the startBar
<code>stockSplitRatio</code>	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.
<code>symbol</code>	Instrument's trading symbol, e.g. S, IBM, CL
<code>symbolWithType</code>	Returns the symbol in "F:GC" or "S:IBM" format. Use when accessing a symbol outside of the system where the symbol portfolio is located. For example in a LoadSymbol operation in a GSS blox.
<code>systemClosedEquity</code>	Current system closed equity for the instrument
<code>systemOpenEquity</code>	Current system open equity for the instrument
<code>systemTotalEquity</code>	Current total system profit/loss for the instrument
<code>testClosedEquity</code>	Current test closed equity for the instrument
<code>testOpenEquity</code>	Current total test open equity for the instrument
<code>testTotalEquity</code>	Current total test profit/loss for the instrument
<code>time[]</code>	Time value for the specified bar. 0 if daily data. In HHMM format.
<code>tradeDayOpen</code>	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

Property:	Description:
	for the current test date/time. Important to use when excluding holidays from a computation.
tradesOnTradeDate (Obsolete)	This Instrument property has been obsoleted, but it is still active in some scripts. When this property is used it will provide the same information as instrument.tradesOnTradeBar , but at some point a Trading Blox Builder upgrade version might remove it from the active instrument properties.
tradingMonths	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.
unadjustedClose	Actual close price unadjusted for contract merging (futures), splits, or dividends (stocks)
unAdjustedVolume	Volume for stocks, unadjusted by stock splits. Typically the the raw OHLC and V in the data series are all adjusted for stock splits.
usedMargin	Total margin used for the current open position. Value returned is purchase equity for stocks and total margin for futures. See instrument.margin for futures margin amount.
volume[]	Available volume known for the specified bar. Note: Future contract volume reported on the last bar of the day, week or month might not be the exchanges settled volume for that day, week or month. Historically Future contract settled volume is reported one trade-day after the reported price record date.
weekClose[]	Close of the current calendar week, as of the current day
weekHigh[]	High of the current calendar week, as of the current day
weekIndex	Current weekIndex used for the week series.
weekLow[]	Low of the current calendar week, as of the current day
weekOpen[]	Open of the current calendar week
weekVolume[]	Available Weekly volume known for the specified bar. Note: Future contract volume reported on the last bar of the week might not be the exchanges settled volume for that week. Historically Future contract settled volume is reported one trade-day after the reported price record date.

Links:

[Data Functions](#)

See Also:

[Unit Indexing](#)

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 388

Data Class Properties

Trading Blox Builder has three Instrument and one System property that can be used to know the data class specified by the system's selected portfolio.

Instrument Object: `.isForex`, `.isFuture`, & `.isStock`

System Object: `.portfolioType`

These instrument properties and the system property return the asset class of the instrument. The instrument asset class is assigned during data load phase of a system/suite test. Result returned is based on which of the selected data class options is selected.

`Instrument` properties return a `TRUE`, " 1 ", or a `FALSE`, "0"

- For example, stocks can be loaded from the stock dictionary, or directly from the data file.
- Futures and Forex instruments are loaded from the Forex or Futures Dictionaries.
- Any instrument loaded from a dictionary will be assigned that dictionary's class designation.

The System object property called "`.portfolioType`" that returns the portfolio selection, `stock`, `futures`, or `forex`.

System property will return a string result

```
FX = "Forex"
F  = "Future"
S  = "Stock"
```

Syntax: - Class Data = Instrument Property

```
' When a Forex instrument is loaded so it is
' in context, use this to determine
If instrument.IsForex THEN
' Update Instrument Class
sInstrumentClass = "Forex"
ELSE
If instrument.IsFuture THEN
' Update Instrument Class
sInstrumentClass = "Future"
ELSE
If instrument.IsStock THEN
' Update Instrument Class
sInstrumentClass = "Stock"
ENDIF ' i.IsStock
ENDIF ' i.IsFuture
ENDIF ' i.isForex
```

Syntax: - Class Data = System Property

```
' Get System Portfolio Class
sPortfolioClass = system.portfolioType
' Or
PRINT "system.portfolioType :", system.portfolioType
```

Parameter:	Description:
<none >	

Example: - Class Data: Instrument Context - System All Scripts

```

' =====
' BEFORE TEST - START
' =====

PRINT
PRINT
' Show Script Section name
PRINT "Script Name: ", block.scriptName
' Show System's Portfolio name
PRINT "Portfolio Name: ", system.portfolioName
' Show System Selected Instrument Class
PRINT "system.portfolioType :", system.portfolioType
PRINT " -----"

' =====
' BEFORE TEST - END
' =====

===== Script_Change:
' =====
' BEFORE INSTRUMENT DAY - START
' =====

PRINT
' Show Script Section name
PRINT "Script Name: ", block.scriptName
' Show System's Portfolio name
PRINT "Portfolio Name: ", system.portfolioName

' Get System Portfolio Class
sPortfolioClass = system.portfolioType
PRINT "system.portfolioType :", system.portfolioType

' When a Forex instrument is loaded so it is
' in context, use this to determine
If instrument.IsForex THEN
' Update Instrument Class
sInstrumentClass = "Forex"
ELSE
If instrument.IsFuture THEN
' Update Instrument Class
sInstrumentClass = "Future"
ELSE
If instrument.IsStock THEN
' Update Instrument Class
sInstrumentClass = "Stock"
ENDIF ' i.IsStock
ENDIF ' i.IsFuture
ENDIF ' i.isForex

PRINT
PRINT "Instrument Portfolio Class is: ",
PRINT " -----"

' =====
' BEFORE INSTRUMENT DAY - END
' =====

```

Returns: - Class Data: Instrument Context - System All Scripts

```
' -----  
' Example Output was created by changing the System Instrument  
' Type selection and system portfolio from one available in  
' of the three instrument type options.  
' -----  
Script Name: Before Test  
Portfolio Name: All Sample  
system.portfolioType : FX:  
-----  
Script Name: Before Instrument Day  
Portfolio Name: All Sample  
system.portfolioType : FX:  
Instrument Symbol: AUDUSD  
  
Instrument Portfolio Class is: Forex  
-----  
Script Name: Before Test  
Portfolio Name: Any Stock  
system.portfolioType : S:  
-----  
Script Name: Before Instrument Day  
Portfolio Name: Any Stock  
system.portfolioType : S:  
Instrument Symbol: AAPL  
  
Instrument Portfolio Class is: Stock  
-----  
  
Script Name: Before Test  
Portfolio Name: AnySelected  
system.portfolioType : F:  
-----  
Script Name: Before Instrument Day  
Portfolio Name: AnySelected  
system.portfolioType : F:  
Instrument Symbol: CT2  
  
Instrument Portfolio Class is: Future  
-----
```

Links:

[Data Properties](#), [System Properties](#)

See Also:

TradesOnTradeBar

Property returns a **TRUE** value when an [instrument.date](#) available to match a [test.currentDate](#) during the Brokerage, Fill, and Update Instrument scripts. When the instrument skips a [test.currentDate](#) value this property will return a **FALSE** value.

NOTE:

Instrument property: [tradesOnTradeDate](#) has been obsolete but it is still active. When used it will provide the same information as [instrument.tradesOnTradeBar](#), but at some point a Trading Blox Builder version release might remove it from the active instrument properties. When reviewing blox where the previous property name is being used, it would be best to change to the current name to prevent unexpected script failures in future versions.

Image on the left shows simulation results where the script sections with normal instrument context do not execute when there is a gap in the [instrument.date](#) records relative to the [test.currentDate](#).

Script sections, [Exit Orders](#), [Entry Orders](#), and [Update Indicators](#) don't execute when the most recent instrument record date will not match the test date.

Script sections [Initialize Risk Management](#) and [Compute Risk Adjustments](#) don't have normal instrument context are shown because they are the default scripts associated with a [Risk Manager](#) blox.

When an script section with normal instrument context executes where its instrument date won't match the test date the instrument property values will be from the most recent previous instrument date that was able to match its companion test date.

Use this property as a control or an alert when it is calculating instrument information that is delayed by a gap in the current instrument dates.

ScriptName	test Day	Mkt-Symbol	test-date	test-Date	Mkt Date	Mkt Date	Date-Diff	Trades	TradesOnBar
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test									
4.3.5.4									
41995									
1/22/2015									
TradesOnTradeDate Suite									
TradesOnTradeDate Script Test					</				

Example:

```
' -----  
' Prevent any instrument calculations when there  
' is a missing date gap between the current instrument  
' date and the executing test.currentDate  
If instrument.tradesOnTradeBar THEN  
' Perform normal instrument calculations.  
' [ Do Something in this area ]  
ENDIF ' i.tradesOnTradeBar = TRUE  
  
' -----  
' Change how instrument calculations are handled when  
' there is a date gap between latest instrument date and  
' current test date.  
If instrument.tradesOnTradeBar = TRUE THEN  
' Perform normal instrument calculations.  
' [ Do Something in this area ]  
  
ELSE ' i.tradesOnTradeBar = FALSE  
' Perform normal instrument calculations.  
' [ Do Something ELSE, or nothing in this area ]  
  
ENDIF  
' -----
```

Returns:

True = 1. It is only returned when there is an instrument record for the next test date.

Links:**See Also:**

[Data Properties](#)

7.5 DataFunctions

Data function can be used to round, format, and find instruments during a test.

When using any of the instrument Functions or Properties, the Object name: `instrument`. must be appended ahead of the Function or Property name: i.e. `instrument.SetRoundLot(1)`

Function Name:	Description:
<u>AddCommission</u>	Adds the specified amount to the commission to the specified unit of the current position for the instrument.
<u>DisableTrading</u>	Disables symbol in the portfolio the user does not want the system to use for a specific test.
<u>Extract()</u>	Extracts all indicators and IPV Series variable by day and by instrument to a file.
<u>GetDateTimeIndex()</u>	Returns the time index within a date.
<u>GetDayIndex()</u>	Returns the day index for a date
<u>PriceFormat()</u>	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.
<u>RealPrice()</u>	Returns the real price, converted back from being negative adjusted.
<u>RoundTick()</u>	$\text{Rounded_Price} = (\text{price} + \text{minimumTick} / 2)$
<u>RoundTickDown()</u>	Returns the price rounded down to the next tick
<u>RoundTickUp()</u>	Returns the price rounded up to the next tick
<u>SetMinimumTick()</u>	This function takes two parameters, tick-size and the decimals that are optional decimals for whole integer numbers. See the description <u>instrument.MinimumTick</u> value above.
<u>SetRoundLot()</u>	Changes the value of the <u>instrument.RoundLot</u> property value.
<u>SetSeriesAutoIndex()</u>	This function, <code>instrument.SetSeriesAutoIndex(series-name, True/False)</code> can change a series instrument from its Auto-Index to Manual Index . To change the series indexing, this function requires two parameters. In the first parameter enter the name of the series. In the second parameter, enter <code>True/False</code> . This function will work with IPV and BPV and Numeric and String series.
<u>SetSeriesEnable()</u>	This series function, <code>instrument.SetSeriesEnable(series-name, True/False)</code> can Disabled or Enable an instrument's indicator series. To change the series operating state, enter the IPV series-name in the first parameter, and enter <code>True</code> or <code>False</code> as the second parameter, enter.

Function Name:	Description:
	This function will work with IPV series or regular indicators. It cannot be used with the built-in calculated indicators that are created in the Instrument section that are initialized at the start of a System-Test.
SetSeriesValueType()	This series Data Function SetSeriesValueType() can only be used in Set Parameters script.

Links:
Data Properties
See Also:
Data Properties

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.

Example below is placed in the **Entry Order Filled** script section and it shows adding commission amount as part of the Order Filled process.

Syntax:
<code>instrument.AddCommission([unitNumber], commission)</code>

Parameter	Description:
:	
<code>[unitNumber]</code>	The unit number to which the commission will be added.
<code>commission</code>	The commission amount to add per share or contract, in base system currency.

Example:
<pre>' 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)</pre>
Returns:
A commission of \$12 is added to the new position unit

Links:
Data Functions
See Also:

DisableTrading

This function will remove instrument from the portfolio. It removes access to the instrument by changing the state of the symbol's [inPortfolio](#) property. When an instrument's symbol is included in the loaded portfolio, the [instrument.inPortfolio](#) property has a value of **TRUE**. When an instrument portfolio, and the [instrument.inPortfolio](#) property is **FALSE**, Trading Blox Builder will not use the symbol.

This Function is an [Instrument Data Function](#). It should be run in the [Set Parameters](#) script section before the data is loaded, and the system's indicators perform the system's indicator calculations. The Set Parameters script section was implemented so the memory needed for each symbol is needed, and the time it takes to load the data. By changing the access state of a symbol to False, the time to load, and calculate indicators isn't used and the memory required isn't needed.

Previously, this function was executed in the **BEFORE TEST** script so the time saving and the memory used is wasted because . When it is used in that script section the instrument's data will be loaded and the system's indicator calculations will run because the instrument's [inPortfolio](#) property is **TRUE** until the [DisableTrading](#) function changes the property to **FALSE**.

The [instrument.inPortfolio](#) property value can be changed by using the [instrument.DisableTrading\(index\)](#). Changing the state of the [instrument.inPortfolio](#) property is best done in the **Set Parameters** script section. The [Set Parameters](#) script section executes ahead of the Before Test script section where the instruments in the portfolio are loaded and used in the calculation of indicators.

When an instrument in the portfolio is removed from being accessible by using the [Set Parameters](#) script section, the data won't consume any time to load it, and it won't need any memory space to hold any calculated indicators.

can change the instrument's is changed by the [instrument.DisableTrading](#) function, it [instrument.inPortfolio](#) value will change from a 1 = (**TRUE**) to a 0 = (**FALSE**).

These script section, [Exit Orders](#), [Entry Orders](#), [Update Indicators](#), won't execute for an instruments when its [instrument.inPortfolio](#) property is **FALSE**.

This function does not remove a symbol's character text from the portfolio's save list of symbols, those will always be in place until changed using the [Portfolio Manager](#). This function's ability allows the trader to load a large list of instrument symbols and provides a method for removing all the symbols the user decides remove from the symbols from being accessed and traded by the system. When the symbols are removed, the processing of the above script section will not execute. By removing the listed scripts from running during a test, the time save will enable Trading Blox Builder to run faster. For example, when a portfolio of 10,000 stock symbols listed, but the system test only wants to allow 700 of those symbols be used the system test, removing .

Syntax:

Syntax:

where none of the instruments will have context. In the example line below, the **MktSymbol** is a **BPV** Instrument Type variable. When the [LoadSymbol](#) loads a symbol, all the normal properties and function in an instrument object are assigned to the MktSymbol BPV Type Instrument.

```
' Example of how to use this function when it is
' it is not used in the Set Parameters script section.
MktSymbol.DisableTrading( index )

' When this function is used in a custom script section
' that is called from the Set Parameters script, the
' custom function will provide the instruments in the
' custom script section to with automatic context.
' The calling script function script.InstrumentLoop will
' automatically cause the instruments to loop through
' Assigned portfolio list of symbols.
instrument.DisableTrading
' Will not need the index parameter to change its
' instrument.inPortfolio property state from True
' to False. See example in script.InstrumentLoop
```

Parameter:	Description:
index	<p>The symbols location in the portfolio is the row-location of the symbol to remove. See the examples below to how it works.</p> <p>Index parameter is not needed and not allowed after the symbol has been loaded by the LoadSymbol function. When a symbol is loaded, the DisableTrading function will be applied to the MktSymbol (BPV=Instrument) instrument assigned by the LoadSymbol function.</p>

Example:

```

' =====
' Portfolio Instrument Disabling
' BEFORE TEST SCRIPT - START
' =====
' ~~~~~
' Variables:
'   MktSymbol = a BPV Instrument Object Type
'   iIndex    = BPV Integer
'   iEnabled  = BPV Integer
'   MktSymbol.inPortfolio = instrument.inPortfolio property
'   sHeader   = Column Title Header
' -----
' Column Title Header
sHeader = "Index," _
         + "Symbol," _
         + "PriorityIndex," _
         + "InPortfolio"
' Clear Enabled Symbol Counter
iEnabled = 0
' Disable The Number of Instrument Symbols in the Portfolio
PRINT "Total Instruments: ", system.totalInstruments
' Send Column Titles
PRINT sHeader

' Load Each Portfolio Instrument into the Mkt Instrument Object
' and disable every other instrument in this test to show the
' current i.inPortfolio value, and the i.inPortfolio value
For iIndex = 1 TO system.totalInstruments STEP + 1
  ' Load symbol in row 'iIndex' location
  MktSymbol.LoadSymbol(iIndex)

  ' Display the symbol, symbol's priorityIndex, and the
  ' status of whether the index will be used by the
  ' system, or it is removed from being available.
  PRINT iIndex, _
        MktSymbol.symbol, _
        MktSymbol.priorityIndex, _
        MktSymbol.inPortfolio

  ' Count Any Symbol that is Enabled for System Testing
  If MktSymbol.inPortfolio = TRUE THEN iEnabled = iEnabled + 1

  ' When status value shows instrument.inPortfolio
  ' value is 1, that indicate the instrument will be
  ' Available to the system.
  '
  ' When an instrument is available, the DisableTrading
  ' Function will execute and change the the
  ' instrument.inPortfolio value to zero.
  '
  ' NOTE: When a LoadSymbol is assigned to the Mkt Object,
  '       DisableTrading function won't use the value
  If iIndex % 2 = 1 THEN MktSymbol.DisableTrading
Next

```

Example:

```

' Show the original the Number of Enabled Symbols
PRINT "Enabled Symbols:", "", iEnabled
' -----
' Clear Enabled Symbol Counter
iEnabled = 0

' Show the current count of all the instruments in the portfolio
PRINT
PRINT "Total Instruments: ", system.totalInstruments

' Send Column Titles
PRINT sHeader

' Loop through each symbol in the portfolio again
For iIndex = 1 TO system.totalInstruments

    ' Assign each instrument symbol to the
    ' MktSymbol Object
    MktSymbol.LoadSymbol(iIndex)

    ' Count Any Symbol that is Enabled for System Testing
    If MktSymbol.inPortfolio = TRUE THEN iEnabled = iEnabled + 1

    ' Show the inPortfolio Status of the portfolio's
    ' symbols again
    PRINT iIndex, _
           MktSymbol.symbol, _
           MktSymbol.priorityIndex, _
           MktSymbol.inPortfolio
Next    ' iIndex

' Show the original the Number of Enabled Symbols
PRINT "Enabled Symbols:", "", iEnabled
' ~~~~~
' =====
' Portfolio Instrument Disabling
' BEFORE TEST SCRIPT - END
' =====

```

Returns:

Returns:

Total Instruments: 20			
Index	Symbol	PriorityIndex	InPortfolio
1	ACWI	1	1
2	BIB	2	1
3	CIU	3	1
4	CURE	4	1
5	DIA	5	1
6	DSI	6	1
7	EFA	7	1
8	EWM	8	1
9	HDV	9	1
10	ITOT	10	1
11	LQD	11	1
12	SCHB	12	1
13	SH	13	1
14	TBF	14	1
15	TLT	15	1
16	VWO	16	1
17	XLE	17	1
18	XLI	18	1
19	XLP	19	1
20	XLV	20	1

Enabled Symbols: 20

Example Portfolio Before Symbols Disabled

Returns:

Total Instruments: 20			
Index	Symbol	PriorityIndex	InPortfolio
1	ACWI	-1	0
2	BIB	2	1
3	CIU	-1	0
4	CURE	4	1
5	DIA	-1	0
6	DSI	6	1
7	EFA	-1	0
8	EWM	8	1
9	HDV	-1	0
10	ITOT	10	1
11	LQD	-1	0
12	SCHB	12	1
13	SH	-1	0
14	TBF	14	1
15	TLT	-1	0
16	VWO	16	1
17	XLE	-1	0
18	XLI	18	1
19	XLP	-1	0
20	XLV	20	1
Enabled Symbols:		10	

Example Portfolio After Symbols Disabled

Links:

[LoadSymbol](#), [instrument.priorityIndex](#), [instrument.inPortfolio](#),
[system.totalInstruments](#)

See Also:

[Instrument Data Function](#), [Portfolio Manager](#)

Extract

The Extract Function can be used data series of as **System** object or **Instrument** Object process. In order for Extract to work, it must be used with a data series that has not been disabled.

Notes:

When used with an **instrument** object, it will export to a file the value for each series and indicator for that instrument, for each day in the test.

When used with a system object, it will export for all instrument in the system.

We recommend only running this once per instrument per test run, as a file is created each time the function is called.

By default, the extra file name will be “**Daily Extract XXX.csv**” where **XXX** is the symbol, and the extract will contain all available dates that have been loaded.

The file is placed at the top level of the Trading Blox Builder installation directory.

Caution:

Be careful with the `system.Extract` because each symbol in the portfolio will create an export file.

When there are many symbols in a portfolio file, the Extract function will create a file for each symbol, which might be more than what you intended.

Syntax:	
' Export optional parameters	<code>instrument.extract([ExtractedFileName] [,StartingDate] [,EndingDate])</code>
OR	
' Use default settings	<code>instrument.extract()</code>
OR	
' Export all series data between 19990101 and 20060101, and use the default file name:	<code>instrument.Extract(19990101, 20060101)</code>
OR	
' Create a file named "My Custom Name XXX.csv" for each instrument, and export all the series data between 19990101 and 20060101:	<code>instrument.Extract("My Custom Name", 19990101, 20060101)</code>

Parameter:	Description:
<code>[ExtractedFileName]</code>	Optional: Default file-name field is used when field is left empty.
<code>[,StartingDate]</code>	Optional: Beginning data date when start-date field is left empty.

Parameter:	Description:
[,EndingDate]	Optional: Ending data date when end-date field is left empty.

Example:

```
' Typical usage in the After Test script:
inst.LoadSymbol("GC")
inst.Extract()

OR
system.Extract()

OR
inst.LoadSymbol("GC")
If inst.Extract( "Output Series\\Symbol" ) THEN
    PRINT "Extract successful"
ELSE
    PRINT "Extract unsuccessful"
ENDIF
```

Returns:

The return value is TRUE if successful and FALSE if not. In this example, we output to a custom folder within the Trading Blox install directory. If this folder does not exist, the function will fail.

Date	Time	Open	High	Low	Close	Volume	Open Interest	Delivery Month	Un-adjusted Close	macd Indicator	average True Range	entry Breakout High	entry Breakout Low	exit Breakout High	exit Breakout Low	average True Range	offset Adjusted Entry High	offset Adjusted Entry Low	offset Adjusted Exit High	offset Adjusted Exit Low	testing
20110419	0	1551.8	1555.7	1543.4	1550.3	138785	537946	201106	1495.1	124.407	18.627	1555.700	1466.700	1555.700	1468.700	18.627	1555.700	1466.700	1555.700	1468.700	1546.180
20110420	0	1551.3	1561.7	1549	1554.1	155371	537767	201106	1498.9	125.260	18.331	1561.700	1466.700	1561.700	1468.700	18.331	1561.700	1466.700	1561.700	1468.700	1551.354
20110421	0	1557.2	1564.8	1555.6	1559	141956	531503	201106	1503.8	126.164	17.949	1564.800	1466.700	1564.800	1484.300	17.949	1564.800	1466.700	1564.800	1484.300	1555.836

Instrument.Extract Output Example from last script section above.

Links:**See Also:**

[Instrument Data Functions](#), [System Functions](#)

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.

Note:

Function will returns `-1` when the date and or time cannot be not found in the instrument series.

Syntax:

```
barIndex = instrument.GetDayIndex( date, time )
```

Parameter:**Description:**

date	The date in YYYYMMDD format.
time	The time in HHMM format.

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
```

Returns:

Displays the close price and date-time values of the specified instrument record.

Links:

[Data Functions](#)

See Also:

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.

Note:

Returns -1 if the date is not found in the instrument series.

Syntax:

```
barIndex = instrument.GetDayIndex( date, [time] )
```

Parameter:**Description:**

date	The date in YYYYMMDD format.
[time]	Optional: The 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
```

Returns:**Links:**

[Data Functions](#)

See Also:

PriceFormat

This function returns a string value, and cannot then assigned to a floating point variable or used in any computations.

Note:

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.

The return value should only be used for Printing or Reporting purposes. Returns the price formatted for printing. For example, a price for Soybeans of 6.25 will return "6 1/4".

Syntax:

```
roundedValue = PriceFormat( price )
```

Parameter:	Description:
price	The price to be formatted.

Example:

```
' Print the price.  
PRINT "Price = ", instrument.PriceFormat( entryPrice )
```

Returns:

Returns a STRING type variable.

Links:

[Data Functions](#)

See Also:

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.

Notes:

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.

Syntax:

```
rPrice = RealPrice( price )
```

Parameter:**Description:**

price

The price value to adjust.

Example:

```
' Print the real price for debugging
PRINT instrument.RealPrice( instrument.close )
```

Returns:

Price returned removes the software's adjusted step so the price on the date for the current record is the value printed.

Links:

[Data Functions](#)

See Also:

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 )
```

Parameter:	Description:
------------	--------------

price	The price to be rounded.
-------	--------------------------

Example:

```
' Round the price to the nearest tick.
entryPrice = instrument.RoundTick( entryPrice )
```

Returns:

Links:

[Data Functions](#)

See Also:

RoundTickDown

Rounds the specified price rounded down to the nearest tick using the instrument's tick value.

Syntax:

```
roundedValue = RoundTickDown( price )
```

Parameter:**Description:**

Parameter:	Description:
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 )
```

Returns:

The adjusted price rounded down the next market required value.

Links:

[Data Functions](#)

See Also:

RoundTickUp

Rounds the specified price rounded up to the nearest tick using the instrument's tick value.

Syntax:

```
roundedValue = RoundTickUp( price )
```

Parameter:	Description:
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 )
```

Returns:

Rounds the adjusted a price up to the next market required value.

Links:

[Data Functions](#)

See Also:

7.6 Group Properties

Group information is one of the primary methods used to manage or identify an instrument during a test.

Note:

These do not include zero sized trades.

Group Index Categories:
1 = <code>group1/industry</code>
2 = <code>group2/sector</code>
3 = <code>country</code>
4 = <code>currency</code>

Dictionary Group Categories:	Description:
<code>group1</code>	Name of the group1. Set in the Futures or Stock Dictionary
<code>group2</code>	Name of the group2.
<code>industry</code>	Same as group1. Used for stocks.
<code>sector</code>	Same as group2. Used for stocks.
<code>country</code>	Name of the country. Set in the Stock Dictionary.

Instrument Group Properties:	Description:
<code>groupLongInstruments[]</code>	Number of long instruments in the group
<code>groupShortInstruments[]</code>	Number of short instruments in the group
<code>groupLongQuantity[]</code>	Quantity long for instruments in the group
<code>groupShortQuantity[]</code>	Quantity short for instruments in the group
<code>groupLongUnits[]</code>	Number of units long for instruments in the group
<code>groupShortUnits[]</code>	Number of units short for instruments in the group
<code>groupLongRisk[]</code>	Total risk for long instruments in the group
<code>groupShortRisk[]</code>	Total risk for short instruments in the group
<code>groupLongMargin[]</code>	Total margin for long instruments in the group

Instrument Group Properties:	Description:
groupShortMargin []	Total margin for short instruments in the group
groupLongProfit []	Total profit for long instruments in the group
groupShortProfit []	Total profit for short instruments in the group

Links:
See Also:
Instruments

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 366

7.7 Historical Trade Properties

Closed trade information is available by accessing any of the historical trade information.

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

Historical Trade Properties:	Descriptions:
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

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 370

7.8 Loading Functions

These functions can be used in conjunction with an Instrument Block Permanent Variable Type once the file has been loaded into a user declared **BPV** Instrument type variable.

Instrument Loading Functions:	Descriptions:
LoadSymbol	<p>Loads an instrument's data into a BPV Instrument class variable. The instrument does not need to be in the current portfolio. When it isn't in the portfolio, this function's parameter will allow Trading Blox Builder to find it and the instrument data file from where it is stored on the disk.</p> <p>This function is the method for accessing disk that don't have Automatic Instrument Context by default. It is also for loading all or any of the instruments that have a specific Rank, or when the data is to added to an existing instrument's IPV variables.</p> <p>To access and load an instrument this function supports the use of a:</p> <ul style="list-style-type: none"> ○ <code>symbol</code> ○ <code>type:symbol</code> ○ <code>current instrument index value</code> <p>To load an instrument, it must be loaded into a BPV Instrument type variable. It can't be loaded directly into an instrument object. However, the data in the file loaded can be used as a source for adding information to an IPV property variable.</p> <p>Note: LoadSymbol is required to first load an instrument ahead of the execution of any of the following functions that can load by a specific value or need.</p>
LoadByLongRank	<p>Loads into an instrument variable the instrument located at the specified long rank position.</p>
LoadByShortRank	<p>Loads into an instrument variable the instrument located at the specified short rank position.</p>
LoadExternalData	<p>Loads into an instrument variable the data from an external file. The data loads into IPV Series variables as defined.</p>
LoadBPVFromFile	<p>Loads a BPV from data in the file. Date Records not in the file data retain their default values.</p>
LoadIPVFromFile	<p>Loads data into an IPV exactly as in the file. Date Records not in the file data retain their default values.</p>

LoadSymbol

Access to instrument's information in any of the script section is possible using this function. When this function is used to access an instrument, it will bring the specified instrument's data into context so that its information is made available using a BPV variable name as an temporary instrument.

This function is most often used in script sections that do not have automatic instrument context, but it can be used in script section that have automatic context when an additional instrument's information is needed. Review the information in this table [Blox Script Timing](#) to see which script sections have automatic context, and which scripts require this function to access an instrument's data.

Instrument access can be to get data from another instrument in the portfolio, or to load a specified instrument that is not in the portfolio. See the "symbolSpecifier" parameter information.

can be set using the symbol, the **TYPE**:symbol, or the portfolio's index for the target instrument. Instrument can also be in another system that is listed in the suite being tested.

You can access each instrument in each system to obtain its unique properties. An instrument has many properties which are not unique across systems, like price and volume, but the trade properties and others are unique across systems and can be accessed by adding the [system.index](#) value of the system where the instrument is located (see example below).

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:
<code><BPVName>.LoadSymbol(symbolSpecifier [or symbol] [or index] [, system index])</code>

Parameter:	Description:
symbolSpecifier	<p>Symbol for the instrument with an optional market type prefix like: "F:GC" or "S:IBM"</p> <p>Valid Prefixes:</p> <ul style="list-style-type: none"> 'F:' - Futures 'S:' - Stocks 'FX:' - Forex <p>Note: Each prefix uses the file location paths established in the Preference's Data Folders and Data Options setting for each of the data types.</p> <p>Symbol information can use the path information as part of the symbol name.</p>

Parameter:	Description:
	<p>i.e. File Drive and Folder Name: "C:\Data\Stocks\ Symbol-name file: "S:@GU6-I" OR CSI# named file: "S:S0052244" symbol = "C:\Data\Stocks\" + "S:@GU6-I"</p>
[or symbol]	Symbol of the instrument like "S" or "AUDCAD"
[or index]	Index in the current portfolio.
[, system index]	Optional system index for the instrument to load.

Example:

' Create a Block Permanent Instrument Object Variable called tempInstrument.

Block Permanent Variable [Block: Instrument Data Access | Group: _Help]

Script Name ←

Display Name

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

Instrument - used to load and access alternate markets

Variable Options

Default Value

Scope

Reset Before Test

Name can be Mkt, Corn, Bonds, LoadedSymbol or many other ideas.

Name replaces the prefix **instrument** object prefix for symbol access .

i.e. "Mkt.close"

OK
Cancel

BPV Instrument Variable Type

Example:

```

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 ' index

' Create a Block Permanent Variable (BPV) 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 it's instrument.priorityIndex value. When it is greater than 0,
instrument
' is in the portfolio.

' Access System in a different system in the suite
myInstrument.LoadSymbol( symbol, systemIndex )

```

Returns:

The `tempInstrument` variable returns = **TRUE** (1) when instrument load is successful; it returns a **FALSE** (0) when it fails to load an instrument.

Links:

[Accessing Instruments](#), [Instrument Loading](#), [System Properties](#)

See Also:

Links:[Instrument, system.index](#)

LoadByLongRank

Sets an instrument variable to a particular instrument.

Syntax:

```
LoadByLongRank( rank )
```

Parameter:

Description:

Parameter	Description
rank	The long rank to load.

Example:

```
' Create a Block Permanent Instrument Object Variable called
tempInstrument.
' See LoadSymbol example.

' Loop printing the symbols in order of the long rank.
FOR index = 1 TO system.totalInstruments STEP 1

    ' Set the portfolio instrument.
    tempInstrument.LoadByLongRank( index )

    ' Print out the file name.
    PRINT "Long Ranking: ", index, tempInstrument.symbol
NEXT
```

Returns:

Returns a **TRUE** value when successful.

Links:

See Also:

[Instrument Loading](#)

LoadByShortRank

Sets an instrument variable to a particular instrument.

Syntax:

```
LoadByShortRank( rank )
```

Parameter:	Description:
rank	The short rank to load.

Example:

```
' Create a Block Permanent Instrument Object Variable called
tempInstrument.
' See LoadSymbol example.

' Loop printing the symbols in order of the short rank.
FOR index = 1 TO system.totalInstruments STEP 1

    ' Set the portfolio instrument.
    tempInstrument.LoadByShortRank( index )

    ' Print out the file name.
    PRINT "Short Ranking: ", index, tempInstrument.symbol

NEXT
```

Returns:

Returns a **TRUE** value when successful.

Links:

See Also:

[Instrument Loading](#)

LoadExternalData

Loads data from external text files and attaches it to a specific instruments.

This Function has been depreciated because the [LoadIPVFromFile](#) & [LoadBPVFromFile](#) functions were added to Trading Blox Builder. Those functions work better than this function and would be a better choice for new projects.

Note:

Option-1 is functional, but **Option-2** has been removed.

Syntax:
<code>instrument.LoadExternalData(externalFileName, "date", "beta", "eps")</code>

Parameter"	Description:
<code>externalFileName</code>	File name of the file you want to load.
<code>date-optional</code>	When this is used, it will add information using the dates.
Optional-Parameters	"date", "beta", "eps" are the additional column names. These are optional parameters that can be left out. When the optional column names for the new property data are used, the Auto-Index IPV property names must match the names of the columns used in calling the function. IF NOT <code>instrument.LoadExternalData(externalFileName, "date", "beta", "eps")</code> THEN

Note:

When 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 the [LoadExternalData](#) function call.

See also [LoadIPVFromFile](#) function.

Option 1:

Declare columns to be added as Instrument Permanent Variables (IPV) Auto Index Series so they can then be accessed using the normal `instrument.property-name` usage and plotted.

File Format:

The [LoadExternalData](#) call requires comma delimited text files with the first column being a date in the format YYYYMMDD.

This code loads external data files which use the symbol in the name and adds two new instrument properties:

```
beta  
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
```

A header is required, but ignored. To load a data file with this name, "CL_ExternalData.csv", using the above [LoadExternalData](#) the script 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
```

If this function is placed in the Before Test script, it will load (refresh the data by reloading it again) before each parameter test run.

Example:

Note:

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.

```
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"
```

Example:

```
' 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 ' index value
```

Returns:

Returns a **TRUE** value when successful.

Links:

[LoadPVFromFile](#)

See Also:

[Instrument Loading](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 405

LoadBPVFromFile

LoadBPVFromFile loads a two column comma delimited file where the first column is a date and the second column contains a numeric value. Most often the user will create a **BPV Auto-Index enable Numeric Series**. However, this function will support a **BPV Auto-Index String Series**.

Think of the process this way:

1. All **BPV** series element values are initialized at the time of creation with the default value the user allows in the series creation.
2. The BPV will contain the values located in the second column for any **test.currentDate** locations that match a date record from the loaded file.
3. The series' default value at a **test.currentDate** location will be found when a matching date wasn't found in the loading file.

Syntax:

```
LoadBPVFromFile(fullPathFileString, numericSeriesBPVname)
```

Parameter:	Description:
fullPathFileString	<p>The drive's letter and full path to the folder that contains the named file.</p> <p>The file information needed to load a file must be in the String that contains the <code>fullPathFileString</code> parameter text. That parameter needs the text information to find and then load the two-column file. In that string variable, the full-Path-name plus the File-name must match an actual directory's path location where the file name including its suffix is located (see the below example's "Setup the path for the file to Load into a BPV" section for how it might be obtained.)</p>
numericSeriesBPVname	<p>BPV Auto-Index enabled numeric series.</p> <p>During the file loading process, all <code>test.currentDate</code> locations that match a loaded file record date will contain the loaded file's second column value at the <code>test.currentDate</code> date location that matched a record in the loaded file. When the loaded file does not have a date record for a <code>test.currentDate</code> value, the initialize BPV Auto-Index enable Numeric Series value won't be changed from its previous initialized value.</p>

Example:

```
=====
' LoadBPVFromFile is an Import File Example available in the Blox
MarketPlace
' https://tinyurl.com/y7slhqrl
'
' In the example space below, a second BPV manually-indexed numeric
series was created so the
' value in the second column would have a series lookup location to
display the holiday date name
' that appears when the scripting display the second column value for the
matching dates.
```

Example:

BLOCK: Load BPVFile into a Numeric Series

Block Permanent **Variables**

```
tBPV_SeriesName [ BPV Variable Series Name ]; Series String; 0.000000;
Block
aHoliday [ Holiday as a YYYYMMDD Float ]; Series; 0.000000; Block
tPathName [ Any File Path Location String ]; String; ; Block
tFileName [ Any file name String ]; String; ; Block
tFullPathFileName [ BPV Load String Variable Name ]; String; ; Block
tHolidayNames [ Human readable name goes here... ]; Series String;
0.000000; Block
iFileFound [ True File Found ]; Integer; 0; Block
```

Parameters

```
VarToShowBlox [ Param to Displays System Blox ]; String; Block
```

SCRIPT: Before **Test**

```
'
=====
' LoadBPVFromFile-Example
' BEFORE TEST - START
'
' =====
' ~~~~~
' Setup the path for the file to Load into a BPV
' -----
' Assign a file path location (example uses
' Example uses TB's default installation path.
' Example information file is placed in the default
' path's data folder.
tPathName = fileManager.DefaultFolder + "Data\"

' Give the file name the text name of the file.
tFileName = "US-Holiday-Series_20201225.csv"

' Append the Path, the "\" and the Filename
' to create a full File & Path name text value
tFullPathFileName = tPathName + tFileName

' Validate File's Name and Location.
iFileFound = FileExists(tFullPathFileName)
' -----
' Test the presence of the file. If it is found, it will load
' the file into the BPV variable 'holidays'
If iFileFound AND
    test.LoadBPVFromFile(tFullPathFileName, "aHoliday" ) THEN
ELSE
    ' When the conditional test fails, this
    ' error message will appear.
    MessageBox("Unable to Find or Load data!")
ENDIF ' test.LoadBPVFromFile...
```

Example:

```

' ~~~~~
' Holiday Name Lookup List
' Holiday File Example uses the Holiday sequence number in the
' the series name braces to select the holiday name to display
' -----
tHolidayNames[ 1] = "New Year Day"
tHolidayNames[ 2] = "Martin Luther King Jr. Day"
tHolidayNames[ 3] = "Presidents Day (Washingtons Birthday)"
tHolidayNames[ 4] = "Memorial Day"
tHolidayNames[ 5] = "Independence Day"
tHolidayNames[ 6] = "Labor Day"
tHolidayNames[ 7] = "Columbus Day"
tHolidayNames[ 8] = "Veterans Day"
tHolidayNames[ 9] = "Thanksgiving Day"
tHolidayNames[10] = "Christmas Day"
' ~~~~~
' =====
' BEFORE TEST - END
' LoadBPVFromFile-Example
' =====
-----
SCRIPT: Before Trading Day
-----
' =====
' LoadBPVFromFile-Example
' BEFORE TRADING DAY - START
' =====
' ~~~~~
' Check Current Test Date to the Loaded Holiday dates.
' When a matching Holiday date is found, the name of
' the US Holiday will appear
' -----
' Each new date it will show a holiday record
' That aligns with a holiday date
If aHoliday != 0 THEN
  PRINT test.currentDate, _
    " is Holiday#: ", _
    AsInteger(aHoliday), _
    tHolidayNames[aHoliday]
ENDIF
' ~~~~~
' =====
' BEFORE TRADING DAY - END
' LoadBPVFromFile-Example
' =====

```

Returns:

Example above returns the annual sequence count of each of the ten US-Federal Holidays show next:

...[SNIP]

```

20141111 is Holiday#: 8 Veterans Day
20141127 is Holiday#: 9 Thanksgiving Day
20141225 is Holiday#: 10 Christmas Day
20150101 is Holiday#: 1 New Year Day

```

Example:

```
20150119 is Holiday#: 2 Martin Luther King Jr. Day
20150216 is Holiday#: 3 Presidents Day (Washingtons Birthday)
20150525 is Holiday#: 4 Memorial Day
20150703 is Holiday#: 5 Independence Day
20150907 is Holiday#: 6 Labor Day
20151012 is Holiday#: 7 Columbus Day
20151111 is Holiday#: 8 Veterans Day
20151126 is Holiday#: 9 Thanksgiving Day
20151225 is Holiday#: 10 Christmas Day
20160101 is Holiday#: 1 New Year Day
20160118 is Holiday#: 2 Martin Luther King Jr. Day
20160215 is Holiday#: 3 Presidents Day (Washingtons Birthday)
20160530 is Holiday#: 4 Memorial Day
20160704 is Holiday#: 5 Independence Day
20160905 is Holiday#: 6 Labor Day
20161010 is Holiday#: 7 Columbus Day
20161111 is Holiday#: 8 Veterans Day
20161124 is Holiday#: 9 Thanksgiving Day
[SNIP]...
```

Links:[FileExists](#)**See Also:**[General](#), [File & Disk](#), [Test Object Functions](#)

LoadIPVFromFile

This instrument function loads data from an external comma separated text file into one or more user created instrument property names. The data column names that match the user's previously added IPV properties names must match for the transfer to succeed.

The IPV property types must be an **Auto-Index Numeric** or a **String** series. When the **IPV** properties are **Auto-Index Numeric** series Trading Blox Builder will match the dates in the imported text file to align the property values to those same dates in the instrument where they are being added.

Each date available in the text file is less than the instrument's record dates, the instrument record dates that didn't have a matching date in the loaded text file, will leave those instrument records with the default value that Trading Blox Builder assigns when it initializes the user created **Auto-Index Numeric** or a **String** series name.

The dates where the default value in the instrument file, the default value will be used when scripting accesses a instrument record date that wasn't updated with data from the imported text file data. A solution to that will provide the last know update to a matching date record is available in an example [here](#):

This function most often is used in the [Before Simulation](#) or [Before Test](#) script. To make this transfer work, it requires a [BPV Instrument Variable](#) to act as an instrument container. This container is needed because the instrument will need to be loaded with an instrument using the [LoadSymbol](#) function. This loading function is needed when the script sections where an instrument needs to be loaded does not support automatic [Instrument Context](#).

Once the instrument is loaded, this [LoadIPVFromFile](#) function can then load the Instrument Data file information and update the instrument properties that match the data column names.

The [LoadIPVFromFile](#) function can work in script sections where the [Instrument Context](#) is automatically provided.

Syntax:

```
' No "Date" parameter is used with this function.
' isLoaded = Returns TRUE when the function's operation
' is successful, and FALSE when it fails.
isLoaded = LoadIPVFromFile( filePathName, columnsToSkip[,myIPVSeries1]
[,myIPVSeries2][,etc.] )
```

Parameter:	Description:
filePathName	The name of the file to open. If no path name is given, it defaults to the location of the instrument data file.
columnsToSkip	Add the number of comma separated columns in the file that has the values to be added to the new IPV series properties created to store addition property data. If data is available right after the date column, enter a Zero to specify no columns should be skipped.

Parameter:	Description:
[,myIPVSeries1]	Adding More Columns is Optional: The name for the first column of data after the date and optional time.
[,myIPVSeries2]	Adding More Columns is Optional: The name for the second column of data after the date and optional time.
[,etc.]	Adding More Columns is Optional: The name of an additional column of data.

LoadIPVFromFile Information:

This function will load **date** and **time** data into an **IPV** as well. To update the date and time, the file format would then be "Date, Time, column1, column2, etc.". This different format will work correctly only when the instrument data is an intraday data series. All date time combinations in the file must also be present in the instrument data file.

For this function example, the process requires the added text files and a **IPV Numeric Series Auto-Index Properties** to be created before this function can succeed. The column heading names in the text file and in the added IPV Property names must be the same. This example's uses a **beta** and **eps** for the column property names.

This image shows how one of the **IPV Numeric Series Auto-Index Property** used the **eps** name. The other property created used the **beta** name for the text column and for the added **IPV Property**:

Instrument Permanent Variable [Block: Help-LoadIPVFromFile | Group: ...]

Script Name:

Display Name:

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

Variable Options:

Default Value:

Scope:

Reset Before Test

Auto Index

Enabled

IPV LoadIPVFromFile IPV Requirements

Block Permanent Variables

- Mkt BPV Instrument Type container
- Instrument Permanent Variables
 - Beta Added Numeric Auto-Index
 - Eps System Scoped Series
- Parameters
- Indicators

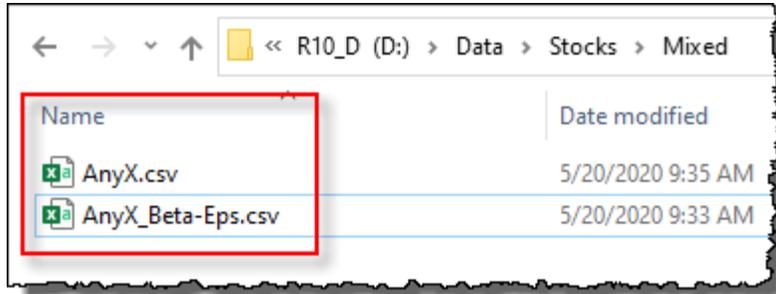
IPV LoadIPVFromFile IPV Requirements

This function example used the Before Simulation script section. That script section will only load the added data information once during the entire simulation. Before using the **LoadIPVFromFile** function in the Before Simulation script section, set the new **IPV Numeric Auto-Index Series Property** variables to **System** scope. This scope setting will avoid overwriting any new property data with the **IPV's** default Block scope setting.

The example text file below will only provide new information every new quarter in the year. This means, dates that are not in the text file columns will be set to the default value for the series. When creating a new numeric IPV property, set the default value to **"-1.000000"** to indicate the property update is **False**.

If it needs to be used with a Multi-Step simulation, the Before Test script section can be used so the added data will be refreshed before each parameter test-step execution.

To simplify the use of the example scripts below, we placed the instrument and sample data into the same disk folder location. The instrument symbol is also the first part of the added data sample. The instrument's symbol should be the lead characters in both of the files. In this example, both the symbol and added data information are fictitious symbol files created to provide a more complete example process.



IPV LoadIPVFromFile IPV Data Location Example

Once the function loads the external property data into the symbol's added numeric series, the names in the sample data that match the names of the added **IPV** properties will be updated. In this case, the names of the added properties and the column names must be the same: **beta & eps**.

To access these new IPV properties, they can be accessed in other locations:

```
IF instrument.beta > 1.2 THEN
```

OR

```
IF instrument.eps > instrument.eps[90] THEN
```

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:

```
value = instrument.beta      ' returns 1.112
value = instrument.beta[1]   ' returns default value
value = instrument.beta[2]   ' returns default value
```

Sample Property Data Example:

```
' Data is Random information created so the
' transfer process can use the following
' scripts to move the data from the sample
' into the symbol's Property data.
' -----
```

Sample Property Data Example:

```

Date,      beta,      eps
20050118,  1.201,  5.8
20050415,  1.345,  6.2
20050715,  1.112,  5.3
20051017,  1.535,  6.9
20060117,  1.231,  8.4
20060417,  1.159,  8.3
20060717,  1.188,  8.8
20061016,  1.208,  7.7
20070116,  1.434,  4.6
20070416,  1.459,  5.6
20071015,  1.484,  6.1
20080115,  1.509,  8.3

```

```

' -----
' The dates of the Sample data must match a
' date in the instrument data series for the
' data updating from the external data file
' to be successful.
'
' Added data files must be comma separated values.
' All dates in the added file must also be in the
' instrument file in the YYYYMMDD format. When a
' date in the sample file is not found in the
' instrument file, the added data will not be
' added to the instrument information.

```

Example - Step 1:**Notes:**

- The "Mkt" variable is a **BPV Instrument Type Variable** that is the container that holds the loaded instrument. The Sample data in a separate file will move its column data into the instrument's same name properties.
- [LoadSymbol](#) function loads the text file named column information into instrument symbol information using the `Mkt.LoadIPVFromFile` function.
- In this example, the default location for the Sample data and its symbol-matching instrument are both in the same folder location. This means the script path and file names will be in the same folder in the disk.
- The same full path is used to simplify the access and loading of an instrument and its new property.
- Place the scripted example statements below into the **Before Simulation** script section.
- Edit the disk and folder information so that it matches the names and path information you are going to use. In this case, there are 12 new IPV Property values to add to an instrument, and the instrument output results show there are 12 update symbol data records in the Results section of this table.

```

' =====
' LoadIPVFromFile - Help File Example - Step 1
' BEFORE SIMULATION - START
' =====

```

Example - Step 1:

```

' ~~~~~
VARIABLES: instrumentCount, _
              iSymbolExist, _
              iDataFileExist, _
              index, _
              iLoadOK, _
              instrumentCount Type: Integer

VARIABLES: DataFullPathName, _
              SymbolFullPath, _
              DataPathName, _
              SymbolName, _
              DataFileName Type: String

' ~~~~~
' Enter the Full Disk & Folder Path Sequence destination
DataPathName = "D:\Data\Stocks\Mixed"
' Enter the Symbol Series Data File to load
SymbolName = "AnyX.CSV"
' Enter the IPV Series Data File to load
DataFileName = "AnyX_Beta-Eps.csv"
' Assemble Path & File Strings for Access
SymbolFullPath = DataPathName + "\" + SymbolName
' Is Symbol File Found?
iSymbolExist = FileExists( SymbolFullPath )

' -----
' Create a Full Path & File Name String
DataFullPathName = DataPathName + "\" + DataFileName
' Is Data File Found?
iDataFileExist = FileExists( SymbolFullPath )

' -----
PRINT "iSymbolExist", iSymbolExist
PRINT "iDataFileExist", iDataFileExist

' Get the instrument count.
instrumentCount = system.totalInstruments
' Print out the file name.
PRINT "Loading External File: ", iDataFileExist

' ~~~~~
' Loop initializing each instrument.
For index = 1 TO instrumentCount STEP +1
  ' Set the portfolio instrument.
  ' "Mkt" is defined as a BPV Instrument variable.
  iLoadOK = Mkt.LoadSymbol( index )

  ' Load the external data.
  If iLoadOK = TRUE THEN
    PRINT index, Mkt.fileName, Mkt.symbol, DataFileName
    ' Load IPV Data File
    Mkt.LoadIPVFromFile(DataFullPathName,0,"beta","eps")
  ELSE
    ' Report External Instrument Load Failure

```

Example - Step 1:

```

    PRINT "Could not Load Property Data: ", DataFileName
  ENDIF
Next ' index value
' ~~~~~
' =====
' BEFORE SIMULATION - END
' LoadIPVFromFile - Help File Example
' =====

```

Step 1 Returns:

```

iSymbolExist 1
iDataFileExist 1
Loading External File: 1
1 ANYX.CSV ANYX AnyX_Beta-Eps.csv

```

Example - Step 2:**Notes:**

- This scripting is placed into the **BEFORE INSTRUMENT DAY** script section.
- Its only purpose is to verify and audit the transfer from the new property data file to the instrument's numeric series properties of the same name.
- It also provides a day-of-week name for the data date aligned with the new property data.
- Output only shows the dates where the data from the new property values file to the instrument data records is a match.
- If any data didn't get transferred, it is likely there is a date that isn't a market day in one of the files. Fix the missing data problem by validating the instrument date and the added property data dates.

Example - Step 2:

```

' =====
' LoadIPVFromFile - Help File Example
' BEFORE INSTRUMENT DAY - START
' =====
' ~~~~~
' Local-Variables Definitions
VARIABLES: iNdx, iHeader Type: Integer
' -----
' If the Output Column Header has not been displayed,...
If iHeader = FALSE THEN
'   Generate a Header to the columns
PRINT "Symbol,Date, Close, Beta, EPS, DayName"
'   Update iHeader State
iHeader = TRUE
ENDIF ' iHeader = FALSE
' -----
' When an Updated Property is found,...
If instrument.beta <> 0 OR instrument.eps <> 0 THEN
'   Report the update records updated
PRINT instrument.symbol, _
        instrument.date, _
        instrument.close, _
        instrument.beta, _
        instrument.eps, _
        DayOfWeekName( DayOfWeek(instrument.date) )
ENDIF ' i.beta != 0 OR i.eps != 0
' ~~~~~
' =====
' LoadIPVFromFile - Help File Example
' BEFORE INSTRUMENT DAY - END
' =====

```

Step 2 Returns:

```

Symbol,Date, Close, Beta, EPS, DayName
ANYX 20050118 39.951000000 1.201000000 5.800000000 Tuesday
ANYX 20050415 39.734333330 1.345000000 6.200000000 Friday
ANYX 20050715 40.971000000 1.112000000 5.300000000 Friday
ANYX 20051017 44.011000000 1.535000000 6.900000000 Monday
ANYX 20060117 50.321000000 1.231000000 8.400000000 Tuesday
ANYX 20060417 54.421000000 1.159000000 8.300000000 Monday
ANYX 20060717 50.721000000 1.188000000 8.800000000 Monday
ANYX 20061016 58.231000000 1.208000000 7.700000000 Monday
ANYX 20070116 63.464000000 1.434000000 4.600000000 Tuesday
ANYX 20070416 69.944000000 1.459000000 5.600000000 Monday
ANYX 20071015 74.714000000 1.484000000 6.100000000 Monday
ANYX 20080115 66.334000000 1.509000000 8.300000000 Tuesday

```

Missing Property Data:

In the past, Trading Blox Builder provided the option to get the last known value when there hadn't been an update to the property for all instrument dates. This isn't an option any longer, but it can be obtained by a simple conditional statement that executes in the [UPDATE INDICATORS](#) Script Section.

```

' =====
' LoadIPVFromFile - Help File Example
' UPDATE INDICATORS - START
' =====
' ~~~~~
' When the IPV auto indexed series loaded from a text file that
' didn't update each instrument record, and the need is to know
' most recent property value, this script will carry that value
' in the new user created IPV static values.
'
' NOTE:
'   -1 Is recommended Default value To use when LoadIPVFromFile
'   Data might skip LoadIPVFromFile Data might leave the default
'   instrument record value, use this process to carry the
'   value forward.
' -----
'   If the value Is Not -1,...
' If instrument.eps <> -1 Or instrument.beta <> -1 Then
'   Update static IPV with an Updated value
'   currentEPS = instrument.eps
'   currentBETA = instrument.beta
' ENDIF
' ~~~~~
' =====
' LoadIPVFromFile - Help File Example
' BEFORE INSTRUMENT DAY - END
' =====

```

Links:

[LoadExternalData](#), [LoadBPVFromFile](#)

See Also:

[Block Permanent Variables](#), [Instrument Permanent Variables](#)

7.9 Position Functions

These instrument Object functions are used to assign or change the value of position property values.

Positions Functions:	Descriptions:
IBSynchPositions	
NetIBPosition	
NetTBPosition	
order.SetRuleLabel()	This function is part of the Order object. When an order is active and in context, it will set the text description created so it will be available during the order process, and if the order is filled, be available in the Trade Log.
SetExitLimit()	Sets the exit limit price for the specified unit
SetExitStop()	Sets the exit stop price for the specified unit
SetUnitCustomValue()	Sets the custom value property of a current position to a specified unit's custom value
SetUnitPositionMessage()	Sets a text description to a specified unit in a current position.

Links:

[Position Properties](#), [Order Functions](#)

See Also:

SetExitLimit

Sets the Exit-Price Limit for an active position's specified unit.

This function works where order are being processed to assign a value:

Order object script sections:

[Unit Size](#)

[Can Add Unit](#)

[Can Fill Order](#)

Instrument object scripts:

[Entry Order Filled](#)

[Exit](#)

[Entry](#)

[Adjust Stops](#)

After Instrument Day

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 ).
```

Syntax:

```
SetExitLimit( [unitNumber,] limitPrice )
```

Parameter:	Description:
[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 )
```

Returns:

Values assigned can be printed and used after assignment. It is also available to the Broker object to execute.

Links:
unitExitLimit , Unit Size , Can Add Unit , Can Fill Order , Entry Order Filled , Exit , Entry , Adjust Stops , After Instrument Day
See Also:
Position Functions , Position Properties

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 542

SetExitStop

Sets the exit stop for an active position specified unit.

This function works where order are being processed to assign a value:

Order object script sections:

[Unit Size](#)
[Can Add Unit](#)
[Can Fill Order](#)

Instrument object scripts:

[Entry Order Filled](#)
[Exit](#)
[Entry](#)
[Adjust Stops](#)
After Instrument Day

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 ).
```

Syntax:
<code>SetExitStop([unitNumber,] stopPrice)</code>

Parameter:	Description:
<code>[unitNumber,]</code>	The unit number (optional). If not supplied this will default to the first unit.
<code>stopPrice</code>	The value of the stop to be set.

Example:

```
' 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 )
```

Returns:

Values assigned can be printed and used after assignment. It is also available to the Broker object to execute.

Links:

[unitExitStop](#)

See Also:

[Position Functions](#), [Position Properties](#)

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:	
<code>SetUnitCustomValue([unitNumber,] value)</code>	

Parameter:	Description:
<code>[unitNumber,]</code>	The unit number (optional). If not supplied this will default to the first unit.
<code>value</code>	The custom value to be set. This value can be retrieved using the <code>unitCustomValue</code> property.

Example:
<pre>' Set the custom value. instrument.SetUnitCustomValue(1.5) PRINT "The custom value is ", instrument.unitCustomValue</pre>

Returns:
When value is printed, it will show the value entered. In this example, 1.5 will appear. Value will also appear in the Trade Log.

Links:
See Also:
Position Functions

SetUnitPositionMessage

When an added unit is created, the `instrument.SetUnitPositionMessage` function will add a message to the unit number listed in the parameter area of the function. The text of the message in the text-message area of the function will be the text assigned to the unit created.

Syntax:

```
instrument.SetUnitPositionMessage( [unitNumber], unitMessageText )
```

Parameter:**Description:**

```
[unitNumber  
]
```

Integer value that identifies the unit number. See: [Unit Indexing](#)

```
unitMessage  
Text
```

Text information that identifies the reason or method that helps to understand why the order appeared.

Example:

```
' The ENTRY ORDER FILLED script section is
' where a position's order becomes a unit.
' It is good place to add a specific message
' to a position's specific unit number.
'
' The first unit in a position filled will
' return a UnitCount = 1. Each unit added
' unit to that same position will increment
' the UnitCount by adding 1 to the previous
' UnitCount in a position.
'
' To add a message to a new unit this script
' property statement will be provide the unit
' number of the newest unit order filled:
UnitCount = instrument.currentPositionUnits
'
' Add a specific message to a position's
' newest unit use this statement:
instrument.SetUnitPositionMessage( UnitCount, "EntryRisk" )
```

Returns:**Links:**

[Position Properties, Unit Indexing](#)

See Also:

[Trade Control Functions, Trade Control Properties](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 29

7.10 Position Properties

Active trade information is made available by accessing these Instrument Position Properties.

Unit Indexing:

Properties listed with a '[]' following them may be indexed using a number. Index number is to identify which unit in the position a property is to reference.

```
' This statement will return the position quantity, in
' contracts or shares, for the FIRST unit in the position
firstUnitQuantity = instrument.unitQuantity[ 1 ]
```

Position can have more than one unit. To discover how many units are in a position use this property:

```
' Count Position Units
UnitCount = instrument.currentPositionUnits
```

When a position has more than one unit change the value index value to reference that unit.

```
' This statement will return the position quantity, in
' contracts or shares, for the SECOND unit in the position
firstUnitQuantity = instrument.unitQuantity[ 2 ]
```

When no index is supplied the property will return the information for the first unit "[1]".

```
' This statement will return the position quantity, in
' contracts or shares, for the FIRST and possibly the
' ONLY unit in the position
firstUnitQuantity = instrument.unitQuantity
```

Syntax:

```
value = instrument.<position-property-name> [ index ]
```

Position Properties:	Descriptions:
<code>barsSinceExit</code>	Number of bars since the last exit, regardless of unit
<code>currentPositionGrossProfit</code>	Gross profit is the basic profit computation, while net (<code>currentPositionProfit</code>) includes roll commission, entry commission, custom commission, roll slippage carry cost, dividends, and saved walk forward profit.
<code>currentPositionProfit</code>	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.
<code>currentPositionQuantity</code>	Total number of contracts or shares on for the current position
<code>currentPositionRisk</code>	Total risk for this position, in dollars, based on the close and the stop prices
<code>currentPositionUnits</code>	Total number of units on for the current position

Position Properties:	Descriptions:								
<code>ts</code>									
<code>position</code>	Current position, represented by the numerical constants <code>SHORT</code> , <code>OUT</code> , or <code>LONG</code> .								
<code>positionDescription</code>	Current position as a string equal to <code>SHORT</code> , <code>OUT</code> , or <code>LONG</code> . Useful for printing.								
<code>purchaseEquity</code>	For use with active stock positions. A stock's Initial equity position amount required to purchase a current positions (entry fill price times the instrument's current share quantity) is returned..								
<code>system</code>	System index for the this instrument's position.								
<code>systemMarketOrderNetQuantity</code>	Total net quantity (long minus short) for that instrument/system or market on open orders								
<code>unitBarsSinceEntry []</code>	The number of bars since the entry of this unit								
<code>unitCarry []</code>	The total carry cost of the unit for forex trades.								
<code>unitCommission []</code>	The computed commission that will be applied to the trade when the position exits.								
<code>unitCustomCommission []</code>	The accrued custom commission added to the instrument through the use of the <code>AddCommission</code> function.								
<code>unitCustomValue []</code>	The custom value as set through scripting into the unit, or the order that created the unit. Float value.								
<code>unitDeliveryMonth []</code>	The delivery month (<code>YYYYMM</code>) of the unit								
<code>unitEntryDate []</code>	The entry date of the unit								
<code>unitEntryDayIndex []</code>	This property return is the bar index count since entry.								
<code>unitEntryFill []</code>	The fill price for the unit								
<code>unitEntryFillTime</code>	<table border="1"> <thead> <tr> <th>Fill Time:</th> <th>Fill Type:</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Open</td> </tr> <tr> <td>1</td> <td>Day</td> </tr> <tr> <td>2</td> <td>Close</td> </tr> </tbody> </table> <p>Function returns an integer indicating the type of fill that happened.</p>	Fill Time:	Fill Type:	0	Open	1	Day	2	Close
Fill Time:	Fill Type:								
0	Open								
1	Day								
2	Close								
<code>unitEntryOrder []</code>	The order price for the unit								
<code>unitEntryRisk []</code>	The entry risk for the unit adjusted by the fill price								
<code>unitEntryTime []</code>	The entry time of the unit								

Position Properties:	Descriptions:
<code>unitExitLimit[]</code>	The exit limit for the unit as set by SetExitLimit
<code>unitExitStop[]</code>	The exit stop for the unit as set by the original broker function call, or set by SetExitStop
<code>unitMaxAdverseExcursion[]</code>	The maximum unfavorable excursion of the unit
<code>unitMaxFavorableExcursion[]</code>	The maximum favorable excursion of the unit
<code>unitMinFavorableExcursion[]</code>	The minimum favorable excursion of the unit
<code>unitNoExitStop[]</code>	Returns true if there is no exit stop set for the unit. Useful for cleaner reporting.
<code>unitPositionReferenceID[]</code>	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
<code>unitProfit[]</code>	The open profit of the trade, net of commissions etc, in instrument currency.
<code>unitQuantity[]</code>	The quantity for the unit
<code>unitRollCommission[]</code>	The accrued commission of the futures trade from all rolls, in base currency.
<code>unitRollPrice[]</code>	The roll price is the last roll price, which is the price at which the open equity was moved to closed. So from this price you can compute new unaccrued profit
<code>unitRollProfit[]</code>	The accrued closed profit of all the Futures rolls in the trade is returned in the simulation's base currency
<code>unitRollSlippage[]</code>	The accrued slippage of the futures trade from all rolls, in base currency.
<code>unitRuleLabel[]</code>	The rule label as set through scripting into the unit, or into the order that created the unit. String value.
<code>unitSavedWFFProfit[]</code>	
<code>unitStockDividends</code>	
<code>usedMargin</code>	Each symbol's in use margin amount in a stock or a futures system.

Links:

[Position Functions](#)

See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 468

UnitRollCalculations

the internal properties that you need to audit the current position profit.

Syntax:

```
instrument.unitRollSlippage
instrument.unitRollCommission
instrument.unitRollProfit
instrument.unitRollPrice
```

Parameter:	Description:

Example:

```
' Futures Contract Unit Roll Calculations
' Start with the accrued profit.
' -----
' The accrued closed profit of all the Futures
' rolls in the trade is returned in the
' simulation's base currency
' computedProfit = instrument.unitRollProfit
' -----
If instrument.position = LONG THEN
  computedProfit = computedProfit _
                  + ( instrument.close _
                    - instrument.unitRollPrice ) _
                  * instrument.conversionRate _
                  * instrument.unitQuantity _
                  * instrument.nativeBPV
ENDIF
' -----
If instrument.position = SHORT THEN
  computedProfit = computedProfit _
                  + ( instrument.unitRollPrice _
                    - instrument.close ) _
                  * instrument.conversionRate _
                  * instrument.unitQuantity _
                  * instrument.nativeBPV
ENDIF ' i.position = SHORT
' -----
```

Returns:

An example Blox is available here:

Returns:

[Example Roll Profit.tbx](#)

Links:**See Also:**

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 717

7.11 Ranking Functions

The ranking functions are usually used in the [Portfolio Manager](#) to set ranking values and to filter the instrument from the portfolio. They can also be used in other script section when a custom portfolio process is needed, or when a ranking property needs to be changed.

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.
SetCustomSortValue()	Set the value in the instrument.customSortValue .

An instrument's ranking values can also be set in others scripts, such as in a manual instrument loop in the **Before Test**, or **Before Trading Day** script sections. These two scripts don't provide [Automatic Instrument Context](#). To work with an instrument in script sections where instrument context is not there automatically, you can use the [LoadSymbol](#) function and examples to update instruments.

When scripting is providing the updates, the following

- Set the Long Ranking and/or Short Ranking values an each instrument until all instruments you want to update are processed.
- Next, execute the [system.RankInstruments](#) function.
- When the order of instruments need to be sorted, execute one of the [system.SortInstrumentList\(method# \)](#) funtions.
- 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:

[Instruments](#), [System](#)

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. However it can be used in any script as shown where an instrument has context, or it can be used in any script when used in conjunction with a BPV Instrument Type using the procedures shown in [Accessing System Portfolio Instruments](#)

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.

Syntax:

```
instrument.SetLongRankingValue( rankingValue )
```

Parameter:	Description:
rankingValue	Value used for ranking an instrument for long trades.

Example:

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

Returns:

The assigned RSI value assigned is available in the `instrument.longRankingValue`. It can be used as a comparison sort value to change sequential order of the instruments in the portfolio when that type of process is needed.

Links:

[Accessing System Portfolio Instruments, Ranking Functions, Ranking Properties](#)

See Also:

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. However it can be used in any script as shown where an instrument has context, or it can be used in any script when used in conjunction with a BPV Instrument Type using the procedures shown in [Accessing System Portfolio Instruments](#)

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.

Syntax:

```
instrument.SetShortRankingValue( rankingValue )
```

Parameter:**Description:**

rankingValue

The short trade value to be used for ranking this instrument.

Example:

```
' Set the short ranking value.
instrument.SetShortRankingValue( rsi )
```

Returns:

The assigned RSI value assigned is available in the `instrument.shortRankingValue` property. It can be used as a comparison sort value to change sequential order of the instruments in the portfolio when that type of process is needed.

Links:

[Accessing System Portfolio Instruments](#), [Ranking Functions](#), [Ranking Properties](#)

See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 558

SetCustomSortValue

This function is used to set the value in the `instrument.customSortValue` property.

When a value is given to each instrument the portfolio, those custom values can be sorted to change the order in which the instruments in the portfolio are sequenced.

Syntax:

```
instrument.SetCustomSortValue( value )
```

Parameter:**Description:**

value	Any positive or negative number.
-------	----------------------------------

Example:

```
' Assign RSI value 23
instrument.SetCustomSortValue( RSI )

' Sort Portfolio
system.SortInstrumentList ( 4 )
```

Returns:

RSI value of 23 will be available in instrument's where the RSI value of 23 was assigned.

Links:

[Accessing System Portfolio Instruments, Ranking Functions, Ranking Properties](#)

See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 540

7.12 Ranking Properties

The ranking properties can be accessed from anywhere in the system where an instrument is accessible. 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.

Property 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:**

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 485

7.13 Trade Control Functions

Trade control functions are designed to allow a portfolio process to control which instruments are allowed to create an order.

Usually used in a portfolio managing process 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
DisableTrading()	This function is executed in the SetParameters script section. When the parameter is set It scan remove instruments from the portfolio. so the entry and exit and update indicators scripts do not run. This can be useful when loading up 10,000 stocks in a portfolio but only really wanting to trade 700 of them.

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:	Description:
none	Function works without any user assigned values.

Example:

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

Returns:

Allows Long order positions to process when the instrument's long-rank value is \geq the rank-threshold value.

Links:

See Also:

[Trade Control Functions](#), [Trade Control Properties](#)

AllowShortTrades

Marks the instrument to allow short trades.

Syntax:

```
instrument.AllowShortTrades
```

Parameter:**Description:**

none

Function works without any user assigned values.

Example:

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

Returns:

Allows Short order positions to process when the instrument's short-rank value is \leq the rank-threshold value.

Links:**See Also:**

[Trade Control Functions](#), [Trade Control Properties](#)

AllowAllTrades

Marks the instrument to allow both long and short trades.

Syntax:
<code>instrument.AllowAllTrades</code>

Parameter:	Description:
none	Function works without any user assigned values.

Example:
<pre>' Allow all trades unless we filter below based on ' further criteria. instrument.AllowAllTrades</pre>

Returns:
This instrument setting allows position orders in a Long and Short direction to be processed.

Links:
See Also:
Trade Control Functions , Trade Control Properties

DenyLongTrades

Marks the instrument to deny long trades. This function is the opposite of [AllowLongTrades](#).

Syntax:
<code>instrument.DenyLongTrades</code>

Parameter:	Description:
none	Function works without any user assigned values.

Example:
<pre>' If this instrument is NOT in the top rankings... IF instrument.longRank > rankThreshold THEN instrument.DenyLongTrades ENDIF</pre>
Returns:
When Long-Order instrument's Long-Rank value > the rank-threshold value, the order will be disabled and removed.

Links:
See Also:
Trade Control Functions , Trade Control Properties

DenyShortTrades

Marks the instrument to deny short trades. This function is the opposite of [AllowShortTrades](#).

Syntax:
<code>instrument.DenyShortTrades</code>

Parameter:	Description:
none	Function works without any user assigned values.

Example:
<pre>' If this instrument is NOT in the top rankings... IF instrument.shortRank > rankThreshold THEN instrument.DenyShortTrades ENDIF</pre>
Returns:
When Short-Order instrument's Short-Rank value > the rank-threshold value, the order will be disabled and removed.

Links:
See Also:
Trade Control Functions , Trade Control Properties

DenyAllTrades

Marks the instrument to deny both long and short trades. This function is the opposite of [AllowAllTrades](#).

Syntax:

```
instrument.DenyAllTrades
```

Parameter:**Description:**

none

Function works without any user assigned values.

Example:

```
' Deny all trades unless we allow below based on further criteria.  
instrument.DenyAllTrades
```

Returns:

This function forces all orders to be disabled and removed.

Links:**See Also:**[Trade Control Functions](#), [Trade Control Properties](#)

7.14 Trade Control Properties

These control properties provide a **True** or **False** value of the most recent trade control function used.

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. When they show a trade is not allowed, they can be used as a condition that allows or ignores a order to create a new position unit.

Property Name:	Description:
<code>canTradeLong</code>	True if the instrument is allowed to trade long today.
<code>canTradeShort</code>	True if the instrument is allowed to trade short today.

Example:

```
' Get Long Trade State
IF instrument.canTradeLong THEN
  ' Enable a Long Order to be created.
ENDIF
```

Returns:

When property is True, orders are allowed to process.

Links:

See Also:

[Trade Control Functions](#)

Section 8 – Order

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:
Changing Orders	Information about when an order can be changed.
Creating Orders	Process of how orders are created.
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.

Automatic Order Context Access:

Order object properties and functions are available in the [Unit Size](#), [Can Add Unit](#), and [Can Fill Order](#) script section where they get 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 returns a `TRUE` result after a [Broker](#) function statement returns execution from [Unit Size](#) and [Can Add Unit](#) script sections to the [Entry Orders](#) or [Exit Orders](#) script section.

Note:

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. Order object must be in context by default or by using the [AlternateOrder](#) function. When it is in context active order property information and functions can be accessed or executed to make changes.

Do Not use the Order object in any of the script sections where it does not get automatic object context by default.

Accessing Order Functions and Properties:

Order-Type:	Description:
Entry	<p>Broker-functions that create new Entry orders call the <code>UNIT SIZE</code> script. <code>UNIT SIZE</code> scripts are where order quantity is determined and assigned. All orders that are intended to have an effect when it becomes a position must have a quantity.</p> <p>After an order is sized it can be processed through the <code>CAN ADD UNIT</code> script section when that script exists. <code>CAN ADD UNIT</code> script can apply filtering logic to determine if an order can be added to the system.</p> <p>Allowed order show their <code>order.continueProcessing</code> condition as <code>TRUE</code>. Rejected orders because of size is too small, insufficient margin, cash, number of orders or positions, etc. show the <code>order.continueProcessing</code> condition set to <code>FALSE</code>.</p> <p>Allowed orders are accessible in other script section once they are returned to Broker-Function that created the order. To know if an order is available test the <code>system.orderExists()</code> function to see if it returns a <code>TRUE</code> state.</p>
Exit	<p>Broker-functions create Exit-Order without having to process the order through any other script section.</p>

Order-Type:	Description:
AdjustPosition	<p>Orders to Add-Size or Reduce-Size are determined by the ratio value applied to these functions. Values greater than 1 will increase quantity size, and value less than 1 will reduce size quantity.</p> <p>All order that change order quantity size create the order directly and do not send the Broker transaction to any other script section.</p> <p>Orders that increase size create Entry-Orders. Orders that Reduce-Size create Exit-Orders. In both order type cases the order direction emulates the position direction that is being adjusted.</p>
Order Context	When a script section doesn't normally have order context context can be created by using SetAlternateOrder function.

[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:
<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 is properties and functions.</p> <p>Once orders are brought into context their properties and function are available to reference and changes.</p>

Links:
AlternateOrder Object , AlternateSystem Object , Order Functions , Order Object , Order Properties
See Also:
User's Guide Generating Orders

8.1 Order Context

Instrument Context is a process that provides automatic direct access to instrument information. Script sections where instrument context is automatic by default is shown in the [Blox Script Timing](#) table, and in the table [here](#). During the initialization of test, each instrument's data is automatically loaded into an instrument object. Instrument context makes access to the instrument data easy.

In script sections where instrument context information is not automatically provided, instrument data can still be accessed using the Instrument Object's [LoadSymbol](#) function. The [LoadSymbol](#) function uses the [BPV Instrument Type](#) as a container to access an instrument in a script section where instrument context is not automatically provided. Examples showing how to create the scripts statements is available in the [LoadSymbol](#) topic.

The [LoadSymbol](#) function can also be used to bring additional instruments into a script location where instrument context is automatic. This means the active instrument Trading Blox Builder made available can have a companion instrument brought into the script section at the same time. This ability to access more than one instrument provides information that might influence how the script logic considers. An example of how this can be used is with a spread value difference between two instruments, or a ranking comparison of group of instruments.

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 705

8.2 Changing Orders

Changing Order Information:

Functions in the Order Object are the only methods that can change filled orders. All order changes must happen before the order is removed.

Scripts, where the [Order Object Properties](#) and function are normally accessible, are shown in this table. Orders are accessible in other scripts, and in other systems in the same Suite. To access an order that is in the same system, but not in a script where it has context, use the [AlternateOrder Object](#).

To access orders that are not in the same system from where you wish access, use the [AlternateSystem Object](#) to bring that system into context before using the [AlternateOrder Object](#).

When accessing an order keep in mind that access to any order is different than making changes to the order that flow to the position that will end and only exist as a closed trade record.

Changes to an entry order must happen before the order is in the [Entry Order Filled](#) script sections.

Changes to an exit order values are limited. i.e. fill price, quantity, custom value, rules. Attempts to change values in the Exit Order Filled script section won't have any effect.

Links:[Order Functions](#), [Order Properties](#), [Creating Orders](#)**See Also :**[Alternate Order Object](#), [Order Object](#)

8.3 Creating Orders

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.

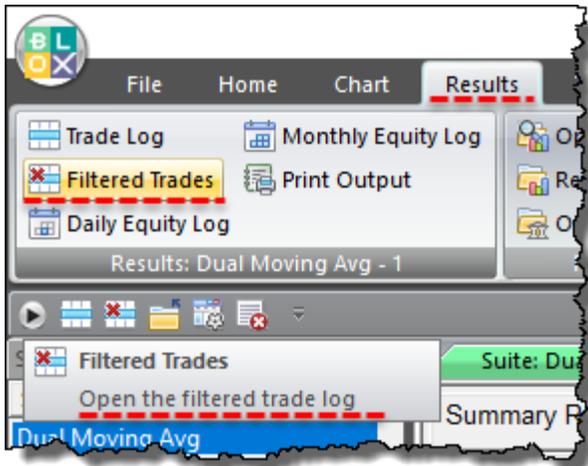
Entry orders not rejected are then sent to the [Can Add Unit](#) script section where other rules and conditions can be applied to allow or reject the order. Rejected orders will no longer exist after the [Can Add Unit](#) script terminates execution. Orders that are not rejected will be accessible in 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:



Filtered Trade Log Menu Access

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.

Syntax:

[See Broker Object Examples](#)

Parameter:**Description:**

[See Broker Object Examples](#)

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
```

Returns:

Above code snippet example uses the [System Object](#) property [orderExists\(\)](#) to discover when an order was successful created, or rejected. Use this property after the broker order execution returns to the script section where the broker object was executed.

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.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

User's Guide Generating Orders

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 240

8.4 Order Functions

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 `order.SetQuantity` function to set the quantity of the order.

Notice 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:
SetHidden	IB. Sets if the order will be hidden (IB definition). Format is True or False.
SetOutsideRTH	IB. Sets if the order can be filled outside of Regular Trading Hours. Format is True or False
Reject	Rejects the order to stop further processing, and sets the reject message that is printed in the Filtered Trade Log
SendToIB	IB Deprecated. Send the order to IB. Created for those that need discreet control of the IB process and use multiple accounts.
SetAlgoParameter	IB. Sets the parameters for the Algo Strategy set with SetAlgoStrategy. Format is text: Tag, Value. Function can be called multiple times.
SetAlgoStrategy	IB. Sets the Algo Strategy. Resets the Parameters. Format is text name of strategy.
SetClearingIntent	IB. Sets the clearing intent property for IB orders. The default is set by the system.SetClearingIntent, and is "IB".
SetCustomValue	Sets the custom value number
SetFillPrice	Sets a new fill price for the order
SetGoodAfterDateTime	IB. Set the good after date time for IB
SetGoodTillDateTime	IB. Set the good till date time for IB
SetLimitPrice	Sets a new profit taking limit for the order. This value is not used automatically by Trading Blox.
SetOCAGroup	IB. Set the OCA group for IB
SetOrderReportMessage	Sets field in the Position & Orders report
SetQuantity	Sets quantity of the order
SetRoutingExchange	IB. Set the routing exchange used by IB if different than the default exchange
SetRuleLabel	Sets the Rule Label for the order as seen on the trade chart

Order Function:	Description:
SetSortValue	Sets the sort value number of the order, for use in sorting the orders
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.
SetTimeInForce	IB. Sets the time in force property for IB orders. The default is "GTC"
SetTransmit	IB. Set the transmit flag for the order. If true the order will be automatically transmitted. If false it will be held in TWS.
SetOrderRef	IB. Sets the reference information for the orders. Often set with the System name for clarity in TWS.

Links:
AlternateOrder Object , AlternateSystem Object , Changing Orders , Order Object , Order Properties
See Also:
Alternate Order Object , Order Object

Edit Time: 3/21/2024 11:44:42 AM

Topic ID#: 450

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:**Description:**

message

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
' ~~~~~
```

Links:

[bigPointValue](#), [entryRisk](#), [SetQuantity](#), [tradingEquity](#), [order.referenceID](#)

See Also:

Links:

[Can Add Unit](#), [Can Fill Order](#), [Unit Size](#), [AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

Edit Time: 3/21/2024 11:45:12 AM

Topic ID#: 495

SetAlgoParameter

SetClearingIntent

This function is an advanced "IB" ([InteractiveBrokers](#)) trade clearing option. Clearing can be "IB" to clear the trade at "IB", or "Away" which means the trade is placed with [InteractiveBrokers](#), but cleared elsewhere.

Syntax:

```
order.SetClearingIntent( "textClearingOption" )
```

Parameter:

textClearingOption

Description:

Use one of the following:
 "IB" to clear the trade at [InteractiveBrokers](#)
 "Away" to place the trade at [InteractiveBrokers](#), but clear it at another facility.

Returns:**Example:**

```
' When the order returns after the Broker Statement,
' Check to ensure the order is active. Then
' only use one the following statements to specify
' where the order is to be cleared.
If system.OrderExists THEN
  ' Use this statement to clear the order at IB
  order.SetClearingIntent( "IB" )
  OR
  ' Use this statement when the order is placed
  ' at IB, but it is not cleared at IB
  order.SetClearingIntent( "Away" )
ENDIF
```

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:**Description:**

Parameter:	Description:
value	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
```

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:	Description:
fillPrice	Price at which the order is filled

Returns:

[order.fillPrice](#) assigned in the [Can Fill Order](#) script section, or the price assigned by the software's built-in fill algorithm

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 )
```

Links:

[fillPrice](#)

See Also:

[Can Fill Order](#), [AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetLimitPrice

Sets a new profit taking limit for the order.

This value is not used automatically by Trading Blox.

Syntax:

```
order.SetLimitPrice( anyPriceValue )
```

Parameter:	Description:
anyPriceValue	Price value that will exit the position when the data reaches the specified price level.

Returns:

The position will exit when the specified price value is reached. Nothing will happen if the price doesn't reach that value.

Example:

Links:

See Also:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

SetOrderRef

To set the IB orderRef property use order.SetOrderRef

Syntax:

```
order.SetOrderRef( IOrderRef )
```

Parameter

:

Description:

IOrderRef

Sets the IB Order Reference ID

Example:**Returns:****Links:****See Also:**

Edit Time: 3/21/2024 11:15:44 AM

Topic ID#: 725

SetOrderReportMessage

Send information about an Order reason so that it will appear in the Positions and Order Report.

Syntax:

```
order.SetOrderReportMessage( message )
```

Parameter:**Description:**

message

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
```

Example:**Results:**

```
order.ruleLabel = Lx@100.35s
```

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:**Description:**

Parameter:	Description:
quantity	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
' ~~~~~
```

Links:

[bigPointValue](#), [entryRisk](#), [Reject](#), [tradingEquity](#)

Links:**See Also:**

[Can Add Unit](#), [Can Fill Order](#), [Data Group and Types](#), [Unit Size](#), [AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

Edit Time: 3/21/2024 11:45:38 AM

Topic ID#: 550

SetRuleLabel

Send order information about an order's reason to the Trade Log report, and so it appears in the Trade Log list below the chart area.

Syntax:

```
order.SetRuleLabel( ruleLabel )
```

Parameter:**Description:**

ruleLabel

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:
```

Example:

```
Print "order.ruleLabel = ", order.ruleLabel
```

Results:

```
order.ruleLabel = Lx@100.35s
```

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 )
```

Parameter:	Description:
sortValue	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

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 Use 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:**Description:**

Parameter:	Description:
stopPrice	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 )
```

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](#)

Edit Time: 3/21/2024 11:46:08 AM

Topic ID#: 562

SetTimeInForce

This function sets the time in force property for IB orders. The default entry is "GTC"

Syntax:
<code>order.SetTimeInForce("GTC")</code>

Parameter:	Description:
"GTC"	IB Order use "GTC" = "Good Until Cancelled."

Returns:

Example:

Links:

See Also:
AlternateOrder Object , AlternateSystem Object , Order Functions , Order Object , Order Properties

8.5 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 [SetAlternateOrder](#) system function.

Properties:	Description:
blockName	Name of the order's originating block.
brokerSymbol	Returns the broker symbol
continueProcessing	When this property return a TRUE condition, the order can continue. When this property returns a FALSE condition, 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[x] property so that it can be used in the system, and discovered in the system's TradeLog .
entryRisk	Entry risk of the order. Value is the difference between the order-price and the stop-price. This is not adjusted by the fill price.
executionType	Returns 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	Calculated fill price of an order. This value is not used automatically by Trading Blox.
isBuy	TRUE when order is a buy order
isEntry	TRUE when the order is an entry. When it is an Exit order, the results will be False .
isWholeExit	TRUE when Exiting a position unit where the order.quantity = instrument.unitQuantity[x] Note: Each unit in a multiple unit position will have an entry and exit order. Some exit orders can be less the total unit. This property only return information about the unit number being examined.
limitPrice	Profit taking limit price of the order
noStopPrice	TRUE when order has no order.stopPrice has no value
orderPrice	Price of the order for stop or limit orders uses the close price for OnOpen orders and OnClose orders
orderReportMessage	Order report message as set by order.SetOrderReportMessage("Message")

Properties:	Description:
orderType	Order is a string with one of the following: "Long Entry", "Short Entry", "Long Exit", "Short Exit"
position	order.position returns an integer where: 1 = LONG and -1 = SHORT. Constants LONG and SHORT can be used for comparison purposes.
processingMessage	Contains the reject message when an order is rejected.
quantity	Quantity of shares for Stock Class instruments and contracts for Futures Class instruments.
referenceID	Unique order reference ID value
ruleLabel	Information added by <code>order.SetRuleLabel("text")</code> after the order has been created.
scriptName	Script section name where an order was created.
sortValue	Numeric sort value assigned by <code>order.SetSortValue(iNum)</code>
stopPrice	The protect stop price of the order. This value is used automatically on the order entry day, if Entry Day Retracement is greater than zero. Retained in <code>instrument.unitExitStop</code> .
symbol	Returns the market symbol of the order.
symbolIndex	Returns the instrument.priorityIndex from the System that created the order. Can be used with LoadSymbol together with System Index.
symbolRef	When using <code>Instrument.LoadSymbol(order.symbolRef)</code> , this property enables a direct access to order instrument that is quicker than using the instrument's symbol with LoadSymbol
symbolWithType	Useful when looping over orders and want to load the order's symbol. Example S:IBM or F:GC.
systemBlockName	Combined system and block name of the order's originating system blox.
systemIndex	Returns the system index. The systems are loaded alphabetically and looped over by index for placing orders.
systemName	Returns name of system that created the order.
unitNumber	Returns the unit number of the order.
IBOrderNumber	Returns the order number assigned by IB
timeInForce	Returns the TIF of the order.
clearingIntent	Returns the Clearing Intent of the order.
routingExchange	Returns the Routing Exchange of the order.

Properties:	Description:
goodAfterDateT ime	Returns the Good After Date Time of the order.
goodTillDateTi me	Returns the Good Till Date Time of the order.
outsideRTH	Returns if the order is allowed to fill outside of Regular Trading Hours
hidden	Returns if the order is flagged as hidden by IB
algoStrategy	Returns the Algo Strategy 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 functions are available to reference and changes.

Links:
AlternateOrder Object , AlternateSystem Object , Changing Orders , Order Functions , Order Object
See Also:
Alternate Order Object , Order Object

blockName

This property contains the name of the blox that created the order.

Example:

```
' To access see where the order was created,...  
PRINT "order.blockName = ", block.blockName
```

Returns:

Returns the name of block that created 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](#), [Order Properties](#)

See Also:

brokerSymbol

When the symbols assigned to an instrument is different than the symbol assigned by the brokerage, the dictionary provides an additional column where the Broker's symbol can be entered.

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:

--

clearingIntent**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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

continueProcessing

This property can return a **True** or **False** condition based upon what happens in the Unit size script, or in this script section. When an order arrives in this script section and its [order.continueProcessing](#) property returns a condition = **True**, the order is still active because it hasn't been rejected. When the [order.continueProcessing](#) property = **False**, the order has been canceled.

Orders are rejected using the [Order.Reject](#) ("reason for rejection text") function as it is processed in the [Unit Size](#) script section.

Example:

```
' Check the Rejection status of the order
If order.continueProcessing = TRUE THEN
  ' Continue Processing Order
  ' Apply other filters to the order
ELSE
  ' Do Not Continue Processing Order
  ' Ignore the rejected order
ENDIF
```

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:

[Broker Functions](#), [Can Place Order](#), [Unit Size](#), [Can Add Unit](#),

customValue

This property returns the user created value that was assigned to the order.customValue property.

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, = ", order.customValue  
' ~~~~~
```

Returns:

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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#), [SetCustomValue](#)

See Also:

entryRisk

When an order is created, the entry risk that was calculated when a protective exit price is included will determine the risk.

Example:

```
Print "order.entryRisk = ", order.entryRisk
```

Returns:

The order's risk amount in points.

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:

executionType

Property `<Object_Name>.executionType` returns the text description of how this order will be executed when brokered. Order execution property value is controlled by the [Broker Object](#) function used to generate the order.

Prefix "`Object_Name`" can be `order` or `alternateOrder`, when an order is being accessed outside of a script when instruments are in context.

Execution Types:	Execution Requirements
at Market	All these order types are "at Market" executions that will happens when the order is given to your brokerage.
on Open	
on Close	
on Stop	Execution happens when Stop Price Conditions are enabled by the market.
on Stop Close	
on Stop Open	
Limit	Execution happens when Limit Price Conditions are enabled by the market.
on Limit Open	
at Limit Close	

Example:

Trading Blox order execution types are controlled by which [Broker Object](#) function is selected for generating the order.

Information about what order execution is active is available by accessing property with [Order Object](#) or [AlternateOrder Object](#) this way: `<object_name>.executionType`

```
' Send Order's execution method to Print Output.csv file
Print order.executionType
```

Returns:

Any one of the execution type names listed in the table above.

[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](#), [Broker Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

fillPrice

Fill price is the price returned by the market. When slippage is enabled, the price will be different that the price specified a On Stop or At Limit controlled entry.

Example:

```
' To access see the order's fill price,...  
PRINT "order.fillPrice = ", order.fillPrice
```

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 is 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:

goodAfterDateTime

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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

goodTillDateTime**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:

hidden

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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

IBOrderNumber**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:

isBuy

This property will report the type of order that is being examined.

Example:

```
' To access see where the order was created,...  
PRINT "order.isBuy = ", order.isBuy
```

Returns:

When the order is a Long entry, the return will be TRUE.

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:

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 391

isEntry

Entry orders are created with `order.inEntry` property set to **True**. Exit orders show the `order.isEntry` is set to **False**.

Example:

```
' Is the order an Entry Order....  
If order.isEntry = TRUE THEN  
  ' Process Order for a New Entry  
ELSE  
  ' Process the Order as an Exit or Size Reduction  
ENDIF
```

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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

isWholeExit

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:

limitPrice**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

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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

orderPrice**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:

orderReportMessage

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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

orderType

Property `<Object_Name>.orderType` returns the text description of this order type of order when brokered. Order type property value is controlled by the [Broker Object](#) function used to generate the order.

Prefix "`Object_Name`" can be `order` or `alternateOrder`, when an order is being accessed outside of a script when instruments are in context.

Order Types:	Use Description:
Long Entry Short Entry	Any of the various order execution types that will generate a Long or Short entry order
Long Exit Short Exit	Any of the various order execution types that will generate a Long or Short exit order.

Example:

Trading Blox order types are controlled by which [Broker Object](#) function is selected for generating an order.

Information about what order type is active is available with [Order Object](#) or [AlternateOrder Object](#) this way: `<object_name>.executionType`

```
' Send Order's execution method to Print Output.csv file
Print order.executionType
```

Returns:

Any of the order type [Broker Object](#) methods available.

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

Notes:

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:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 455

OutSideRTH**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 :

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 686

position

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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

processingMessage**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:

quantity

The number of shares or contracts assigned to an order can be returned using this order property.

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:

--

referenceID

Each order created is assigned a reference ID number so the order can be accessed when needed. The number is assigned as part of the order's creation

Syntax:

```
Print order.referenceID
```

Parameter:**Description:**

none

Property will return the order's assigned number when it is accessed after it has been created.

Returns:

The number assigned to the most recent order created.

Example:

```
' To get the order number of the most recent order created
Print order.referenceID
```

Returns:

Number of the order just created.

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:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 491

ruleLabel

When an order is created or being adjusted or removed, it is often helpful to have the reason for the order to be added to the output of the results.

Example:

```
' When an order is created, a message can be helpful.
order.SetRuleLabel( "Order Reason" )
'
' To access see what instrument created the order,...
PRINT "order.ruleLabel = ", order.ruleLabel
```

Returns:

Order Reason will be returned if that was the text inserted by the script.

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:

scriptName**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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

sortValue

This value is an assigned order value by the .

Example:

```
' An order sort value can be assigned to an
' new order using this function.
'
' Update the sorting order value to this order
order.SetSortValue( 12 )
'
' To access see what instrument created the order,...
PRINT "order.SortValue = ", order.sortValue
```

Returns:

The value returned will be the value previously assigned.

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:

stopPrice

The protective price associated with the order when it was created, or added before the order is completed, will appear when the property is used in a **PRINT** statement.

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:

symbol

This property will reveal the symbol of the instrument that created this order.

Example:

```
' To access see what instrument created the order,...  
PRINT "order.symbol = ", order.symbol
```

Returns:

The return will be the symbol letters associated with the symbol's 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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

systemBlockName**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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

systemName**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:

unitNumber

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 functions are available to reference and changes.

Links:

[AlternateOrder Object](#), [AlternateSystem Object](#), [Order Functions](#), [Order Object](#), [Order Properties](#)

See Also:

Section 9 – Script

Trading Blox Builder Script Object allows any user to create a custom processing method. This ability is similar to how a subroutine or a function is created. Once a Custom-Script is created, it can be executed when the the purpose of the Custom-Script is needed.

Everything needed to create the Custom-Functions and Custom-Properties is built into the abilities of **Trading Blox Basic**. With these abilities and the understanding of what is needed from a customized process, the custom script can be called when it is available within a system list of blocks and script sections.

In simple terms, a Custom-Process/Function creates a Custom Script section that contains scripts that can perform a process that isn't already built-into Trading Blox Builder.

Note:

There is only one Script Object. A custom Script Functions can call other custom script. Because there is only one script object, the calling script will loose its parameter values during the process of the custom function being called. To prevent that from happening, if the parameter values of the calling custom script are needed, they should be assigned to local variables in the calling script before that script calls another custom function. Keep in mind that each call to a custom function will overwrite any passed in parameters, and passed back return values that haven't been assigned to a local variable.

Script Object Functions & Properties:

All functions and properties require the script-object name as a prefix word connected by a period '.' to a function or property keyword: `script.` plus the `FunctionName` or `propertyName`.

```
' This Script built-in Function will execute
' the text-named script section shown next:
script.Execute("AnyNameScript")
```

Similar to: `script.Execute()` is the function: `Script.InstrumentLoop()`. This additional function will run a custom script that will loop each enabled instrument in the portfolio. Click on this [InstrumentLoop](#) link for an example of how it can be used.

Functions Name:	Descriptions:
<code>Execute()</code>	Use this function to Execute a custom script name that is available to the system.
<code>Exists()</code>	This function will return the existence of a custom script. This function can be Executed any block to discover the existence of the named custom script exists is included or not found in the system. The ability to know a script exists prior to making an <code>script.Execute("Script-Name")</code> call provides an opportunity to

Functions Name:	Descriptions:
	<p>know if a named custom script is available before an error happens.</p> <p>This next Code-Example provides a process that can be used to know what scripts are in the system and how to effectively access them quicker.</p> <pre> ' ----- ' script.Exist & script.GetReference Example ' ----- ' When not certain, test to ensure Custom ' Script "MyCustomScript" is accessible. If script.Exists("MyCustomScript") THEN ' Get the Custom Scripts Numeric ID ' for Faster Custom-Script access. BPV_ID = script.GetReference("MyCustomScript") ' Script is Available, call the script ' script using Reference ID-Number script.Execute(BPV_ID) ELSE ' Script is NOT Available, do something else PRINT "Bad New !!!" ENDIF ' ----- </pre> <p>Custom-Script sections now have the ability to GetReference a ID for each custom script name in a system. Read the GetReference function for more information about its purpose and use.</p>
GetReference()	<p>The GetReference function returns an integer reference ID value that makes access to a custom script more efficient. Once a custom script reference is found, the ID-number can then be used to access that script faster and efficiently than will happen when the name of the custom script section is used.</p> <p>Review the Example Code in the Exists function description.</p> <p>In the Before Test script, it is possible to get the Custom-Script's ID for each of the Custom-Script section that can be saved in a BPV-Integer variables so they are available to access each Custom-Script during a test. In use, it is faster to use Reference-ID numbers to access a Custom-Script ID number than to use the Custom-Script's "text-name".</p>
GetSeriesValue()	<p>Used to access a value from a passed in series. Access is as defined for the series so that auto indexed series use an offset base, and non auto indexed series are manual direct index access.</p>
InstrumentLoop()	<p>This function is similar to Script.Execute function. This function when run in a custom script will access each enabled instrument in the portfolio.</p>

Functions Name:	Descriptions:
	To enable or disable an instrument, use the instrument.DisableTrading function to change the instrument's inPortfolio state from TRUE to FALSE . Typically this inPortfolio state is executed from SetParameters script section. Instrument data that is disabled will not be loaded into memory when the Before Test script section runs.
OrderLoop()	Loops over all open orders in the system and calls the custom script.
SetReturnValue()	Used to set the return value to a number or string.
SetReturnValueList()	Used to set a list of number return values.

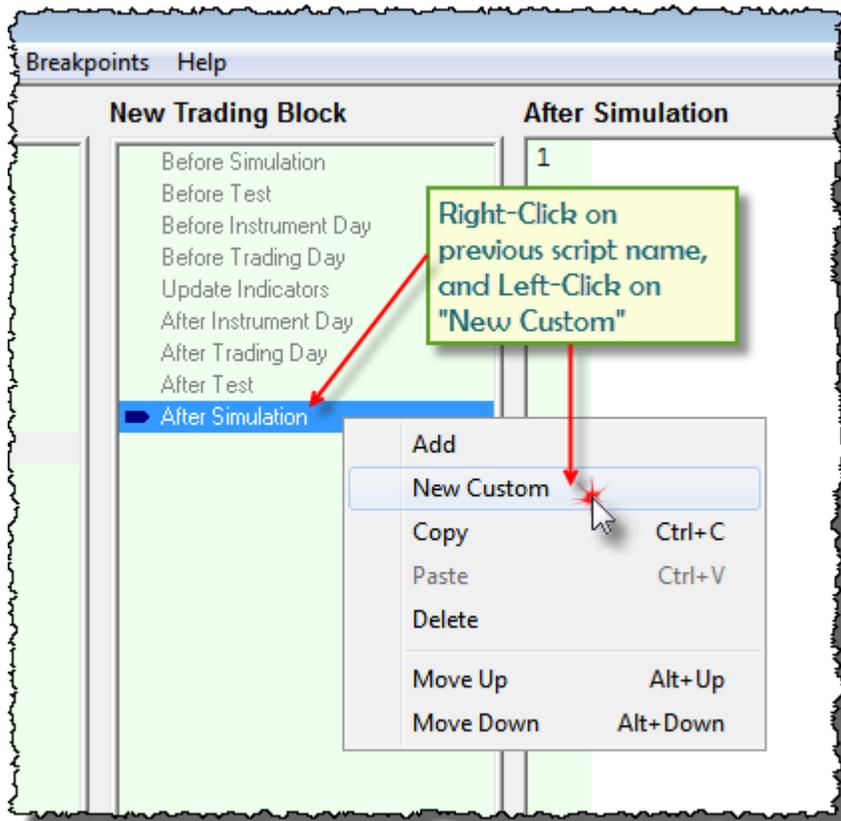
Script Object Properties:

Properties Name:	Description:
parameterCount	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.

9.1 Creating Custom Scripts

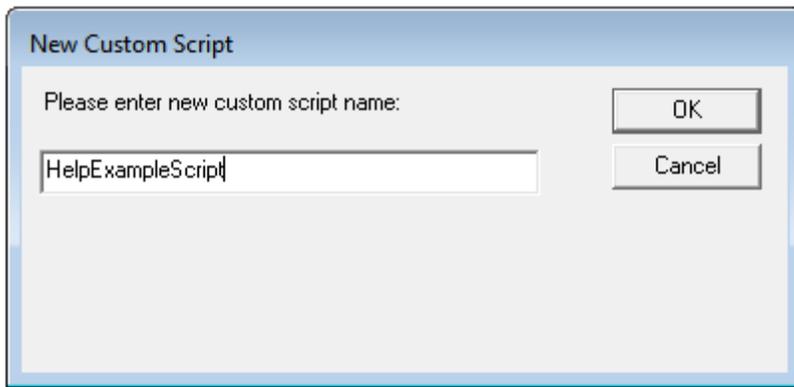
Custom scripts are created by creating a user-name for a new script section in a blox. Name used when creating the script must not be one of the standard script names Trading Blox provides. Instead it should be a name that best describes the purpose of the custom script. Whatever name is used, the name 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

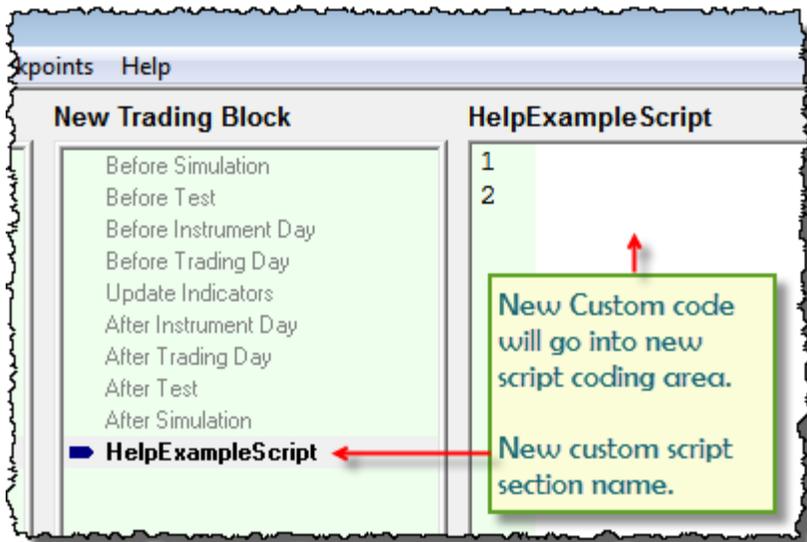
When the New Custom menu item is clicked in the pop-up menu the next dialog will appear.



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

9.2 Custom Parameters

Custom Function Parameter Index Values:

Each Parameter used in a Custom Function is given an index value. Index values are given to each data-type that is passed into a Script-Object function. This ability allows different types of data can vary in each left to right placement of parameters.

Trading Blox Data Types:

- Numeric : Any Integer, Boolean (TRUE/FALSE), Selector, or Floating value.
- String: Any text value.
- Series: Any Auto-Index or Manual-Index Series

Within a Custom-Function it is easy to get how many parameters of each data-type are available by using one or all of the following Script-Object properties:

Example:

```
' Count numeric parameters
x = script.ParameterCount
' Count String parameters
y = script.stringParameterCount
' Count Series parameters
z = script.seriesParameterCount
```

Each of the above script parameter properties will return how many Numeric, String, and Series (Auto-Index or Manual-Index) parameters were passed into the body of a Custom Function. Once it is known how many of each of data-type function is available each parameter can be accessed using their sequence location as their index value.

Parameter Counting Sequence:

Sequence of each data-type is assigned a value of 1 for the left-most parameter of each data-type up to its parameter count value as the count moves to the right. This means each data-type will have a "1" value that increments to the count value of the right-most parameter of that data-type.

In this next example we have 3-numeric, 1-string, and 1-series parameters.

Example:

```
' ~~~~~
' Method to Call the Execution of Custom Script from
' within the script area of the calling script section.
' ~~~~~

' Custom of a Mixed Parameter Function Example:
script.Execute("NumericExample", _      ' Function Name
              value1, value2, value3, _  ' 3-Numeric Parameters
              sTestItem, _              ' 1-String Parameter
              AsSeries(aAnySeries) )    ' 1-Series Parameter
' ~~~~~
```

Example:

```

' -----
' Placed within the Custom Function's Scripting area:
' -----
' Parameter counts within the above function will return:
' Count numeric parameters
PRINT "Numeric Count: ", script.parameterCount
' Count String parameters
PRINT "String Count: ", script.stringParameterCount
' Count Series parameters
PRINT "Series Count: ", script.seriesParameterCount
' -----

```

RETURN:

```

Numeric Count: ,3
String Count: ,1
Series Count: ,1

```

Access to each of the passed parameters is available by using the access property for each data type and then using that data-type's index value location of where it is placed in the calling script list of parameters:

Parameter access:

```

' Within Custom Function Body
NumVal1 = script.parameterList[1]      ' Numeric
NumVal2 = script.parameterList[2]      ' Numeric
NumVal3 = script.parameterList[3]      ' Numeric

TxtVal1 = script.stringParameterList[1]  ' STRING

' x = Any known element index value
' Incrementing an index is often done in a FOR-Loop
SeriVal[x] = script.GetSeriesValue( 1, x )  ' Series Element x

```

9.3 Custom Script Use

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.

9.4 Creating A User Function

Once the script space area has been created as shown in [Creating Custom Scripts](#), that space is where the method needed from a special function can be scripted. A user's Custom Function can be complex, or as simple as adding two numbers, or changing the case of any or all text values.

Values to be used in a user function are sent to the user function by adding parameters to the calling statement of the user function:

Syntax:

```
' This statement will work
script.execute("AddTwoNumbers", AnyRandom1, AnyRandom2)
```

Parameter:

Description:

Returns:

Example:

Links:

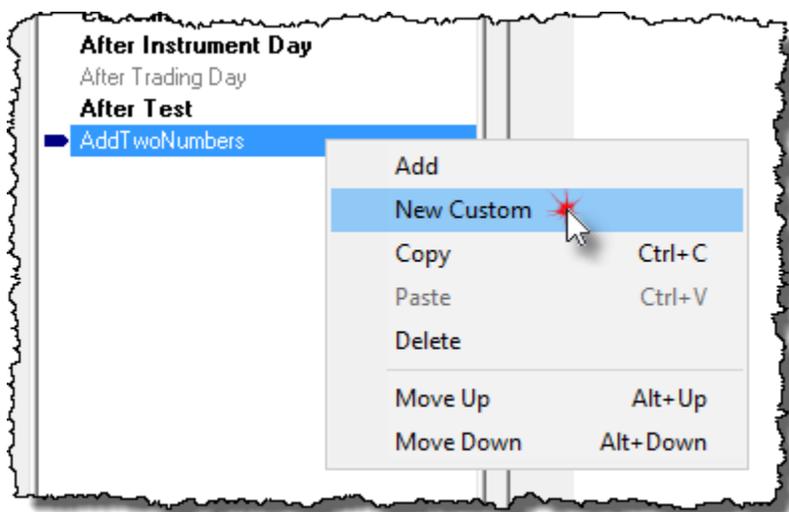
See Also:

Example Description:

This example will add two numbers together. To create a custom function, we need to give the function a name. In this example we'll call the function "**AddTwoNumbers.**" This function will have two parameters that will pass the two numbers we want to add together.

```
' Assign a value to each of the two Floating Point BPV values:
AnyRandom1 = Random( 1, 100 )
AnyRandom2 = Random( 1, 100 )
```

Custom Functions can be created in their own Blox module, or they can be added to a Blox that is already created. Let's start with adding a Custom Script Section to an existing Blox:



Add to Blox a Custom Script Section

When a custom script section is added to an existing blox, the addition of the numbers is all that is required to obtain a return value.

Before we call the custom script, we need to put numbers into each of the two parameters:

Calling "AddTwoNumbers" needs a statement that will access the created custom script section and pass that section the value of two number:

```
' This statement will work
script.execute("AddTwoNumbers", AnyRandom1, AnyRandom2)
```

To call this function, we will give the two

To execute that function we need to call that function from another script. Any other script can call any other script section. This also includes any of the Trading Blox normal script sections.

For our example we need to pass our function two values. How we get those values is not important, but they must be valid within the script area from which we call this custom function:

Within another blox that isn't the Custom Script, we need to have two values we can pass:

```
' Create two local floating point values
VARIABLES: anyValue1, anyValue2, sumValue TYPE: Floating
' Access the values in each of the numeric
' parameters passed to this function
anyValue1 = script.parameterList[1]
anyValue2 = script.parameterList[2]

' Add the numbers together
sumValue = anyValue1 + anyValue2
```

our custom script section we will use two script properties to access the passed value so we can add them together:

Example Description:

Calling "AddTwoNumbers" needs a statement that will access the created custom script section and pass that section the value of two number:

```
' This statement will work
script.execute("AddTwoNumbers", AnyRandom1, AnyRandom2)
```

Within our custom script section we will use two script properties to access the passed value so we can add them together:

```
' Create two local floating point values
VARIABLES: anyValue1, anyValue2, sumValue TYPE: Floating
' Access the values in each of the numeric
' parameters passed to this function
anyValue1 = script.parameterList[1]
anyValue2 = script.parameterList[2]

' Add the numbers together
sumValue = anyValue1 + anyValue2

' Return the sum of the two values
' to the calling script section
script.SetReturnValue( sumValue )
```

If the value of the returned numbers were 1 and 2 the return value will equal 3.

Access to a single return value, multiple values can be returned, is available in three ways:

```
' To directly return the results of a called script
' let the calling statement assign its return value
addedValue = script.execute("AddTwoNumbers", AnyRandom1, AnyRandom2)

' To access the value returned using a property, place
' the property assignment after the calling statement
addedValue = script.returnValue

' To send the returned value to the Print Output.csv file
' call the custom script statement using the PRINT function.
PRINT script.execute("AddTwoNumbers", AnyRandom1, AnyRandom2)
```

If more examples are needed, use the Help file's email option or use the static Help file topic in the forum.

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 were 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.

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 237

9.5 Custom Function Example

This source code below was created in 2008 and it is still being used today, but there is now a Trading Blox function: [GetFieldCount\(\)](#)
This new function performs a similar count.

This custom function `Field_Count` returns the number of characters in the tested string that are a comma. Each comma found increments the field count. This source ensures that when no commas are in a text string, but the character count of the string is greater than zero, the count will be one.

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 the count of the number of comma separated values is
'   needed required for the GetField function 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)
```

Example:

```

' 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 THEN
      ' 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
' =====

```

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 242

9.6 Script Functions

Script Object Functions & Properties:

All functions and properties require the script-object name as a prefix word connected by a period '.' to a function or property keyword: `script.` plus the `FunctionName` or `propertyName`.

```

' This Script built-in Function will execute
' the text-named script section shown next:
script.Execute( "AnyNameScript" )

```

Similar to: `Script.Execute()` is the function: `Script.InstrumentLoop()`. This additional function will run a custom script that will loop each enabled instrument in the portfolio. Click on this [InstrumentLoop](#) link for an example of how it can be used.

Functions Name:	Descriptions:
Execute()	Use this function to Execute a custom script name that is available to the system.

Functions Name:	Descriptions:
<p><code>Exists()</code></p>	<p>This function will return the existence of a custom script. This function can be Executed any block to discover the existence of the named custom script exists is included or not found in the system.</p> <p>The ability to know a script exists prior to making an <code>script.Execute("Script-Name")</code> call provides an opportunity to know if a named custom script is available before an error happens.</p> <p>This next Code-Example provides a process that can be used to know what scripts are in the system and how to effectively access them quicker.</p> <pre> ' ----- ' script.Exist & script.GetReference Example ' ----- ' When not certain, test to ensure Custom ' Script "MyCustomScript" is accessible. If script.Exists("MyCustomScript") THEN ' Get the Custom Scripts Numeric ID ' for Faster Custom-Script access. BPV_ID = script.GetReference("MyCustomScript") ' Script is Available, call the script ' script using Referemce ID-Number script.Execute(BPV_ID) ELSE ' Script is NOT Available, do something else PRINT "Bad New !!!" ENDIF ' ----- </pre> <p>Custom-Script sections now have the ability to GetReference a ID for each custom script name in a system. Read the GetReference function for more information about its purpose and use.</p>
<p><code>GetReference()</code></p>	<p>The GetReference function returns an integer reference ID value that makes access to a custom script more efficient. Once a custom script reference is found, the ID-number can then be used to access that script faster and efficiently than will happen when the name of the custom script section is used.</p> <p>Review the Example Code in the Exists function description.</p> <p>In the Before Test script, it is possible to get the Custom-Script's ID for each of the Custom-Script section that can be saved in a BPV-Integer variables so they are available to access each Custom-Script during a test. In use, it is faster to use Reference-ID numbers to access a Custom-Script ID number than to use the Custom-Script's "text-name".</p>

Functions Name:	Descriptions:
GetSeriesValue ()	Used to access a value from a passed in series. Access is as defined for the series so that auto indexed series use an offset base, and non auto indexed series are manual direct index access.
InstrumentLoop ()	This function is similar to Script.Execute function. This function when run in a custom script will access each enabled instrument in the portfolio. To enable or disable an instrument, use the instrument.DisableTrading function to change the instrument's inPortfolio state from TRUE to FALSE . Typically this inPortfolio state is executed from SetParameters script section. Instrument data that is disabled will not be loaded into memory when the Before Test script section runs.
OrderLoop ()	Loops over all open orders in the system and calls the custom script.
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
for
' 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

Any of the standard, or user created custom, script sections can be called using this `script.Execute("AnyScriptName")` function.

Syntax:

```
script.Execute( "CustomScriptName", [parameterlist...] )
Or
script.Execute( "EXIT ORDERS" )
```

Parameter:	Description:
"CustomScriptName"	Custom script's given name when it was created.
parameterlist	Optional: 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](#), [Script Functions](#), [Script Properties](#)

See Also:

GetSeriesValue

Used this function to access a value in a series that is passed to Script-Object function.

Access to a series is defined as follows:

- Auto-Indexed properties or Auto-Index series are offset based.
- Non-Auto-indexed series are manual direct index based.

Access to any series to be passed requires the use of the [AsSeries\(\)](#) function.

Script-Object parameters are numbered by their sequence location in the left-to-right order in how they appear in the [script.Execute\(\)](#) call statement.

See [Custom Parameters](#) for more details.

Syntax:

```
y = script.GetSeriesValue( SeriesParameterNumber , offsetOrIndex )
```

Parameter:	Description:
SeriesParameterNumber	Series parameter sequence location number.
offsetOrIndex	Element offset for use with Auto-Index series, or a series element index for use with Manually index series.

Example:

```
' -----
' AsSeries() function to send is required when a series,
' is passed as a Script-Object parameter. AsSeries()
' passes the series memory location. Without the
' AsSeries() function only the value of an element is
' passed and other elements will not be accessible.
' -----

' Count the number of series parameters
x = script.seriesParameterCount

SetSeriesSize( testSeries, 10 )
For i = 1 TO 10
    testSeries[ i ] = Random( 100 )
Next

For i = 1 TO 10
    PRINT i, "Series Value", testSeries[ i ], _
        script.Execute( "SeriesExample", _
            AsSeries( testSeries ), i )
Next
```

Example:

```
' -----
```

Note:

Earlier version of `GetSeriesValue()` required a "-1" added to the users offset to get the correct series element. This next example shows how to adjust the offset:

```
' SeriesExample is defined as:  
  script.SetReturnValue( script.GetSeriesValue( 1, x-1 ))
```

Results:

Value in each accessed element of the series.

Links:

[Script](#), [Script Functions](#), [Script Properties](#), [AsSeries\(\)](#), [SetReturnValue](#), [Execute](#), [SetSeriesSize](#), [ParameterList](#), [seriesParameterCount](#)

See Also:

InstrumentLoop

This function should be used in the [Set Parameters](#) script section because it executes **after** the Before Simulation and **before** the Before Test sections.

This function is similar to the way that the [Script.Execute](#) function will execute "[UserCreatedCustomName](#)." Both the [Script.Execute\(\)](#) and this function [script.InstrumentLoop\(\)](#) will call a custom script named section to perform tasks.

See the [Creating Custom Scripts](#) topic for more information.

When this function is used, it will call a "[UserCreatedCustomName](#)" that **will** be able to **automatically loop-over** all of the instruments in the system's portfolio.

The list for changing an instrument, is a list created by the user. The script will use those some or all of those symbol when it is deciding to change the state of the [instrument.inPortfolio](#) status. When a symbol match is found, the [instrument.DisableTrading](#) function will change the state to **False**. Which symbols in a portfolio are removed from trading is dependent upon how the conditional conditional statements are created for matching a symbol in the portfolio with a symbol-name on the list.

Syntax:

```
' Function calls the Custom-Script created for
' handling the instrument and or indicator
' activity state.
script.InstrumentLoop( "anyCustomScriptName" )
```

Parameter:

anyCustomScriptName

Description:

Custom script's name created to perform a task.

Example:

```
' Custom script adds two numbers together
Print script.InstrumentLoop( "DisableSymbolList" )
```

Returns:

The return will be the execution of the scripts in the custom script name used as the script parameter.

Example:

This is an example of how a custom script can be created to disable symbols in a portfolio from being used by the system. The name of the custom script created is "[iOnOff_Instrument](#)."

Example:

```

' =====
' Control Active Symbols
' iOnOff_Instrument - SCRIPT START
' =====
' This Script Section Disables specific Instruments in a
' Portfolio. It then Display Current Date, Symbol, and its
' Trading State condition.
' ~~~~~
' Create Temp Variables for Print & Debug Stepping Display
VARIABLES: iDate, iInPortfolio      Type: Integer
VARIABLES: sSymbol                  Type: String
' -----
' Temp Print & Debug Assignments
iDate = instrument.date
sSymbol = instrument.symbol
iInPortfolio = instrument.inPortfolio
' -----
' Display the loaded instrument.
PRINT instrument.date, instrument.symbol, instrument.inPortfolio
' If the symbol is listed here, turn make it InActive
If ( sSymbol = "CD" ) OR ( sSymbol = "DJ" ) THEN

    ' Remove From Portfolio
    instrument.DisableTrading

    ' Example code in a custom script called
    ' "iOnOff_Instrument"
    iDate = instrument.date
    sSymbol = instrument.symbol
    iInPortfolio = instrument.inPortfolio

    ' Display Current Date, Symbol, and its Trading State
    PRINT instrument.date, instrument.symbol, instrument.inPortfolio
ENDIF

' Add a Space between each symbol
PRINT
' ~~~~~
' =====
' iOnOff_Instrument - SCRIPT END
' Control Active Symbols
' =====

```

Returns:

System's Portfolio has four symbols:

AD, DJ, ED, CD

Prior to this custom script running, all of the instruments were active and would load if this script had not been run.

Calling Script Name: Set Parameters

Output shows the Date when the instrument's `inPortfolio` property was changed.

Returns:		
<code>instrument.inPortfolio</code> at start, shows all the symbols will show TRUE .		
When the <code>instrument.DisableTrading</code> is executed, it will change to FALSE .		
Date:	Symbol	<code>inPortfolio</code> -State
20140402	AD	1
20140402	DJ	1
20140402	DJ	0
20140402	ED	1
20140402	CD	1
20140402	CD	0

Links:
Script , Script Functions , Script Properties
See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 95

OrderLoop

`Script.InstrumentLoop`

Similar to `Script.Execute`, runs a custom script for each enabled instrument in the portfolio. Sets the default instrument object.

Typically used in script with no instrument context, like the Before Trading Day or After Trading Day scripts, to loop over the portfolio instruments.

Example code in a custom script called "instrumentloopscript"

```
PRINT instrument.date, instrument.symbol
```

Example code in the Before Trading Day script:

```
script.InstrumentLoop( "instrumentloopscript" )
```

Syntax:
<code>script.InstrumentLoop("instrumentloopscript")</code>

Parameter:	Description:
"CustomScript Name"	Custom script's given name when it was created.

Parameter:	Description:
parameterlist	Optional: 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:
<pre>' Custom script adds two numbers together Print script.Execute("AddTwoNumbers", 2, 2)</pre>
Return:
4

Example:
<pre>' Custom script adds two numbers together value = script.Execute("AddTwoNumbers", 2, 2) Print value</pre>
Return:
4

Links:
Script , Script Functions , Script Properties
See Also:

SetReturnValue

Used this function to return a value from a Number or a String.

When a numeric value is being returned use the [script.returnValue](#) property.

When a string value is returned use the [script.stringReturnValue](#) property.

Syntax:

```
script.SetReturnValue( anyValue )
```

Parameter:

Description:

Parameter:	Description:
anyValue	Value can be any numeric or string value.

Example:

```
' Numeric Example
SizeToRemove = ( 1 * 4 )

' Return numeric value
script.SetReturnValue( SizeToRemove )

' Return a text value
PRINT script.returnValue
' -----
sStringItem = "ABCD"
' Return a string value
script.SetReturnValue( sStringItem )
```

Results:

```
Numeric value returned: 4
String value returned: ABCD
```

Links:

[Script](#), [Script Functions](#), [Script Properties](#)

See Also:

SetReturnValueList

Use this function when a custom script function will need to return more than one value.

Syntax:

```
' Used to set a list of number return values.  
script.SetReturnValueList( value1, value2, value3... )
```

Parameter:	Description:
value1	First of a list of values to be returned.
value1	Second of a list of values to be returned.
value3	Third of a list of values to be returned.
...	Additional value(s) that can be returned.

Example:

```

CUSTOM FUNCTION -> START =====
' =====
' AnyFunction01 - START
' =====
' ~~~~~
' AnyFunction01 example uses a simple Fibonacci process
' to increment a value for each value variables
' -----
' script.Execute("AnyFunction01", AnyNum1)
' -----
VARIABLES: x      Type: Integer
VARIABLES: value1, value2, value3, value4, value5  Type: Floating
' -----
' Get parameter value
value1 = script.parameterList[1]

' Calculate first 5-Fibonacci numbers
If value1 > 0 THEN
    value1 = value1 + 0
    value2 = value1 + value1
    value3 = value2 + value1
    value4 = value3 + value2
    value5 = value4 + value3
ELSE
    value1 = 0
    value2 = 0
    value3 = 0
    value4 = 0
    value5 = 0
ENDIF ' value1 > 0
' -----
' Used to set a list of number return values.
script.SetReturnValueList( value1, value2, value3, value4, value5 )
' -----
' =====
' AnyFunction01 - END
' =====
CUSTOM FUNCTION -> END =====

```

Example:

```
' Custom Function Calling Script
' ~~~~~
VARIABLES: AnyNum1    Type: Floating
' ~~~~~
' AnyNum1 is gets its value from a parameter in this example
AnyNum1 = startValue

' "AnyFunction01" is a user created function shown below
script.Execute("AnyFunction01", AnyNum1)
' ~~~~~
PRINT "----- ",
PRINT "value1 ", script.returnValueList[1]
PRINT "value2 ", script.returnValueList[2]
PRINT "value3 ", script.returnValueList[3]
PRINT "value4 ", script.returnValueList[4]
PRINT "value5 ", script.returnValueList[5]
PRINT "----- ",
' ~~~~~
```

Results:

```
----- START,
value1 ,1.000000000,
value2 ,2.000000000,
value3 ,3.000000000,
value4 ,5.000000000,
value5 ,8.000000000,
----- END,
```

Links:

[script.Execute](#), [Script](#), [Script Functions](#), [Script Properties](#),
[script.returnValueList](#)

See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 553

9.7 Script Properties

Script Object Properties:

Properties Name:	Description:
parameterCount	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.

Properties Name:	Description:
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.

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

Count of the number of parameters passed into the custom function.

In the topic: [Custom Parameters](#) where the [Parameter Counting Sequence](#) section describes how parameters of each data-type are sequenced, the details will help to understand the order in which each parameter of a different type is given an index value.

Syntax:

```
value = script.parameterCount
```

Parameter:	Description:
------------	--------------

<none>	
--------	--

Returns:

Count of script created parameters.

Example:

```
' ~~~~~
' In the script calling this function
' of a Mixed Numeric & String Parameter
' Custom Function
script.Execute("NumericExample", _
              value1, value2, value3, _
              sTestItem )
' ~~~~~

' -----
' Contained in the Custom Function shown
' above this is shown.
VARIABLES: n, s TYPE: INTEGER
' Count the number of numeric parameters
n = script.ParameterCount
' Count the number of String parameters
s = script.stringParameterCount

' Show parameter value results
Print n, s
' -----
```

Results:

3,1

Links:

[Script](#), [Script Functions](#), [Script Properties](#), [ParameterList](#), [StringParameterList](#)

Links:
See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 456

ParameterList

Access a specific parameter, or a number of parameters is accomplished by referencing the parameters by its index number in each of the two different list of customer parameters.

In Custom Scripts references there can be a numeric, or a string parameter type. Numeric Parameters are indexed in the numeric list of parameters, and the String parameters are listed in the string list of custom parameters.

This means that if there are three numeric parameters, and four string parameters, the numeric list will reference each of the three numeric parameters by a different integer index value, and each of the string parameters will be listed in the string parameter list by an integer index value. Index values for both types of parameter start at one, and increment up the count of each type of parameter.

Parameters are listed in both the numeric list and in the string list by the order in which they are added to their type of parameter list. Both types of parameters are assigned an integer value that is determined by their left to right order in which they are listed in the custom script call statement.

Syntax:

```
' Custom Script numeric parameter access reference
NumericValue = script.parameterList[IndexNumber] ' Numeric

' Custom Script string parameter access reference
TestValue = script.stringParameterList[IndexNumber] ' STRING
```

Parameter	Description:
:	
[IndexNumber]	Index number of passed Numeric Parameter. Index number increments for each Numeric Parameter. Index position of 1 is assigned to the left-most Numeric Parameter. Additional Numeric parameters get an incremented value of 1 added their index value.

Example:

Parameters are indexed by their left to right placement in the '`script.Execute.`' location.

In the next statement, there is one String parameter, and two numeric parameters. The string parameter in the next scripted call statement will assign the String Parameter index value of 1. In the same statement the left most numeric parameter, '`fFirstNumber`', will be assigned an index value of 1 and '`fSecondNumber`' numeric parameter will be assigned an index value of 2:

```
' Custom Function Call Statement Types
script.Execute( tTextValue, fFirstNumber, fSecondNumber )

' Custom Function executing statement values on this next line.
script.Execute( "AddNumbers", 3.40, 0.25 )
' ~~~~~
VARIABLES: tTextValue                               Type: STRING
VARIABLES: fFirstNumber, fSecondNumber              Type: FLOATING
' -----
' Get File item string assigned
fValue1 = script.parameterList[1]
fValue2 = script.ParameterList[2]
textValue = script.stringParameterList[1]
```

Results:

```
PRINT iValue1 ' Output will show 3.4
PRINT iValue2 ' Output will show .25
PRINT textValue ' Output will show 3.65
```

Links:

[Script](#), [Script Functions](#), [Script Properties](#)

See Also:

ReturnValue

Used to return a numeric value from a custom function.

Syntax:
<code>lResult = script.returnValue</code>

Parameter:	Description:
<None>	Values assigned using the <code>script.SetReturnValue()</code> function.

Example:
<pre>' 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</pre>
Results:
Print statement will display: 4

Links:
Script , Script Functions , Script Properties
See Also:

ReturnValueList

Used this function when the custom script function is going to return more than one value.

Syntax:

```
returnvalue1 = script.returnValueList[1]
```

Parameter:**Description:**

<number
>

Number value used here will return the value at comma location in the return list of values.

See example shown below to see how the values as passed and accessed.

Example:

```

CUSTOM FUNCTION -> START =====
'
' =====
' AnyFunction01 - START
' =====
'
' ~~~~~
' AnyFunction01 example uses a simple Fibonacci process
' to increment a value for each value variables
' -----
' script.Execute("AnyFunction01", AnyNum1)
' -----
VARIABLES: x      Type: Integer
VARIABLES: value1, value2, value3, value4, value5  Type: Floating
' -----
' Get parameter value
value1 = script.parameterList[1]

' Calculate first 5-Fibonacci numbers
If value1 > 0 THEN
value1 = value1 + 0
value2 = value1 + value1
value3 = value2 + value1
value4 = value3 + value2
value5 = value4 + value3
ELSE
value1 = 0
value2 = 0
value3 = 0
value4 = 0
value5 = 0
ENDIF ' value1 > 0
' -----
' Used to set a list of number return values.
script.SetReturnValueList( value1, value2, value3, value4, value5 )
' -----
' =====
' AnyFunction01 - END
' =====
CUSTOM FUNCTION -> END =====

```

Example:

```
' Custom Function Calling Script
' ~~~~~
VARIABLES: AnyNum1    Type: Floating
' ~~~~~
' AnyNum1 is gets its value from a parameter in this example
AnyNum1 = startValue

' "AnyFunction01" is a user created function shown below
script.Execute("AnyFunction01", AnyNum1)
' ~~~~~
PRINT "----- ",
PRINT "value1 ", script.returnValueList[1]
PRINT "value2 ", script.returnValueList[2]
PRINT "value3 ", script.returnValueList[3]
PRINT "value4 ", script.returnValueList[4]
PRINT "value5 ", script.returnValueList[5]
PRINT "----- ",
' ~~~~~
```

Results:

```
----- START,
value1 ,1.000000000,
value2 ,2.000000000,
value3 ,3.000000000,
value4 ,5.000000000,
value5 ,8.000000000,
----- END,
```

Links:

[script.Execute](#), [Script](#), [Script Functions](#), [Script Properties](#),
[script.SetReturnValueList](#)

See Also:

SeriesParameterCount

Return the number of Numeric-Series Type parameters passed to a custom function.

Syntax:

```
x = script.seriesParameterCount
```

Parameter:**Description:**

<none>

Example:

```
' Custom Function with 3-Numeric Parameters
script.Execute("SeriesExample", AsSeries(aSeriesName), 0 )

' Place this property in the custom function
x = script.seriesParameterCount

PRINT "x ", x
```

Results:

```
x , 1
```

Links:

[Script](#), [Script Functions](#), [Script Properties](#), [ParameterCount](#), [StringParameterCount](#)

See Also:

StringParameterCount

Return the number of String-Type parameters passed into a custom function.

Syntax:

```
x = script.stringParameterCount
```

Parameter:	Description:
<none>	

Returns:

Count of script created string parameters.

Example:

```
' Example Custom String Function with 2-string parameters
script.Execute("StringExample", test1, test2)

' Place this property in the custom function
x = script.stringParameterCount

PRINT "x ", x
```

Results:

```
"x ", 2
```

Links:

[Script](#), [Script Functions](#), [Script Properties](#), [ParameterCount](#), [SeriesParameterCount](#)

See Also:

StringParameterList

Used to access one specific String parameter, or a number of String parameters.

String Parameters are indexed independently from Numeric parameters.

Index is sequenced from left to right in the order in which they are passed to the custom function.

Syntax:

```
sStringValue = script.stringParameterList[StringIndexNumber]
```

Parameter:	Description:
StringIndexNumber	Index number of passed String Parameter. Index number increments for each String Parameter. Index position of 1 is assigned to the left-most String Parameter.

Returns:

String value passed by the selected script created parameter.

Example:

When a String-Value is one of the items passed to a custom function the "[StringIndexNumber]" is based upon the order in which the string-values are listed in the calling statement.

In this example there is an Numeric parameter and a String parameter. Each will use 1 as their index to pass their value to the custom function's local Variables.

```
' Custom Function Statement
'script.Execute( "FileNameExists", iFileType, sFileName )

' Custom Function is Executed on this next line.
script.Execute( "FileNameExists", 1, test.futuresDataPath )
' ~~~~~
VARIABLES: sFileItem_FINE           Type: String
VARIABLES: iFileType_FINE, iCreatOK_FINE   Type: Integer
' -----
' Get File item string assigned
iFileType_FINE = script.parameterList[1]
sFileItem_FINE = script.stringParameterList[1]
```

Results:

```
PRINT iFileType_FINE ' Results will show 1
PRINT sFileItem_FINE ' Results will show "Default Trading Blox Futures
Data Drive and Path
```

Links:

[Script](#), [Script Functions](#), [Script Properties](#), [ParameterList](#), [ReturnValue](#), [StringReturnValue](#)

See Also:

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 590

StringReturnValue

Function will access any String value returned using the [script.SetReturnValue](#)

Use this property to return to the calling script the single string being returned.

Multiple String Returns will use [script.stringParameterList](#) property.

Syntax:

```
sTextItem = script.stringReturnValue
```

Parameter:**Description:**

<none>

Example:

```
' ~~~~~
' Within a Custom Function where a simple
' text item is given a value
sStringItem = "ABCD"

' After All the calculates are done, use
' this function to Return a string value
script.SetReturnValue( sStringItem )
' ~~~~~

' -----
' In the script where the Custom Function is called,
' use this function's string return value.
PRINT "String Value Returned: ", script.stringReturnValue
' -----
```

Results:

String Value Returned: ABCD

Links:

[Script](#), [Script Functions](#), [Script Properties](#), [ParameterList](#), [ReturnValue](#), [StringReturnList](#)

See Also:

Section 10 – System

A system's object properties provide access to its system's information.

Since systems control the portfolio results, the most common use for system properties is to determine the number of instruments in the current portfolio, and how to work with a symbol.

System objects can also be used to determine dynamic correlations between instruments in the portfolio.

System Functions & Control Areas:	Description:
Accessing System Portfolio Instruments	Any instrument can be accessed within a system.
Global Suite System	A global suite system is a special type of system. When a system has the same name as the suite, it will be by definition a Global Suite System (GSS).
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	Functions used to make changes to a system being tested.
System Properties	System information in a system being tested.

10.1 Global Suite System

Creating a Global Suite System (GSS):

Suites can support more than one system. It can be an added Global Suite System (**GSS**) to provide more access capabilities to a suite of systems.

Adding a **GSS** to a suite provides the ability to access one or all of the systems in the suite. This added capability can happen in a manner that is harder to do in a Suite that only has a standard system.

GSS Suite's can access each system's changes in a review of each system's same script name makes it simple to understand that script's changes.

An example that might be needed in a suite of system where a GSS is active is to limit how many of the systems can take a position in the same market. If the limit for the suite was to have only one crude oil in the suite, the other systems will know an order to enter crude oil is too many.

In addition, the **GSS** can remove a pending order in the [Before Order Execution](#) script to prevent a second crude oil position. Another example might be in how capital allocations can be adjusted by the system's demonstrated performance.

Access to a system's performance in the After Trade Day script section provides updated access to each system's performance. Suites that know how the systems are performing, enables an adjustment to a system's allocation, or a signal to change the risk exposure during the next trade day.

How is this possible? Global Suites have access to many of the same script sections that are in a standard system. The access by a **GSS** can be ahead of the system's access in some of the same name script section, or after each same name script section. Not all of the script section supported by a standard system are available to a GSS, but there are enough of them to provide opportunity to make changes during the testing.

To understand which script sections can be support by a **GSS**, or to understand when some script sections execute ahead of , or after the standard system same name script sections, click this link: [Global Script Timing](#) table.

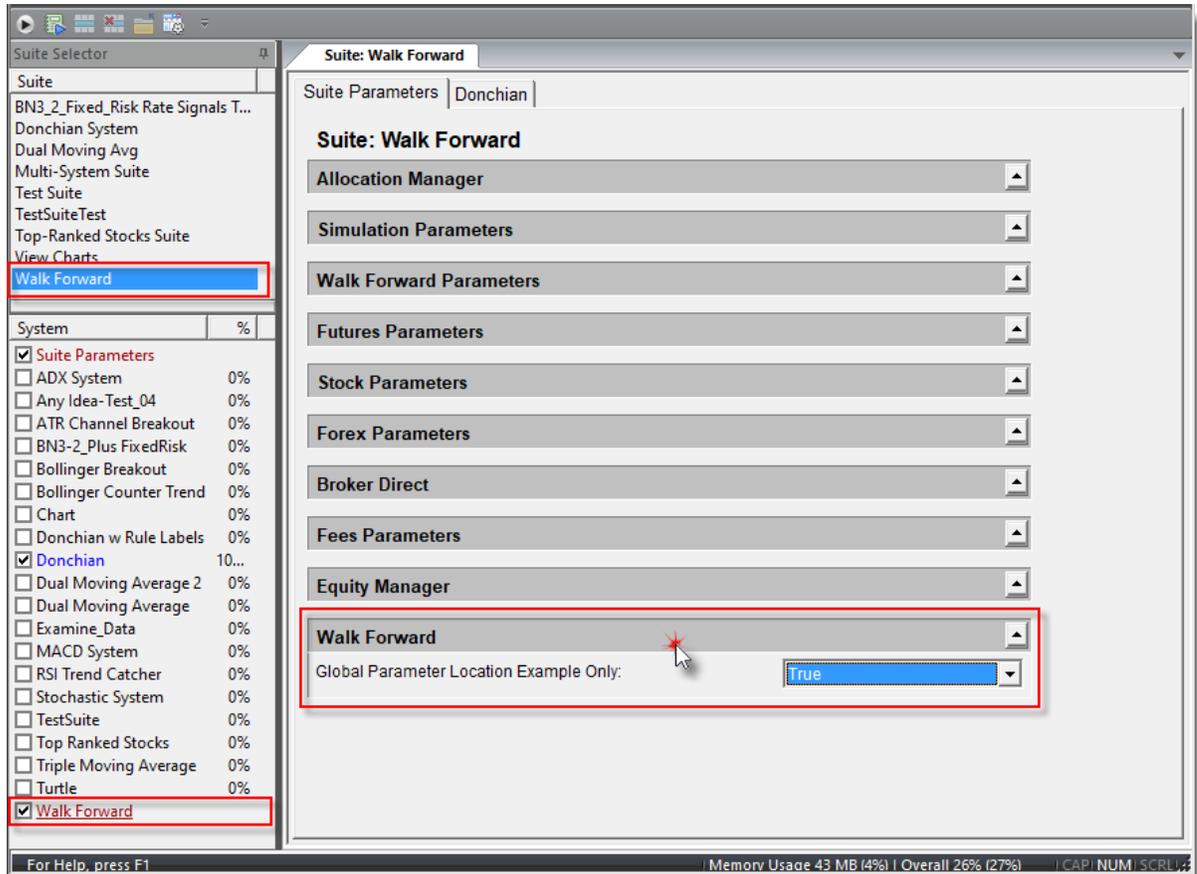
Creating a Global Suite System:

The name of a **GSS** systems must be the same as the Suite where it will be attached. For example, the default Suite "**Walk Forward**", will automatically attach a **GSS** system named "**Walk Forward.**" All **Global Suite Systems** are assigned a [system.index](#) value of 0, and a [system.name](#) "**Global Parameters**".

The first standard system assigned to that suite is assigned a system index of one. A second system assigned is given the system index of two, the third is given the value of three. As each additional system is assigned to the same suite it will get the next incremental integer value.

GSS Parameter Access:

Any **GSS System** that has Parameters in a **GSS** blox, the parameters will be displayed at the bottom of the **Suite Parameter** Tab. The parameters in the **GSS** blox will appear after the **Equity Manager** parameter section:



GSS Blox Parameters are in the Simulation Suite. Global Blox are display as the last item in the Simulation Suite's Trading Blox options.

Using GSS Script Sections:

Not all the available scripts sections in Trading Blox Builder can be executed in a GSS.

The GSS script sections that support Auto Context for instruments, are limited to:

- Can Add Unit
- Can Fill Order
- Entry Order Filled
- Exit Order Filled

Instruments can be accessed from the other script section when the [LoadSymbol](#) function provides Instrument Context.

A major purpose of the scripts in a **GSS** system is to influence one or more of the suite's systems, or to harvest information from the systems the GSS can analyze and make decisions that it can apply to a system. Harvested and analyzed information can be used to monitor and adjust how the suite is performing.

The scripts sections in a blox that are selected to be used with as a **GSS** will be processed using different rules than those for normal systems. 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-order 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](#). The `'.orderExists'` property checks for an available order to access.

Use [system.IsGlobalSuiteSystem](#) property to check if the system is a Global Suite System when it is important to know an order is available.

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. However, a **BPV** and a **IPV** 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 to 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 whether or not the script is 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.

Example:

```
PRINT
PRINT test.currentDate

totalRisk = 0

' Loop over all the instruments that are used in this test.
```

Example:

```

' Includes all systems and all support forex conversion markets.
' For Each instrument in the test,...
For testInstrumentIndex = 1 TO test.instrumentCount STEP 1
  ' Get the symbol name.
  PRINT "Getting instrument number", testInstrumentIndex

  ' Obtain the list of symbols in the test.
  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.
      test.SetAlternateSystem( systemIndex )

      PRINT "Setting to system index", 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
            ' Get instrument's current net risk
            suiteInstrument.netRisk = suiteInstrument.netRisk _
              +
systemInstrument.currentPositionRisk
          PRINT "Long net risk: ", suiteInstrument.netRisk
          ENDIF

          If systemInstrument.position = SHORT THEN
            ' Get instrument's current net risk
            suiteInstrument.netRisk = suiteInstrument.netRisk _
              -
systemInstrument.currentPositionRisk
          PRINT "Short net risk: ", suiteInstrument.netRisk
          ENDIF
        ELSE
          PRINT "Not in portfolio for system", alternateSystem.name

```

Example:

```
        ENDIF
    ELSE
        PRINT "Unable to load", suiteInstrumentSymbol
    ENDIF
Next ' systemIndex

' Whether positive or negative, it's still risk.
suiteInstrument.netRisk = Abs( suiteInstrument.netRisk )

' Update Total Risk
totalRisk = totalRisk + suiteInstrument.netRisk

' Print out the net risk & Total Risk for this instrument.
PRINT "Net Risk  ", suiteInstrument.netRisk
PRINT "Total Risk ", totalRisk
ELSE
    PRINT "Unable to load", suiteInstrumentSymbol
ENDIF
Next ' testInstrumentIndex

totalRisk = totalRisk / test.totalEquity * 100
PRINT "Total Risk All: ", totalRisk
```

Returns:**Links:****See Also:**

10.2 System Functions

Referencing any System Object function requires the "`system.`" prefix be appended to a System Object function.

Syntax:
<pre>' Sort Instruments by Dictionary Order Value: system.SortInstrumentList(3)</pre>

System Function:	Description:
<code>Extract</code>	Extracts all data series and indicators for all instruments into a file. Optional parameters to control date range and file name. Typically used in the After Test script for audit and reconciliation purposes.
<code>OrderExists()</code>	Returns true if the default Order object has context, and false if the Order object is null. Null orders are orders that were unable to complete. They cannot be accessed because they no longer exists.
<code>RankInstruments</code>	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.
<code>SetAccountNumber()</code>	Sets the IB account number for all orders in the system.
<code>SetAlternateOrder()</code>	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.
<code>SetClearingIntent()</code>	Set to "IB" to clear the system orders in IB, and set to "AWAY" when sending to IB but clearing elsewhere
<code>SendToIB</code>	This function is somewhat deprecated. It is for manual control of the IB process. Users can now connect to IB and send orders to IB through scripting. This function is mostly replaced by the integrated Broker Direct functionality that sends the orders automatically.
<code>SetOutsideRTH</code>	This function is used to enable or disable the outsideRTH property is a Boolean, and sets whether the IB order can trade outside of Regular Trading Hours. We just pass this to IB.
<code>SetRoutingExchange</code>	This function Sets the default routing exchange to be used for all instruments in the system, when sending orders to IB. The information to use with this function is controlled by IB. When it is used, the property information is stored in the System's property: routingExchange .

System Function:	Description:																		
<code>SetTimeInForce</code>	This function controls the timeInForce property that can be sent with an order. The information is passed to IB so they will know the intent of the order. An examples of what is used is: “ GTC ” and “ Day ”, etc. To get a list of what is allowed check with IB for the correct values.																		
<code>SetVirtualSystem()</code>	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.																		
<code>SortInstrumentList()</code>	Sorts the physical instrument list that is used for the simulation loop using the method indicated: <table border="1" data-bbox="506 730 1455 1339"> <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>7</td> <td>Short Ranked number.</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.	7	Short Ranked number.		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.																		
7	Short Ranked number.																		
	Note: All markets are sorted regardless of whether they are primed.																		
<code>SortOrdersBySortValue</code>	Sorts the open order list by the order sort value, as set into each order by <code>order.SetSortValue</code> . This function should only be used in the Before Order Execution script.																		
<code>TotalOrders()</code>	Use this function without a symbol to return all the open orders in the system. Use this function with a symbol to return the total open orders in the system assigned to the symbol.																		

Links:[System Properties](#)**See Also:**[Correlation](#), [CorrelationLog](#), [Correlation Functions](#),

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 610

Accessing System Portfolio Instruments

When you want to access one or more of the portfolio's instruments to read an instrument property, execution an instrument function or access one of the indicators associated with an indicators, this is possible using the BPV Variable Type Instrument.

BPV instruments can access and instrument in the portfolio, or they can emulate a specific instrument so that they can be easily accessible anywhere in the system without regard for script section's instrument context normal state.

To get access to a specific IPV instrument the target instrument must first be brought into context using Instrument Object's [LoadSymbol](#) function. This function will accept a portfolio index number in the range of 1 to the total number of instruments in the portfolio, or it will accept an instrument's symbol.

Naming the BPV instrument variable can be any name like "[portfolioInstrument](#)" as a replacement object name. This replacement of the instrument object name will make it easier to understand, and it will be available to use in scripts that have a automatic instrument context, and in scripts that don't support automatic instrument context.

A specific name like "[SPX_I](#)," "Corn," "Gold" and other names can be used. Specific names for an individual symbol reports the information contained is for only the symbol identified. When batch processing of instruments is needed, use a generic name. When a specific instrument needs to be available throughout the testing, use a specific name to make script coding more understandable.

More details are available here: [LoadSymbol](#)

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:

Example:

Block Permanent Variable

Name for Code: ← Temp. IPV Container Name

Name for Humans:

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:

Scope:

OK Cancel

Selecting Instrument as the data type let's Trading Blox know you will be working with IPV data and want to be able to access all IPV properties, functions and user added IPV variables.

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`

Example:

```
' 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:

[LoadSymbol](#), [Instrument Loading](#), [System Properties](#),

See Also:

Extract

The Extract Function can be used data series of as **System** object or **Instrument** Object process. In order for Extract to work, it must be used with a data series that has not been disabled.

Notes:

When used with an **instrument** object, it will export to a file the value for each series and indicator for that instrument, for each day in the test.

When used with a system object, it will export for all instrument in the system.

We recommend only running this once per instrument per test run, as a file is created each time the function is called.

By default, the extra file name will be “**Daily Extract XXX.csv**” where **XXX** is the symbol, and the extract will contain all available dates that have been loaded.

The file is placed at the top level of the Trading Blox Builder installation directory.

Caution:

Be careful with the `system.Extract` because each symbol in the portfolio will create an export file.

When there are many symbols in a portfolio file, the Extract function will create a file for each symbol, which might be more than what you intended.

Syntax:
<pre>' Export optional parameters instrument.extract([ExtractedFileName] [,StartingDate] [,EndingDate])</pre>
<p>OR</p> <pre>' Use default settings instrument.extract()</pre>
<p>OR</p> <pre>' Export all series data between 19990101 and 20060101, and use the default file name: instrument.Extract(19990101, 20060101)</pre>
<p>OR</p> <pre>' Create a file named "My Custom Name XXX.csv" for each instrument, ' and export all the series data between 19990101 and 20060101: instrument.Extract("My Custom Name", 19990101, 20060101)</pre>

Parameter:	Description:
[ExtractedFile Name]	Optional: Default file-name field is used when field is left empty.
[,StartingDate]	Optional: Beginning data date when start-date field is left empty.

Parameter:	Description:
[,EndingDate]	Optional: Ending data date when end-date field is left empty.

Example:

```
' Typical usage in the After Test script:
inst.LoadSymbol("GC")
inst.Extract()

OR
system.Extract()

OR
inst.LoadSymbol("GC")
If inst.Extract( "Output Series\\Symbol" ) THEN
    PRINT "Extract successful"
ELSE
    PRINT "Extract unsuccessful"
ENDIF
```

Returns:

The return value is TRUE if successful and FALSE if not. In this example, we output to a custom folder within the Trading Blox install directory. If this folder does not exist, the function will fail.

Date	Time	Open	High	Low	Close	Volume	Open Interest	Delivery Month	Un-adjusted Close	macd Indicator	average True Range	entry Breakout High	entry Breakout Low	exit Breakout High	exit Breakout Low	average True Range	offset Adjusted Entry High	offset Adjusted Entry Low	offset Adjusted Exit High	offset Adjusted Exit Low	testing
20110419	0	1551.8	1555.7	1543.4	1550.3	138785	537946	201106	1495.1	124.407	18.627	1555.700	1466.700	1555.700	1468.700	18.627	1555.700	1466.700	1555.700	1468.700	1546.180
20110420	0	1551.3	1561.7	1549	1554.1	155371	537767	201106	1498.9	125.260	18.331	1561.700	1466.700	1561.700	1468.700	18.331	1561.700	1466.700	1561.700	1468.700	1551.354
20110421	0	1557.2	1564.8	1555.6	1559	141956	531503	201106	1503.8	126.164	17.949	1564.800	1466.700	1564.800	1484.300	17.949	1564.800	1466.700	1564.800	1484.300	1555.836

Instrument.Extract Output Example from last script section above.

Links:**See Also:**

[Instrument Data Functions](#), [System Functions](#)

OrderExists

When a Broker function is called to create a new order, this system Function will report the existence or an order, or non-existence of order the most recent broker function order attempt.

When an order exists, this function return a [TRUE](#), and a [FALSE](#) when an order does not exists.

When a [Broker](#) functions is used to create an order it might get canceled by rules in other scripts. This most often will happen with Entry Orders, but it can in some cases happen with and Exit Order. In both cases it is always best to use this function before attempting to access any order object properties or functions (See Example below).

Orders that are not in context, or do not exists when scripts attempt to access them, will cause a run-time error during execution. When Order Object doesn't normally have [Blox Script Access](#), use the [AlternateOrder Object](#) to access any object property or function.

NOTE:

Always append an open and closed parentheses () symbol after its name (See examples).

Syntax:

```
x = system.OrderExist() ' parenthesis is left empty
```

Parameter:	Description:
<none>	Returns TRUE when an order exists, and if the script section has Blox Script Access for Order properties or function. When there isn't normal Blox Script Access , use AlternateOrder object to bring an order into context to access.

Returns:

When an order exists function will return [TRUE](#). When an order does not exist, or isn't in context, a [FALSE](#) condition returns.

Place the script for this property right after the broker function that is attempting to create an order. When this is done, the scripted property, `system.OrderExists()` will return a [TRUE](#) when the order is created, and a [FALSE](#) when the order has been rejected.

Example:

```
' Generate a Long Entry on Next open,  
' without a protective exit price  
broker.EnterLongOnOpen  
  
' Create details why an order is created.  
sRuleLabel = "Order Details go here."  
  
' When Broker Order Exist,...  
If system.OrderExists() THEN  
' Apply Order Information to Position & Order Report  
order.SetOrderReportMessage( sRuleLabel)  
  
' Apply Order Information to the Trade Log  
order.SetRuleLabel( sRuleLabel)  
ENDIF ' system.orderExists
```

Links:

[AlternateOrder Object](#), [Blox Script Access](#), [Broker](#), [Order](#)

See Also:

[Data Group and Types](#)

RankInstruments

This function ranks the instrument by long sorting value and short sorting value. This replicates the action taken between the Portfolio Manager's Rank Instruments script and the Filter Portfolio script.

Syntax:

```
system.RankInstruments()
```

Parameter:**Description:**

<None>

Returns:**Example:**

Can be used in the Before Trading Day script as a more flexible replacement for the Portfolio Manager. Do a manual loop over the instruments setting the Ranking Values, and then use [system.RankInstruments](#), and then do another manual loop over the instruments allowing and denying trading based on the resultant Rank.

Note that this only ranks the instruments to create a Long Rank and a Short Rank. To physically sort the instrument list to change the order in which they are looped over for order processing, use the [system.sortInstrumentList](#) function.

Links:

[system.sortInstrumentList](#)

See Also:

SetAccountNumber

This function sets the account number for use by Interactive Brokers. Orders placed from this system will default to this account number.

Syntax:

```
system.SetAccountNumber( "account Number String" )
```

Parameter:**Description:**

account Number	A string representation of the IB account number.
-------------------	---

Returns:**Example:****Links:****See Also:**

SetAlternateOrder

Access to an existing order is only possible when the script section can provide the order within the script's context. Many script sections normally have Order Object context. See table: [Blox Script Access](#).

Order Context is the normal a permission state enabled in a script section that allows direct [Order Object](#) access. When order script does not normally provided direct **Order Object** access, it can be provided by using [AlternateOrder](#) Object for any active order access.

Syntax:

```
system.SetAlternateOrder( [symbol], orderIndex )
```

Parameter:	Description:
[symbol]	<p>Symbol is an optional parameter and can be omitted. When used this function can bring order for a specific symbol into context and the index range to use should be within the range for a symbol shown in note #1.</p> <p>When all the orders are to be processed for possible changes, omit the symbol and use the index range shown in note #2.</p> <p>Notes:</p> <ol style="list-style-type: none"> 1. When used with a symbol in system.TotalOrders([symbol]) function the index value must be one to the value returned by the system.TotalOrders([symbol]) function. 2. When used with system.totalOpenOrders property the index value must be in the range of value returned by the system.totalOpenOrders property.
orderIndex	Any integer value must be one to the value returned by the returned by the TotalOpenOrders , or when a symbol value is return within the range of a symbol specific parameter with the TotalOrders function .

Example:

```

' ~~~~~
' When Orders are available,...
If system.totalOpenOrders > 0 THEN
' Loop through orders to process volatility sizing
For index = 1 TO system.totalOpenOrders STEP 1

    ' Bring this order record into Context
    system.SetAlternateOrder( index )

    ' Volatility Adjustments need a Std-Order Size to adjust
    If alternateOrder.quantity > 0 THEN
        PRINT test.currentDate, _
            sSymName, _
            system.totalOpenOrders, _
            system.TotalOrders

        PRINT test.currentDate, _
            sSymName, _
            system.totalOpenOrders, _
            system.TotalOrders(sSymName)

    ENDIF ' alternateOrder.quantity > 0
Next ' index
ENDIF ' system.totalOpenOrders > 0
' ~~~~~

```

Returns:

```

' Example Values When system.TotalOrders omits symbol:

```

t.cDate	sSymName	s.totalOpenOrders	s.TotalOrders
5/1/2014	SPY	5	5
5/1/2014	VTI	5	5
5/1/2014	XLB	5	5
5/1/2014	XLE	5	5
5/1/2014	XLK	5	5

```

' Example Values When system.TotalOrders uses symbol:

```

t.cDate	sSymName	s.totalOpenOrders	s.TotalOrders(sSymName)
5/1/2014	SPY	5	1
5/1/2014	VTI	5	1
5/1/2014	XLB	5	1
5/1/2014	XLE	5	1
5/1/2014	XLK	5	1

Links:

[Alternate Objects](#), [AlternateOrder Object](#), [OrderFunctions](#), [Order Object](#), [OrderProperties](#), [TotalOpenOrders](#), [System Properties](#), [TotalOrders](#)

See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 530

SetRoutingExchange

Sets the default routing exchange to be used for all instruments in the system, when sending orders to IB

Syntax:

--

Parameter:**Description:**

Parameter:	Description:

Returns:

--

Example:

--

Links:

--

See Also:

--

SortInstrumentList

Ranked order values are created by the execution of [system.RankInstruments](#). Each time an instrument's ranking values are updated the [system.RankInstruments](#) function should be executed so the new order of instruments will be ranked using the updated ranking values.

Instrument Properties:

```
instrument.longRank
instrument.shortRank
instrument.longRankingValue
instrument.shortRankingValue
instrument.customSortValue
```

Instrument Functions:

```
instrument.SetLongRankingValue( )
instrument.SetShortRankingValue( )
instrument.SetCustomSortValue( )
```

Syntax:
<code>system.SortInstrumentList(iMethod)</code>

Parameter:	Description:
iMethod	The number listed in the next table describes the type of sorting that will happen.

Sorts the physical instrument list that is used for the simulation loop using the method indicated:

Method Value:	Sorting Method Description:
1	Long Ranked value sorts the instrument list by the current Long Rank number order.
2	Long Ranking script assigned value will be sorted in ascending order.
3	Dictionary assigned Original Order Sort value.
4	Custom Sort Value order.
5	Alphabetical by Symbol
6	Alphabetical by Group.
	<p>Note: All market symbols are sorted regardless of whether they are primed when the sorting happens.</p> <p>Instruments for these Sort Types need a dictionary source for these methods: 3 Order Sort Value</p>

Method Value:	Sorting Method Description:
	4 Custom Sort Value 6 Group Name

The ranking process that creates the Long Rank from the Long Ranking Value is the [system.RankInstruments](#) function and/or the Portfolio Manager internal process.

The Ranking process only ranks trading instruments (those that are primed and ready to be used in scripting). So as you said, you need to be aware of this if you set the Long Ranking Value, and then Rank the list, and then Sort the list by the Long Rank.

A more direct approach if you need the instrument list sorted, is to set the Long Ranking Value and then sort by the Long Ranking Value directly using the [system.SortInstrumentList\(2\)](#). Note that the Long Rank by definition is created by ranking the Long Ranking Value from highest to lowest, so the Long Rank is sorted "opposite" of Long Ranking Value.

As for markets that are not primed, you can check this when you use the Rank or Ranking Value, by checking the `instrument.isPrimed` property or the [instrument.tradesOnTradeBar](#) property if necessary. Note that the Entry and Exit scripts do not run if the instrument is not primed or does not trade on the trade date. So you only need these properties if looping over the instrument list manually. Likewise if you loop over the instruments manually to set the Long Ranking Value, you can set this to some out of bounds value as necessary to move to the back of the list. The only time you can't set the value is in the Portfolio Manager where the non primed instruments are not looped over.

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.

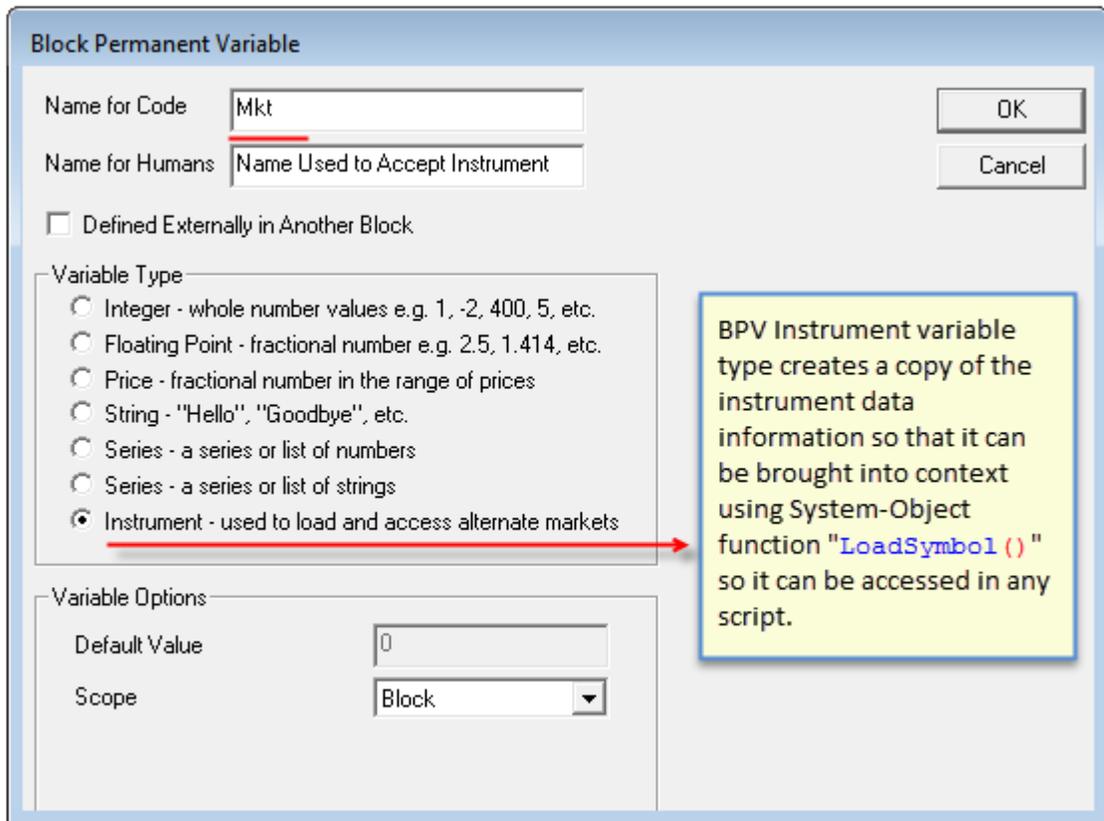
-- Loop over each instrument setting the custom sort value

-- 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.

Example:
<pre>' Create a BPV Instrument Variable Name</pre>

Example:


Block Permanent Variable

Name for Code:

Name for Humans:

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:

Scope:

OK

Cancel

BPV Instrument variable type creates a copy of the instrument data information so that it can be brought into context using System-Object function "LoadSymbol ()" so it can be accessed in any script.

BPV Instrument Variable Type Example

```

' Get Total Number of Instruments in Portfolio
iTotalSymbols = system.totalInstruments
' Sort Portfolio in Alphabetical Order
system.SortInstrumentList ( 5 )
' Create Column Header Titles
PRINT "Mkt#", "Mkt-Symbol"
' Generate a List of Portfolio Symbols
For x = 1 TO iTotalSymbols STEP 1
  ' Bring each instrument into Script Context
  iLoadOK = Mkt.LoadSymbol( x )
  ' When Instrument Loads OK,...
  If iLoadOK THEN
    ' Increment Loaded Instrument Counter
    iCount = iCount + 1
    ' Send Mkt Record to 'PRINT OUTPUT.csv' file
    PRINT iCount, Mkt.symbol
  ENDIF
Next ' x

```

Returns:

Mkt#	Mkt-Symbol
1	EFA

Returns:

2	GLD
3	IWM
4	LQD
5	MDY

Links:

[system.RankInstruments](#), [instrument.tradesOnTradeBar](#)

See Also:

TotalOrders

Total number of open orders in the system not filled or rejected. This property updates right after each order is created, and after any order is rejected.

When a symbol is entered return results will be the number of system orders for the symbol not filled or rejected.

Syntax:

```
iOrderCount = system.totalOrders( [symbol] )
```

Parameter:	Description:
[symbol]	<p>[symbol] is an optional parameter.</p> <ul style="list-style-type: none"> When a portfolio symbol is entered return results will be the number of system orders for the symbol not filled or rejected. When a portfolio symbol is omitted the number returned is the same value as the System.totalOpenOrders.

Notes:

[system.totalOpenOrders](#) and [system.totalOrders](#) can return the same value. They can also return different values when [system.totalOrders\(symbol \)](#) is called with a symbol value in its optional parameter.

Understanding how the two above methods can be used is in [system.SetAlternateOrder](#) topic.

Example:

```
' Show the number of orders open for this symbol
PRINT "VTI orders: ", system.totalOrders( VTI )
```

Results:

```
VTI orders: ,1
```

Links:

[Alternate Objects](#), [OrderFunctions](#), [Order Object](#), [OrderProperties](#), [TotalOpenOrders](#), [System Properties](#)

See Also:



Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 626

10.3 System Properties

System 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.

All the properties listed in this table will need to have the object name: `system.` appended to each property:

Property Name:	Description:
<code>allocationPercent</code>	Percentage of test equity available to the current system. Percentage is set using the Global Parameter Allocation Slider.
<code>canTradeInstruments</code>	Total number of instruments in the portfolio being tested that are primed and also have price information, and are allowed to trade by the instrument's Trade Control Properties.
<code>cash</code>	<p>Current cash of the system is determined by subtracting <code>system.currentMargin</code> from <code>system.closedEquity</code>.</p> <ul style="list-style-type: none"> • Futures margin is the value entered into the Futures Dictionary that is summed as the Futures margin amount. • Stock margin is the purchase cost of the instruments summed as the total Stock margin amount. <p>See definitions of: <code>system.currentMargin</code> <code>system.totalMargin</code></p>
<code>clearingIntent</code>	<p>Sets the default clearing intent to be used for all instruments in the system, when sending orders to IB.</p> <p>This is a string that that is used by IB – the user should check with IB to find out what the correct value should be for their situation.</p> <p>Trading Blox function is just passing the IB information back to them.</p>
<code>closedEquity[]</code>	The closed equity of the system as of the close of the prior day.
<code>coreEquity</code>	Core Equity is updated at each <code>test.currentDay</code> location of a test (not indexed).
<code>Correlation</code>	Removed. Use: Correlation Functions , and the math functions: Correlation , CorrelationLog

Property Name:	Description:
<code>currentClosedEquity</code>	Current closed equity of the system. Dynamically takes into consideration positions as they are exited.
<code>currentDrawdown</code>	Current draw down for the system as a percentage from the highest equity peak. Value returned is a positive number between 0% and 100%.
<code>currentMargin</code>	<p>With Futures this is the sum of the Futures margin in use. With Stock class instruments it is the sum of the cash committed to the stocks.</p> <p>Property <code>system.currentMargin</code> returns the margin amount without the margins planned in open orders.</p> <p>When the Futures margin or stock cash needed amount of open orders not yet filled or rejected is needed, use <code>system.openOrderMargin</code> as the incremental amount that will be used when those orders are filled.</p> <p>This property reports the total in use margin across all instruments, and its value is the same as <code>system.totalMargin</code>, which is now depreciated and will be removed from later versions.</p>
<code>currentOpenEquity</code>	Current open equity of the system. Dynamically takes into consideration positions as they are exited.
<code>currentRisk</code>	<p>Risk in dollars of all open positions in the system.</p> <p>Risk for a given position is determined by looking at the difference between the current close price and the protect stop price of each unit to determine the close to protective price spread in points.</p> <p>Points from each unit are multiplied by the <code>instrument.bigPointValue</code> for each instrument to determine a position's current risk amount. Value returned represents the total amount of risk from all active positions in the system.</p>
<code>index</code>	<p>An index number assigned to each system in a Suite by the order in which each system was attached to the Suite. Values for a system are in the range of 1 to the number of systems in the test. The property <code>systemIndex</code> is the same property.</p> <p>A system with the same name as the Suite is considered a Global Suite System (GSS) and that system will always be assigned an index value of zero.</p>
<code>isGlobalSuiteSystem</code>	Returns true if the system is a Global Suite System (GSS) .
<code>isVirtual</code>	Returns True when the system is operating in Virtual Mode.

Property Name:	Description:
marketOrdersValue	Total amount value needed to fill all the current open orders in the system. For futures, this is the margin times order size, for stocks this is the order price times the order size.
maximumDrawdown	Maximum draw down is the worst percentage value from a peak equity amount experienced by the system found up to that location during a test. Value returned is a positive number between 0% and 100%.
name	Name of the system. Useful for printing, and the same can be used access a different system by name in a multiple system suite. system.index can also be used to find a different system in the same suite.
netPositionValue	This property is for Stocks: <ul style="list-style-type: none"> • It computes the quantity times the close adjusted by stock ratio and currency conversion. • Adds long positions and subtracts short positions to come up with a Net Position Value.
openOrderMargin	Removed. Use system.marketOrdersValue instead.
outsideRTH	This property uses the SetOutsideRTH is the function that changes this property. It is also a Boolean Property. It is enabled when a IB order can trade outside of Regular Trading Hours. This information is something we just pass on to IB.
portfolioName	Name of the portfolio being used for the system.
portfolioType	Returns the class type of the system selected portfolio. See: Data Class Properties
purchaseEquity	Use with current stock positions. The system's Initial equity amount required to purchase the current open positions (entry fill price times the system's current quantity) is returned.
rankedInstruments	Function returns the number of ranked instruments in the system. The number of ranked instruments is the total number of instruments that are primed and ready to trade on the trading day. Review the following System Properties: <ul style="list-style-type: none"> • system.totalInstruments • system.tradingInstruments • system.canTradeInstruments • system.totalInstruments is the total number of instruments in the portfolio.

Property Name:	Description:
	<ul style="list-style-type: none"> • <code>system.rankedInstruments</code> is the number of primed instruments. • <code>system.tradingInstruments</code> is the number of primed instruments that trade on trade date. • <code>system.canTradeInstruments</code> is the number of primed instruments that trade on trade date that are set to AllowTrading. <p>Review the following trade control Instrument functions:</p> <ul style="list-style-type: none"> ◦ AllowAllTrades and DenyAllTrades.
<code>routingExchange</code>	<p>Returns the default routing exchange as set by SetRoutingExchange system Function..</p> <p>The information to use with this function is determine by IB.</p> <p>This property is a IB determined string that is controlled by the SetRoutingExchange function. Trading Blox Builder just passes this information to IB when it is needed.</p>
<code>systemIndex</code>	<p>This is property is the same as the system.index property. The value returned is the system number that created the order.</p>
<code>timeInForce</code>	<p>This property is controlled by the SetTimeInForce function. Its information is a String that is passed to IB so they will know the intent of the order. An examples of what is used is: "GTC" and "Day", etc. To get a list of what is allowed check with IB for the correct values.</p>
<code>totalEnabledInstruments</code>	<p>Returns the non-disabled instrument count. It can now include non portfolio instruments that can trade using the broker objects.</p>
<code>totalEquity[]</code>	<p>Uses the initial starting value entered in the Global Parameters, Equity Manager's Test Starting Equity setting. In scripting the value in the system.startingEquity property is the initialization value system.totalEquity.</p> <p>As gains and loss accumulate this property will reflect the summation of all system activity as testing makes progress. Updates to this property happen just after the Can Fill Order script sections finish executing. Adjust Stops section is the first script to execute with the current test day changes.</p> <p>This property can be Indexed to access previous total equity values. However in all the script sections that process ahead when system.totalEquity is updated there will be no difference in the value between the current and previous element values of this property. In otherwords, system.totalEquity[0] will equal system.totalEquity[1]. After the update for the test day has completed the timing of the Can Fill Order, the difference between</p>

Property Name:	Description:
	<p>these two elements will be the amount of change from the previous test day to the current test day.</p> <p>Test simulation with only one system in a Suite will have the same value in <code>system.totalEquity</code> as the <code>test.totalEquity</code>, minus interest when the system's System Allocation control slider is set to 100%.</p> <p>For order generation this is the Order Generation Equity times the allocation.</p> <p>Note: Value in this property in script ahead of when this property is updated will be the settled value of the prior day.</p>
<code>totalEnabledInstruments</code>	Returns the non-disabled instrument count. It can now also include non portfolio instruments that can trade using the broker objects. See Set Parameters script section reference.
<code>totalInstruments</code>	This system property returns the Total Number of instruments in the portfolio that will be used during a test period. It will not include instrument that do not have data record dates that are not in test period between Test Start-Date and the Test End-Date. This was done so difference between the number of instruments in the list of the selected portfolio and the those that Trading Blox Builder prevent the use of memory that will not be available during the planned test dates. It also improves the loading and speed of the test.
<code>totalLongPositions</code>	Number of instruments Long position in the system with a position size greater than zero.
<code>totalLongUnits</code>	Total units for LONG positions with a positions size greater than zero.
<code>totalMargin</code>	<p>This Property has already been depreciated.</p> <p>Value returned by this property is the same as <code>system.currentMargin</code>. In scripts where this was previously used change property to <code>system.currentMargin</code>. In new scripts only <code>system.currentMargin</code>.</p>
<code>totalOpenOrders</code> [Now Obsolete]	<p>Total number of open orders for the system that have not been filled or rejected. Property is updated when an order is not rejected and processing returns to the script where the Broker function was executed.</p> <p><code>system.TotalOrders</code> can provide the same information as this property. This new property name <code>system.totalOpenOrders</code> is kept for backward compatibility, but could be removed from the manual on a later date.</p>

Property Name:	Description:										
<code>totalPositions</code>	Number of instruments of LONG or SHORT position with a size greater than zero.										
<code>totalShortPositions</code>	Number of SHORT positions with a size greater than zero.										
<code>totalShortUnits</code>	Total units for SHORT positions with a position size that is greater than zero.										
<code>totalUnits</code>	Total number of Long and Short unit positions with a position size greater than zero.										
<code>tradingEquity</code>	<p>Amount of money available for trading by the current system.</p> <p>Trading equity is determined by the Global Parameters, Equity Manager's Trading Equity Base parameters:</p> <table border="1"> <thead> <tr> <th>Setting Description:</th> <th>Global Section:</th> </tr> </thead> <tbody> <tr> <td>System Allocation Slider Control</td> <td>System Allocation</td> </tr> <tr> <td>Leverage Contol</td> <td>Equity Manager</td> </tr> <tr> <td>Drawdown Reduction Threshold</td> <td>Equity Manager</td> </tr> <tr> <td>Drawdown Reduction Amount</td> <td>Equity Manager</td> </tr> </tbody> </table> <p>When Global Parameters, Equity Manager's parameter for Trading Equity Base is set to "Total Equity", the value of <code>system.tradingEquity</code> is equal to the <code>test.totalEquity</code> multiplied by the <code>test.leverage</code>. That resulting value is then multiplied by the <code>system.allocationPercent</code> rate.</p> <p>Global Parameters, Equity Manager's section parameter for Drawdown Reduction Threshold and Drawdown Reduction Amount settings will reduce the system's equity amount when these parameter settings are greater than zero.</p> <p>Any value greater than zero entered into the Global Parameters, Equity Manager section for Order Generation Amount will replaces the <code>test.totalEquity</code> value for order sizing.</p>	Setting Description:	Global Section:	System Allocation Slider Control	System Allocation	Leverage Contol	Equity Manager	Drawdown Reduction Threshold	Equity Manager	Drawdown Reduction Amount	Equity Manager
Setting Description:	Global Section:										
System Allocation Slider Control	System Allocation										
Leverage Contol	Equity Manager										
Drawdown Reduction Threshold	Equity Manager										
Drawdown Reduction Amount	Equity Manager										
<code>tradingInstruments</code>	Total number of portfolio instruments testing that are primed and have price information for the current test date.										

Links:

[System Functions](#)

See Also:

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 611

marketOrdersValue

This property returns the value amount required for all of the active open entry orders placed with the broker object and not canceled.

Syntax:

```
x = system.marketOrdersValue
```

Parameter:**Description:**

Parameter:	Description:
none	

Returns:

Amount required to fill all the active orders.

Notes:

A common problem with trading some systems that use less restrictive conditional rules to generate orders on lots of symbols is the overload of entry orders on the first day of order generation. A huge amount of orders means a run on available cash right at the beginning of trading, or at any time when a lot of instruments qualify with the rules for creating an order.

An example of a less restrictive rule is the use of greater than ">" and or less than "<", instead of a restrictive method like [CrossOver](#). When used to acknowledge that one price is above or below another price without some regard as to when that transition happened, means any qualifying relationship of the two or more prices will enable the instrument to generate an order.

Order generation without the concern for available Cash will cause the test results to be unrealistic. Test results will be unreliable because they will be based upon the system that exceeded available Cash, or exceeded Futures margin loading. Results based on any violation of account restrictions like excessive margin load percentages, which can also exceed available Cash, or the use of more cash than what is available is a test failure.

Value returned by this System property for stock systems will be the incremental amount that will be deducted from available Cash for stock type systems, or the incremental amount of margin used for Futures type systems.

Futures margin is based upon the margin value of each instrument in the Futures Dictionary.

This script example is for a Suite with one system.

If the Suite is using multiple systems where the early systems can consume an excessive amount available equity, a more inclusive script will be required so that the equity being used by each of the systems can be analyzed and adjusted as needed. By expanding the range of information

Notes:

need to consider the dynamic interactions between margin/purchase equity, total/closed equity, and cash need the process will able to further explored the accounts changes.

Example:

```

' -----
' FUTURES
' -----
If instrument.IsFuture THEN
' Create a Currency Converted value of this
' order's need for equity
newMarginValue = order.quantity _
                * instrument.margin _
                * instrument.conversionRate

If ( test.totalMargin _
    + system.marketOrdersValue _
    + newMarginValue ) > ( test.totalEquity _
                          * maxMarginEquity ) THEN
' Send the user a rejection message
message = "Cannot add " + AsString( newMarginValue, 2 ) _
        + " additional margin to " _
        + AsString( test.totalMargin, 2 ) _
        + " existing with total equity of " _
        + AsString( test.totalEquity, 2 ) _
        + " and max margin equity threshold of " _
        + AsString( maxMarginEquity, 2 )

' Reject This Order
order.Reject( message )

' Send the message to Log Window & Print Ouput.csv log
PRINT message
ENDIF ' > available equity
ENDIF ' i.IsFuture
system.marketOrdersValue
' -----
' STOCKS
' -----
If instrument.IsStock THEN
' Create a Currency Converted and Split adjust value
' of this order's need for equity
newMarginValue = order.quantity _
                * order.orderPrice _
                * instrument.conversionRate _
                * instrument.stockSplitRatio

If ( test.totalMargin _
    + newMarginValue _
    + system.openOrderMargin ) > ( test.totalEquity _
                                  * maxMarginEquity ) THEN
' Create a Currency Converted value of this
' order's need for equity
message = AsString( test.currentDate ) _
        + " Symbol " _
        + instrument.symbol _
        + " Cannot add " _
        + AsString( newMarginValue, 2 ) _
        + " additional purchase equity with cash of " _

```

Example:

```
+ AsString( test.cash, 2 ) _  
+ " max threshold of " _  
+ AsString( maxMarginEquity, 2 ) _  
+ " and open orders of " _  
+ AsString( system.openOrderMargin, 2 )  
' Reject This Order  
order.Reject( message )  
  
' Send the message to Log Window & Print Ouput.csv log  
PRINT message  
ENDIF ' > available equity  
ENDIF ' i.IsStock
```

Links:

[conversionRate](#), [margin](#), [orderPrice](#), [quantity](#), [Reject](#) [stockSplitRatio](#)

See Also:

totalOpenOrders

Total number of open orders not filled or rejected. This property is updated when an order is created, or an active order is rejected.

When called from a script section where an order is generated, this property will include the most recent order in its return value once the Broker function returns the processing to script section where the Broker function was executed.

Syntax:

```
x = system.totalOpenOrders
```

Parameter:**Description:**

<none>	Property keeps a count of all active and unfilled orders.
--------	---

Notes:

[system.totalOpenOrders](#) and [system.TotalOrders](#) can return the same value. They can also return different values when [system.TotalOrders](#)(symbol) is called with a symbol value in its optional parameter.

Understanding how the two above methods can be used is in [system.SetAlternateOrder](#) topic.

[system.TotalOrders](#) can provide the same information as this property. This [system.totalOpenOrders](#) property is kept for backward compatibility, but could be removed from the manual on a later date.

Example:

```
' Property value change can first be seen here:
' If Broker function creates an order...
If system.OrderExists() THEN
  ' Show order information
  PRINT block.scriptName, _
    test.currentDate, _
    system.totalOpenOrders
ENDIF ' system.OrderExists()
```

Results:

Before Orders Execution, 2012-05-31, 5

Links:

[Alternate Objects](#), [OrderFunctions](#), [Order Object](#), [OrderProperties](#), [SetAlternateOrder](#),

Links:

[System Properties](#), [TotalOrders](#)

See Also:

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 625

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.
Global Properties	Global Parameter setting 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.
String Arrays	Test level global string array functions and properties.
Test Statistics	After test level statistic & summary reporting functions.
Trade Properties	Test level closed trade details for all the instruments in the test.

Links:

See Also:

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 105

11.1 Equity Properties

Test Object properties use equity values from every system in the test.

A test can run multiple systems at once by adding the additional systems to the Suite selections.

Many equity properties are also index-able - if you put an offset 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.

Equity Properties:	Property Descriptions:
Capital Adds Draws Total	<p>Returns the total capital Adds and Draws that have been applied to the account. When this property is accessed in the After Trading Day script, the amount returned is the current summation value as of the current test date. When it is accessed before the After Trading Day script section, the value returned is the summation of the previous test date.</p> <p>As values are added or removed, the Capital Adds Draws file is updated. This File is located in the Trading Blox's installation Data/Auxiliary folder, and the file is named "Suite-Name Capital Adds Draws.csv".</p> <p>Only updated value changes in a test file are included when the sequence reaches the After Trading Day script section in the test simulation. The amount of change in equity that is applied is the amount of change in closed equity.</p> <p>See test.otherExpenses & test.UpdateOtherExpenses .</p>
cash	<p>Cash is the most current cash amount available. It is updated at the end of each test date. The update happens in the After Trading Day script section. This script is used because a test can contain systems that are using Stocks, Futures, and or Forex instruments, this property requires the current close price of all the instruments active in the test.</p> <ul style="list-style-type: none"> • Values returned are closed equity, plus marked to market open equity minus purchase equity. • Marked to market open equity is the Futures instruments open equity, and purchase equity is the total initial purchase equity required to purchase open Stock instruments. • For a futures only test this returns total equity. • For a stock only test this returns closed equity minus purchase equity.
closedEquity[]	<p>Total Closed Equity for all systems.</p>
closedEquityHigh	<p>Current high water mark for closed equity. Used to compute the this test property: test.currentClosedDrawdown .</p>

Equity Properties:	Property Descriptions:
<code>coreEquity</code>	<p>Core Equity is updated for each <code>test.currentDay</code> location of a test (not indexed).</p> <p>Core Equity Calculations are a measure of equity that includes <i>Closed Equity</i> plus that portion of <i>Open Equity</i> that would be realized if all the current positions were exited at their current stop values (i.e. <i>Locked-In Profits</i>).</p> $\text{Core Equity} = \text{Closed Equity} + \text{Locked-In Profits}$
<code>currentClosedDrawdown</code>	Current bar's closed equity drawdown.
<code>currentClosedEquity</code>	This property value is dynamic and it will change as the test works towards completion.
<code>currentDrawdown []</code>	Percent of total equity drawdown. Graphed as the "Drawdown" graph under test results.
<code>currentMargin</code>	Deprecated -- see totalMargin
<code>currentOpenEquity</code>	This property value is dynamic and it will change as the test works towards completion.
<code>currentRisk []</code>	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 ".
<code>currentTotalEquity</code>	This property value is dynamic and it will change as the test works towards completion.
<code>otherExpenses</code>	Total other expenses as set by test.UpdateOtherExpenses .
<code>purchaseEquity</code>	Use this property with stock portfolios. This property will return the current test simulation's Initial equity required amount to purchase current open positions (entry fill price times the quantity of shares) that are active in the test.
<code>smartFillExit</code>	Returns the smartFillExit property as set in global parameters. True or false.
<code>startingEquity</code>	Equity as of the start of the simulation. Equal to the Global Parameters, Equity Manager's Test Starting Equity .
<code>threadCount</code>	Returns the threadCount property as set in global parameters. Value can be a minimum of 1 to the maximum number of threads allowed by the license.
<code>totalEquity []</code>	Starts as the Test Starting Equity as specified in Global Parameters. It is then affected by the total profits, losses of trades by all systems as well as margin costs and interest earned,. Graphed as the " Equity Curve " graph. For order generation this is the Order Generation Equity.

Equity Properties:	Property Descriptions:
	<p>Note: This property is not affected by any of the following Global Parameters settings:</p> <p>System Allocation Slider Draw down Reduction Threshold, or its Amount, Choice of Base Equity selection: Total Equity or Closed Equity.</p>
totalEquityHigh	Current high water mark for test.totalEquity . This value is used to compute the value of test.currentDrawdown .
totalMargin	<p>Total current margin of all open positions in the test. For Futures test.totalMargin represents the margin value of all open Future position margin values. When the instruments are for a Stock type test.totalMargin represents the initial purchase equity.</p> <p>In a Suite where both Stock and Future positions are active, this property's value represents the combined sum of Futures Margin amounts and the sum Stock "Purchase Equity" of all active positions in the Suite.</p> <p>Same as totalMargin, currentMargin, purchaseEquity. These all return the sum of all futures margin and all stock purchase equity.</p>
totalOpenPosition Trades	For order generation, the number of open positions at the end of the test.
Vadi	<p>VADI is the acronym for "Value Added Daily Index". VADI calculations start at the start of trading equity. Values increase and decrease as a ratio of profit/loss of trading equity percent. VADI is net of capital adds and draws, and it includes accrued fees, whereas trading equity only includes booked fees and the current sum of test.otherExpenses.</p>
vadiDrawdown	Current VADI drawdown.

NOTE:

The most current equity numbers are from the prior test date. It works this way when the test date is one trade day ahead of the instrument date the value for equity is based on what happened after all positions were settle on the previous test date. It is important to know that all equity transactions are settle at the end of the test date. At the equity settle location the test date and the instruments that had a trade record for that date will have the same date value when equity settles are made.

Equity va, 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.

Links:
UpdateOtherExpenses
See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 305

Capital Adds Draws Total

Capital Adds & Draws with Total:

This Trading Blox Builder feature provides a trader an option to add or draw capital from a trading account.

Values for an Add, or a Draw can be created by keyboard entry where the user enters a transaction date and an amount that adds or draws down the value.

Values entered are recorded on the test date that will match a simulation test date where the value change can be applied as one of the transaction during the **After Trade Day** script section equity adjustments.

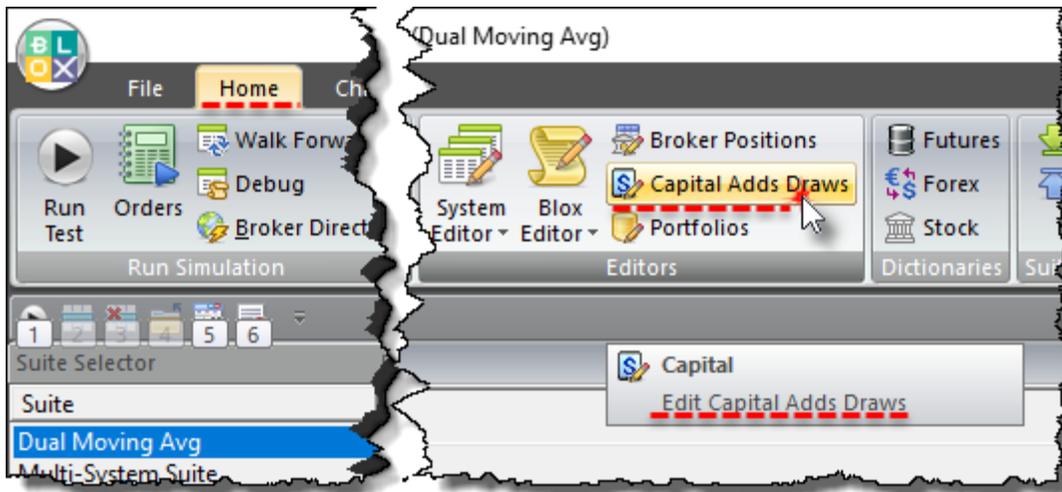
Global Parameter's Fees Parameters can be enabled so the expected fee amounts are entered as reduction records in the **Capital Adds Draws Transaction** file (See User's Guide [Fee Accruals](#) topic for details).

As capital record transactions are recorded, and each transaction record date passes the value of the summed transaction to that date are applied, and also can be reported.

Each time a capital value are added or removed the [Capital Adds Draws](#) file is updated.

Capital Additions & Draws Dialog Access:

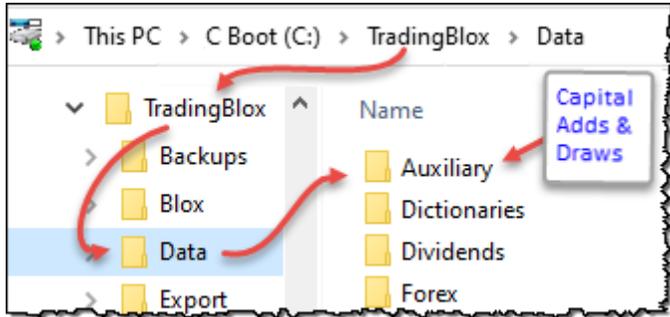
Values are entered through a dialog that is available at the bottom of Trading Blox main screen Suite menu item:



Capital Adds & Draws Menu Access Button

Capital Adds & Draws File:

The file must be created by the user and placed in the Auxiliary Folder that is in the Data folder that is in the Trading Blox installation folder:

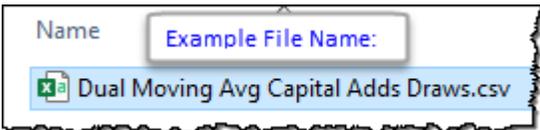


Place the Capital Adds & Draws file you create in this Trading Blox installation Folder.

Reason	Date	Amount	Comment
Details:	YYYYMMDD	Use a "-" sign for a Draw, Use a "+" sign for an Add	Enter the Transaction Reason

All four Columns must have a comma separator.

Text Layout Detail Requirements



Files with Comma separations will load into a spreadsheet program like Excel, and many others when they use the ".CSV" suffix

```

1 | Version,7
2 | Date,Amount,Comment
3 | 20090105,-2500,"1st Transaction"
4 | 20100105,-2500,"2nd Transaction"
5 | 20110105,2500,"3rd Transaction"
6 | 20120104,4500,"5th Transaction"
7 | 20120105,-2500,"4th Transaction"
8 | 20170530,50000,"5th Transaction"
9 |

```

Capital Adds & Draws File Example in a comma separated group of columns

Adding, Changing and Removing Transactions:

When the **Capital Adds Draws** button is clicked, this dialog will appear:

Capital Adds Draws

Date	Amount	Comments
20090105	-2500	Cost type 1
20100105	-2500	Cost type 1
20110105	3500	Account Transfer
20110105	-4500	Management Fee
20120515	-1500	Cost type 2

Records added earlier

Record being added

OK

Cancel

New Add Draw

Delete Add Draw

Accept edits, closes editor

Cancel edits, closes editor

Adds Record

Deletes Record

Date (YYYYMMDD) 20120515

Amount (+ add, - draw) -1500

Comments Cost type 2

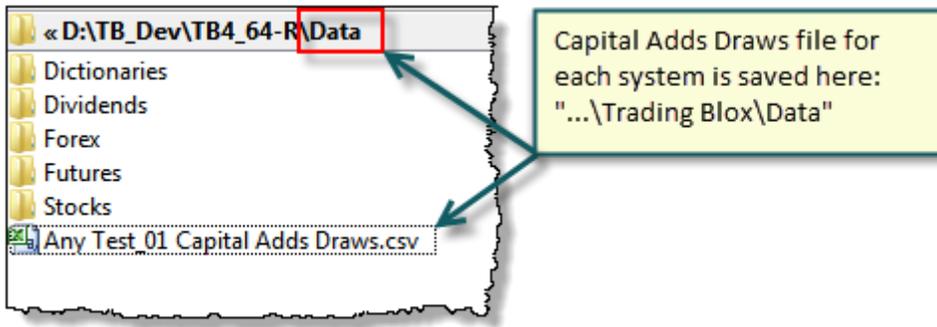
Tuesday

Capital Adds & Draws File Location

Changes made to the Capital Adds Draws process are updated and available.

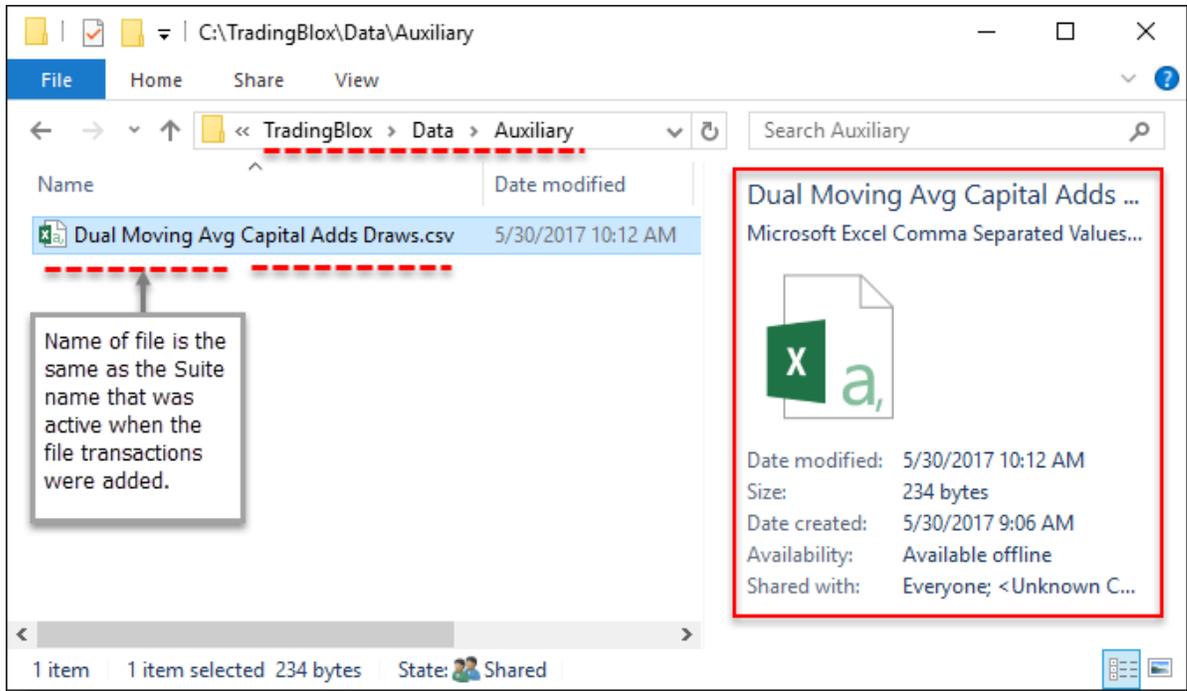
Capital Draw Add File Contents:

This file is located where Trading Blox Builder is installed.



Trading Blox Folder Location

To access the file, OPEN the Trading Blox Builder folder, then open the **Data** folder's **Auxiliary** folder:



Capital Adds & Draws File Location

File is a comma delimited file compatible with most spreadsheet programs.

Capital Add Draw records for each system record is listed by date and amount:

Version	Amount
20090105	-2500
20100105	-2500
20110105	2500
20120105	-2500
20130104	4500
20170530	50000

Above Sample File Contents

Syntax:	
<pre>' Obtain Current Value of recorded Adds & Draws value = test.capitalAddsDrawsTotal</pre>	
Parameter:	Description:
<none>	This Test-Object property provides the cumulative value of the entries in the Capital Add & Draws entered records.

Example:	
<pre>' Access current running total of Capital Adds & Draws value = test.capitalAddsDrawsTotal ' Send capitalAddsDrawsTotal property value to Print Output.csv file. Print "Capital Adds & Draws: ", value</pre>	
Returns:	
Capital Adds & Draws: , 49,500 (Amount is the Summed value of all entries in example table above.)	

Links:
Fee Accruals , Script Adds & Draw Option , UpdateOtherExpenses , Vadi
See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 186

Fee Accruals

Note:

Fee transactions are accrued on a (daily/monthly/quarterly/yearly) basis, and booked as defined in global parameters .

Click to Expand -- Click to Reduce

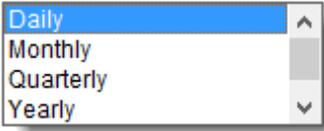
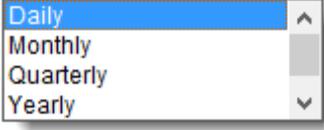
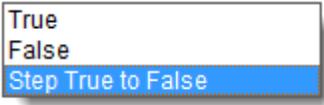
Global Parameters - Suite Fee Parameter Options (User's Guide Help).

When the Use [Global Adds Draws](#) option is set to **TRUE**, fee amounts will be recorded in the **Capital Adds Draws** Transaction file.

Global Parameter Fees will create a fee transactions based upon the **Charge Management Fee** selected setting. The amount of a fee event is applied at the end of a trade day off the current selected test date period. Fee transactions are applied as one of the last **After Trade Day** equity adjustments.

Each fee type can have a different accrual period. As fees accrue, they are booked as draws from the account equity.

Capital Adds & Draws transactions can be included in the transaction basis.

Parameter:	Description:	
Management Fee (Annual % of Equity) ^Top	This fee is a percentage calculation. It is the cost of inclusion into the trading management structure.	
Charge Management Fee ^Top	 <p>Management Fee Timing Options</p>	Management fees accrual timing is determined by this selected value.
Incentive Fee (% of Net New Profits) ^Top	Enter an incentive percentage rate to apply to the added gain amount during each of the gain periods.	
Charge Incentive Fee ^Top	 <p>Incentive Fee Timing Options</p>	Incentive fees are determined by the user's selected accrual period.
Use Capital Adds Draws ^Top	 <p>Fee Accrual Capital Adds & Draws Update Option</p>	<p>When FALSE, none of the fee transaction will not be recorded in the Capital Adds & Draws transaction records.</p> <p>When set to TRUE, the fee accrual transactions will each become a record a recorded item the Capital Adds & Draws records.</p>

Links:

[Fee Accruals](#), [Script Adds & Draw Option](#), [UpdateOtherExpenses](#), [Vadi Capital Adds Draws](#), [Fee Accruals](#), [OtherExpense](#), [Script Adds & Draw Option](#), [UpdateOtherExpenses](#)

Links:

See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 89

Scripting Add-Draws

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:	Description:
expenseAdjustment	<p>Amount to add, or subtract from the other expenses category.</p> <ul style="list-style-type: none"> • Positive value adjustments increase the expense amount of the Other Expense sub-account total. • Negative value adjustments reduce the expense amount of the Other Expense sub-account total.

Returns:

See example.

Example:

```
' Negative value adjustments reduce the Other Expense total.
test.UpdateOtherExpenses( -25000 )

' Positive value adjustments increases the Other Expense total.
test.UpdateOtherExpenses( 25000 )

' 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:

[CapitalAddsDrawsTotal](#), [OtherExpenses](#)

See Also:

VADI

VADI is the acronym for "Value Added Daily Index".

VADI calculations start at the start of trading equity. Values increase and decrease as a ratio of profit/loss of trading equity percent. VADI is net of capital adds and draws, and it includes accrued fees, whereas trading equity only includes booked fees.

Syntax:

```
value = test.vadi
```

Parameter:

<none> No parameters.

Example:

```
' Show current VADI property value
Print test.currentDate, test.vadi
```

Returns:

Print Output file will show the test date and the value of the VADI property.

Note:

Fee transactions are accrued on a (daily/monthly/quarterly/yearly) basis, and booked as defined in global parameters .

Fees Parameters

Management Fee (Annual % of Equity) Step 0%

Charge Management Fee Daily

Incentive Fee (% of Net New Profits) Step 0%

Charge Incentive Fee Daily

Use Capital Adds Draws False

Daily
Monthly
Quarterly
Yearly
True
False
Step True to False

Click to Expand -- Click to Reduce

Global Parameters - Suite Fee Parameter Options (User's Guide Help).

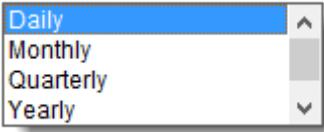
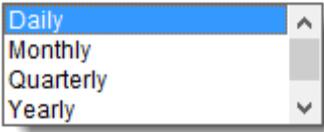
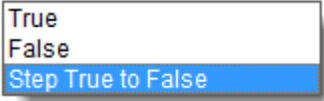
When the Use [Global Adds Draws](#) option is set to **TRUE**, fee amounts will be recorded in the **Capital Adds Draws** Transaction file.

Global Parameter Fees will create a fee transactions based upon the **Charge Management Fee** selected setting. The amount of a fee event is applied at the end of a trade day off the current selected test date period. Fee transactions are applied as one of the last **After Trade Day** equity

adjustments.

Each fee type can have a different accrual period. As fees accrue, they are booked as draws from the account equity.

Capital Adds & Draws transactions can be included in the transaction basis.

Parameter:	Description:	
Management Fee (Annual % of Equity) ^Top	This fee is a percentage calculation. It is the cost of inclusion into the trading management structure.	
Charge Management Fee ^Top	 Management Fee Timing Options	Management fees accrual timing is determined by this selected value.
Incentive Fee (% of Net New Profits) ^Top	Enter an incentive percentage rate to apply to the added gain amount during each of the gain periods.	
Charge Incentive Fee ^Top	 Incentive Fee Timing Options	Incentive fees are determined by the user's selected accrual period.
Use Capital Adds Draws ^Top	 Fee Accrual Capital Adds & Draws Update Option	When FALSE , none of the fee transaction will not be recorded in the Capital Adds & Draws transaction records. When set to TRUE , the fee accrual transactions will each become a record a recorded item the Capital Adds & Draws records.

Links:

[Fee Accruals](#), [Script Adds & Draw Option](#), [UpdateOtherExpenses](#), [Vadi Capital Adds Draws](#), [Fee Accruals](#), [OtherExpense](#), [Script Adds & Draw Option](#), [UpdateOtherExpenses](#)

See Also:

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:
<pre>PRINT "Test Name: ", test.name PRINT "Order Report Path: ", test.orderReportPath</pre>

Properties:	Description:
<code>abortTestPending</code>	Returns true when the Abort Test or Abort Simulation functions have been used.
<code>baseCurrency</code>	Returns the ISO code of the system wide base currency set by the user in the Preference section. i.e. USD is used for United States Currency.
<code>baseCurrencyBorrowRate</code>	Borrow rate of the system wide base currency
<code>baseCurrencyLendRate</code>	Lend rate of the system wide base currency
<code>baseCurrencySign</code>	Returns the ISO currency sign symbol for the system wide base currency set by the user in the Preference section. i.e. USD uses \$
<code>capitalAddsDrawsTotal</code>	Total capital adds and draws to date, from the capital adds draws file.
<code>currentDate</code>	Current simulation date. In YYYYMMDD format.
<code>currentDay</code>	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.
<code>currentParameterTest</code>	Number of the current active parameter step test
<code>currentTime</code>	Current simulation time. In HHMM format.
<code>dayEndTime</code>	For intraday data, last testing time of the day. Computed as the latest time for any market in the test, or as set in scripting. See test.SetEarliestTime and test.SetLatestTime and test.SetTimeIncrement .
<code>feesIncentiveAccrued</code>	Incentive fees accrued but not yet booked. See: Fee Accruals & Capital Adds & Draws
<code>feesIncentiveTotal</code>	Total incentive fees booked to date.
<code>feesManagementAccrued</code>	Management fees accrued but not yet booked.
<code>feesManagementTotal</code>	Total management fees booked to date.

Properties:	Description:
forexDataFolder	
forexDataPath	Provides the full path to the Forex Files. Default Forex path: C:\Trading Blox\Data\Forex\
futuresDataFolder	
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 properties value will include 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.
instrumentListRef	<p>This property provides a faster access to instrument information that using an instrument property. It is faster because it is a memory pointer. Pointers are used a lot in the best program so the software's design allows the program's options to work as fast as possible.</p> <p>The various "ref" properties are object pointers to the actual instrument. Makes them very fast.</p> <p>You can use test.instrumentList[x] to loop over the instruments in the test, but that returns a symbol, and TB then needs to lookup that symbol to find the instrument. Whereas the test.instrumentListRef[x] is an array of object pointers - immediate access to the instrument when used in the LoadSymbol.</p>
leverage	Leverage as set in Global Parameters
marketOrdersValue	Total amount needed to fill all current open orders in all the systems in the suite. For Futures, this is the margin times order size, for Stocks, this is the order price times the order size. Better details available in the system.marketOrdersValue description topic.
maxUnitsSuiteParameter	This property return the max units number that is set in the Simulation's "Max Position Units" parameter setting.
maxUnitsSuiteParameter	
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. Useful with with intraday data where you can

Properties:	Description:
	check the <code>test.orderGenerationBar</code> return value. If its return is TRUE you can output all the relevant data for the last bar.
<code>orderGenerationTest</code>	Returns true if the test is generating orders, rather than just running a performance test.
<u>orderReportPath</u>	Full path of the current order report
<code>primeStart</code>	Earliest date for all loaded data for any instrument
<u>resultsReportPath</u>	Path of the results folder for this test. Used to access charts and graphs in the test results folder for display.
<code>sellStockSplitRemainder</code>	Global parameter setting value of the Sell Stock Split Remainder value. Returns a True or False value.
<code>SetTimeIncrement</code>	<p>You can force the minimum time increment used for testing by changing from the default value.</p> <p>In the Before Simulation Script, use the function to set HHMM time to use:</p> <pre>test.SetTimeIncrement(1) ' This sets the time to 1 minute test.SetTimeIncrement(130) ' This sets the time to 90 minutes</pre> <p>Or, set to your desired bar size, such as 5 minute or 60 minute. The test will run for each bar, and the last available instrument bar will be used for order generation.</p>
<u>smartFillExit</u>	Returns a True or False value of the <u>Global Parameter</u> option "Smart Fill Exit" setting.
<code>stockDataFolder</code>	
<code>stockDataPath</code>	Provides the full path to the Stock Files. Default Forex path: C:\Trading Blox\Data\Stocks\
<u>summaryResultsPath</u>	<p>Full path of the current test results saved file location. file.</p> <p>Example:</p> <pre>' Script sends Test Performance Folder name to Print Output.csv file. Print test.summaryResultsPath</pre> <p>Return Example:</p> <pre>Test 2016-04-28_09_08_04</pre>
<code>systemCount</code>	Number of systems to be tested.
<code>testEnd</code>	Test End date. The user entered end date, or the end of data, which ever comes first.

Properties:	Description:
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. Stepped Parameter information can be accessed using the test.GetSteppedParameter
totalZeroSizedTrades	The number of trades in all of the systems that had a trade quantity of zero. Useful for custom statistics to include or exclude these placeholder trades.
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.

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.

Links:

See Also:

instrumentListRef

This property provides a faster access to instrument information than using an instrument property. It is faster because it is a memory pointer. Pointers are used a lot in the best program so the software's design allows the program's options to work as fast as possible.

The various "ref" properties are object pointers to the actual instrument. Makes them very fast.

You can use `test.instrumentList[x]` to loop over the instruments in the test, but that returns a symbol, and TB then needs to lookup that symbol to find the instrument. Whereas the `test.instrumentListRef[x]` is an array of object pointers - immediate access to the instrument when used in the LoadSymbol.

Syntax:

```
' To access aportfolio instrument
test.instrumentList[x]
```

Parameter:	Description:

Returns:

Example:

Links:

See Also:

Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 82

OrderReportPath

When the Orders button is being used to generate orders, this property will return the full path and folder name of the location where the orders report is located.

Note:

Use this property to obtain the order report saving location when generating orders. When orders are not being generated, the return from this property will be blank.

Syntax:
<code>CurrentOrderFolderPath = test.orderReportPath</code>

Parameter:	Description:
<None>	Output is a String containing the current Trading Blox path for saving order files.

Example:
<pre>' Display Order Report Path information when generating orders. PRINT "test.orderReportPath:" PRINT test.orderReportPath</pre>
Returns:
<code>test.orderReportPath</code> <code>C:\TradingBlox\Orders\Orders 2017-05-31_11_33_42.html</code>

Links:
See Also:
ForexDataPath , FuturesDataPath , ResultsReportPath , StockDataPath , SummaryResultsPath

ResultsReportPath

Property returns the full path and folder name of the current test Suite.

Note:

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.

Syntax:
CurrentTestResultsFolderPath = <code>test.resultsReportPath</code>

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>

Example:
<pre>' Assign results reporting folder path and name to a variable. CurrentTestResultsFolderPath = test.resultsReportPath OR ' Send to your PRINT OUTPUT the path and folder name PRINT "test.resultsReportPath" Print test.resultsReportPath</pre>
Returns:
<p>Printed output would look something like the following:</p> <pre>test.resultsReportPath C:\TradingBlox\Results\Test 2017-05-31_10_11_29</pre>

Links:
Print
See Also:
ForexDataPath , FuturesDataPath , OrderReportPath , StockDataPath , SummaryResultsPath

SummaryResultsPath

Property returns the full path and folder name of the current test Suite.

Note:

Use this property to discover the current test results file saving location.

Syntax:

```
CurrentOrderFolderPath = test.summaryResultsPath
```

Parameter:**Description:**

<None>

Output is a [String](#) containing the current Trading Blox path for saving the current test result files.

Example:

```
' Display Summary Results Path
PRINT "test.summaryResultsPath"
PRINT test.summaryResultsPath
```

Returns:

Printed output would look something like the following:

```
test.summaryResultsPath
Test 2017-05-31_10_11_29
```

Links:**See Also:**

[ForexDataPath](#), [FuturesDataPath](#), [OrderReportPath](#), [ResultsReportPath](#), [StockDataPath](#)

11.3 Global Parameter Properties

Global Parameters are [Test](#) object properties will provide the current setting used. Each test properties listed along side each of the Global Parameter settings will display the current setting that Global Parameter is using when the assigned test property is accessed in scripting.

To use these parameters add the prefix: `test.<propertyName>`
`Print "Earn Interest Setting: ", test.earnInterest`

System Allocation Parameters:

Setting Description:	System Property:	Settings:	Notes:
 <p style="text-align: center;">Global System Allocation Slider Control</p>	allocationPercent	0 to 100%	Each system in a suite will have its name above one of the Allocation Control Sliders.

Global Simulation Parameters:

Setting Description:	Test Property:	Settings:	Notes:
Earn Interest	earnInterest	True/False	
Entry Day Retracement (%)	entrydayRetracement	0.00	
Global parameter test end date value.	testEndOrig		
Ignore Test Positions	ignoreTestPositions	True/False	Use Broker = TRUE

Setting Description:	Test Property:	Settings:	Notes:
Increment Test Start (days) (Removed)	incrementTestStart	0	Use Start Date Stepping = TRUE
Max Margin/Equity to Trade (%)	maxMarginEquityPercent	0.00	Zero disables this filter.
Max Percent Volume Per Trade (%)	maxVolumePerTrade	0.00	Zero disables this filter.
maxUnitsSuiteParameter	maxUnitsSuiteParameter		Returns the max units allowed.
Minimum Slippage (4)	minimumSlippage	\$0.00	
SetMarginEquity	SetMarginEquity		Sets the Suite's Margin Parameter.
Set Test Duration (days) (Removed)	setTestDuration	0 Days	Use Start Date Stepping = TRUE
Slippage (%)	slippagePercent	0.00	
Smart Fill Exit	smartFillExit	True/False	test.SetSmartFillExit() function can enable or disable this Global setting state.
Thread Count	threadCount	0	
Trade Always on Tick	tradeOnTick	True/False	
Use Broker Positions	useBrokerPositions	True/False	
Use Start Date Stepping (Removed)	useStartDateStepping	True/False	

Walk Forward Parameters:

Setting Description:	Test Property:	Settings:	Notes:
Walk Forward Optimization Days		0	
Walk Forward OSS Days		0	

Futures Parameters:

Setting Description:	Test Property:	Settings:	Notes:
Minimum Futures Volume (contracts)	minimumFuturesVolume	0	

Setting Description:	Test Property:	Settings:	Notes:
Commission per Contract	commissionPerContract	0	
Trade Futures on Lock Day	tradeLockLimit	True/False	
Account for Contract Rolls	accountForContractRolls	True/False	
Roll Slippage (% of ATR)	rollSlippageATR	0.00	Account for Contract Rolls = TRUE

Stocks Parameters:

Setting Description:	Test Property:	Settings:	Notes:
Minimum Stock Volume (shares)	minimumStockVolume	0	
Commission per Stock Trade	commissionPerTrade	0	
Commission per Stock Share	commissionPerShare	0	
Commission per Stock Value (%)	commissionPercentValue	0.00	
Sell Stock Split Remainder		True/False	
Earn Dividends	earnDividends	True/False	
Pay Margin on Stocks	payMargin	True/False	

Forex Parameters:

Setting Description:	Test Property:	Settings:	Notes:
Forex Trade Size (in base currency)	forexTradeSize	0	
Account for Forex Carry	accountForForexCarry	True/False	
Use Pip-Based Slippage	usePipBasedSlippage	True/False	

Fees Parameters:

Setting Description:	Test Property:	Settings:	Notes:
Management Fee (Annual % of Equity)		0.00	
Charge Management Fee		Mth/Qtr /Ann.	
Incentive Fee (% of Net New Profits)		0.00	
Charge Incentive Fee		Mth/Qtr /Ann.	
Use Capital Adds Draws		True/Fa lse	

Equity Parameters:

Setting Description:	Test Property:	Settings:	Notes:
Test Starting Equity	startingEquity	\$0.00	
Order Generation Equity		\$0.00	
Order Generation Equity High		\$0.00	
Leverage (fraction)	leverage	1.0	
Trading Equity Base		Total/C losed	
Drawdown Reduction Threshold (%)		0.00	
Drawdown Reduction Amount (%)		0.00	

Links:

See Also:

See User's Guide Global Parameters Topic

11.4 Functions

Test level properties and functions provide access to paths, and can control how test are executed or terminated. They can also report features that are changed or added.

All Test Object properties are used with the "`test.`" object prefix.

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
GetStatisticValue (statisticName)	This function to return the value of any statistic available in the search " SetTotalParameterRuns " (1 hit in 1 file)
GetSteppedParameter (paramIndex , paramValueIndex)	Returns the attributes of each stepped parameter.
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.
SetAutoPriming (True/False)	Use in Before Simulation Script to disable auto priming.
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.
SetExitOpenSimulationPositions	This function will adjust and keep all the selected positions open for a simulation, just like what happens in order generation.
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.
SetGoodnessForWalkForward	Use this in the Before Test script to set the Goodness value used in Large Scale Optimization test.

Test Function Name:	Description:
SetGoodnessToChart (StatList)	Creates multi parameter charts (contour and 3D) for all status in the list, such as ("Mar, Sharpe")
SetIBPositionSynchOrderType	This function sets the order type used for an IB position synch process.
SetIBTransmit (true/false)	Use this function to globally set the transmit flag for all IB orders, including synch and roll orders. Can be overridden by the <code>order.transmit</code> flag.
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
SetMarginEquity()	This function sets the “ Max Margin/Equity to trade (%) ” suite parameter.
SetMaxUnits()	Use this function to set the max units allowed in a normal parameter test.
SetSilentTestRun (True/False)	Used in the Before Test Script to suppress any output from the test run.
SetSmartFillExit (True/False)	Script control to sets or change the Global Smart Fill Exit parameter to True or False. Useful when the system has multiple exits order placed for the same price bar. Functionality , 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.
SetTotalParameterRuns ()	This function is for use in the Before Simulation script. When it is executed, it will override the total parameter runs for the simulation. It Established the max number of parameter runs (steps) for the simulation.
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.

Test Function Name:	Description:
SetTotalParameterRuns()	This function is used generally in the Before Simulation script. When it is used and has a value that is less than the number of test stepped parameters, the number of total parameter runs for the simulation would be limited to the number used with this function.
<code>SortInstruments()</code>	Sorts the test level instrument list. The test level instrument list is all the instruments required for all systems in the test. See test.instrumentCount and test.instrumentList properties.
<code>UpdateCapital()</code>	Use to add/draw from test capital. Use a "+" sign to add expenses, use a "-" sign to remove expenses.
UpdateOtherExpenses()	Adds or subtracts other expenses. Use a "+" sign to add expenses, use a "-" sign to remove expenses.

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 422

AbortSimulation

Stops the entire simulation by aborting all parameter testing.

Syntax:
<code>test.AbortSimulation([message])</code>

Parameter:	Description:
[message]	Information about the reason why the test was aborted. Message information is a scripted statement created at the time this function is added to a blox.

Example:
<pre>' Abort the simulation because we could not load our data. test.AbortSimulation("Test Completed")</pre>
Returns:
<none>

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:**Description:**

```
<none>
```

Example:

```
' Abort the test because of invalid parameters  
test.AbortTest
```

Returns:

```
<none>
```

Links:**See Also:**

AddStatistic

You can add custom statistics to the **Summary Results** test report.

Best if used in the After Test script section and after the statistic values have been updated.

When called, this function will add one or more new user defined statistics to the **Custom Statistics** section. Placements in the **Summary Results** report is towards the end the report, and just above any **System Parameter Settings**.

User added **Custom Statistics** can be sorted.

Syntax:	
<code>test.AddStatistic(statisticName, value, [decimal places], [type])</code>	

Parameter:	Description:
<code>statisticName</code>	String that names the added statistic
<code>value</code>	Value of that statistic
<code>[decimal places]</code>	Optional number of decimal places for display of float numbers. Default is 2 if left out
<code>[type]</code>	Places is still optional. Float is default, although if value is a string the String type is assumed.

Type Options:	Description:
"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
2.12
test.AddStatistic( "My Custom Stat", 2.12345, 3 )     ' Output is
2.123
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
"2.12%"
test.AddStatistic( "Return", 2.12345, 0, "Percent" ) ' Output is
"2%"
test.AddStatistic( "Worst Date", 20010521, "Date" )  ' Output is
"2001-05-21"
test.AddStatistic( "Highest Value", 500000, "Currency" ) ' Output is
"$500,000"
```

Links:**See Also:**

GetStatisticValue

There are many statistics listed in the [Test Statistics](#) topic that return a value. This function uses the name of the statistic to return its value.

Syntax:

```
test.GetStatisticValue( statisticName )
```

Parameter

:

Description:statistic
Name

Name of the statistic.

Example:**Returns:****Links:****See Also:**

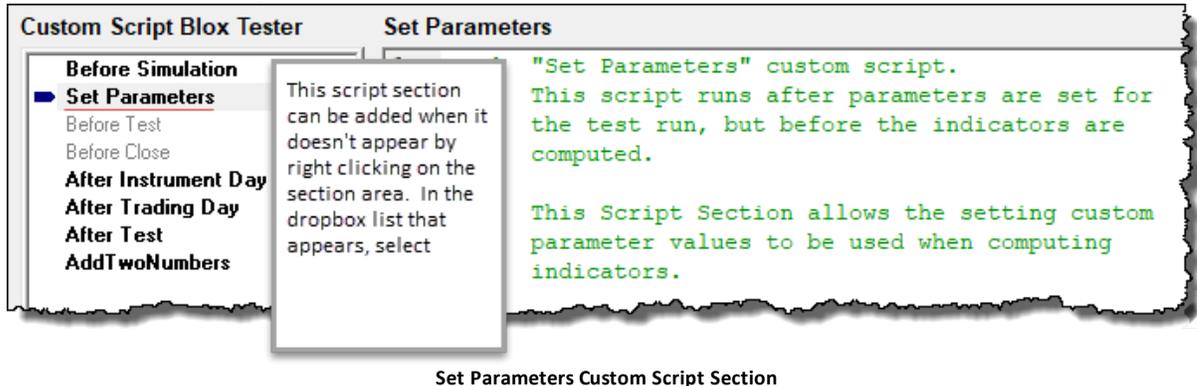
Edit Time: 3/21/2024 10:41:38 AM

Topic ID#: 69

GetSteppedParameter

When a test is running, this function will returns all the attributes of each stepped parameter.

This function is only used in the [Set Parameters](#) script.



To obtain information from each of the stepped parameters, or use the function to assign a value, use this function with an index value, and `paramValueRef`:

Enumeration Value Syntax:

```
test.GetSteppedParameter(steppedParameterIndex, paramValueRef, [value])
```

Parameter:	Description:
<code>paramIndex</code>	Each Stepping Parameter Index is based on its assigned stepping priority when the parameter was created. It is usually used in a look after the number of stepped parameters are known.
<code>paramValueRef</code>	The Parameter Value References are shown in the table below.
<code>[value]</code>	??

This table was created from what had worked before, and what I thought might be the current process to identify the parameter's type:

GetSteppedParameter Reference Table:

Parameter Stepped Index:	Parameter Option:	Enumeration:	Item Type:
0	Stepped Test Count	0	0
1	name	1	Unknown
2	start value	2	Float
3	end value	3	Integer

Parameter Stepped Index:	Parameter Option:	Enumeration:	Item Type:
4	step value	4	Percent
5	step count	5	Dollars
6	current step	6	Cents
7	current value	7	Boolean
8	stepping priority	8	Selector
9	type	9	Set
10	set value	10	String
11			Date

Use the function to get the count the number of parameters being stepped:

Count Parameter Uses this Syntax " 0 " value in the parameter locations:

```
steppingCount = test.GetSteppedParameter( 0 )
```

This is my best guess as to how the might be sourced:

ENUM value is returned when the first parameter is an index that identifies a stepping parameter's index location, and the second represents the information request:

Enumeration Value Syntax:

```
test.GetSteppedParameter( paramIndex, paramValueIndex, itemType )
```

Current Example in the TBB Help:

Example:

NOTE:

`test.GetSteppedParameter(index,6)` to return a 1 based index value, rather than 0 based.
`test.GetSteppedParameter(index, 10, value)` which sets the value of the parameter.
`GetSteppedParameter` returns a string, so convert to number if necessary.

```
steppedParameterCount = test.GetSteppedParameter( 0, 0 )
```

```
PRINT "Using the following stepped parameters."
```

```
PRINT "Name", _
      "Step Start", _
      "Step
      "Step End", _
      "Step Step", _
```

Example:

```
"Step Count", _
"Step Index", _
"Step Value", _
"Step Priority"

FOR i = 0 to steppedParameterCount
  ' Stepped parameter values
  testSteps = test.GetSteppedParameter( i, 0 )
  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 testSteps, _
        stepName, _
        stepStart, _
        stepEnd, _
        stepStep, _
        stepCount, _
        stepIndex, _
        stepValue, _
        stepPriority
NEXT
```

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.

Links:

Was: Before Test, NOW: [Set Parameters](#)

See Also:

SetAlternateSystem

Companion function to the software's System Object. It is used to allow access to scripts and values in systems other than the system in which this function is being executed. Most often this function is used in a Suite where a GSS system (a system with the same name as the Suite name) that is intended to access information in other systems in the Suite.

It gets access to the information in other systems in the Suite when it is called using the index of the system where it wants to access information. It gets access to information by bringing that system into context so the system executing this function can access information using the "alternateSystem" object prefix. When a system is brought into context any of the functions or properties of that system become available from within a system where SetAlternateSystem is being used. Property and function 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:	Description:
systemIndex	<p>System index number. When this function is executed it sets the special built-in object "alternateSystem" with a new system by its System Index.</p> <p>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.</p> <p>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.</p>

Example - Alternate System Access:

```
' 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
```

Return - Alternate System Access:

Out will display the system's suite index, its system-name, and its total-equity amount.

Example - Alternate system with the alternateBroker:

```
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
```

Return - Alternate system with the alternateBroker:

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.

function to access instruments in other systems in the suite:

Example - Alternate system with the LoadSymbol:

```

' =====
' _Alternate System Access Example
' AFTER TRADING DAY SCRIPT - START
' =====
' ~~~~~
' Create Position Margin Header Details
PRINT "t.Date", _
      "t.TotEqty", _
      "altSys.TotEqty", _
      "SysNum", _
      "Symbol", _
      "posCnt"
' ~~~~~
' Loop over the systems in the suite.
For iSysNdx = 1 TO test.systemCount
' Set the alternate system by index.
test.SetAlternateSystem( iSysNdx )
' -----
' Access All Systems in this Suite
' -----
' Clear Position Counter
iActivePositionCount = 0
' -----
' Loop over the instruments.
For iSymbolNdx = 1 TO alternateSystem.totalInstruments STEP 1
' -----
' PROCESS ALL OPEN POSITIONS
' -----
' Load System Number: iSysNdx & Symbol Number: iSymbolNdx
iLoadOK = MktSym.LoadSymbol(iSymbolNdx, iSysNdx)
' When Symbol Loads,...
If iLoadOK THEN
' Get Position Unit Count
iPositionQty = MktSym.currentPositionQuantity
' Positions with Quantities > 0 Counter
If iPositionQty > 0 THEN
' Count System Active Positions
iActivePositionCount = iPositionQty + 1
ENDIF ' PositionQty > 0
ENDIF ' iLoadOK
Next ' Symbol_Ndx
' ~~~~~
' Send Data to PrintOutput.csv file
PRINT test.currentDate, _
      test.totalEquity, _
      alternateSystem.totalEquity, _
      iSysNdx, _
      iActivePositionCount
Next ' iSysNdx
' ~~~~~
' =====
' AFTER TRADING DAY SCRIPT - END
' _Alternate System Access Example
' =====

```

Result - Alternate system with the LoadSymbol:

When the above Alternate System Access Example is executed, it will generate a table similar to how this next image displays its PRINT OUTPUT information in a spreadsheet.

	A	B	C	D	E
1	t.Date	t.TotEqty	altSys.TotEqty	SysNum	posCnt
2	1/4/2017	461498.982	127310.25	1	2
3	1/4/2017	461498.982	50000	2	0
4	1/4/2017	461498.982	472332.3615	3	3
5	1/4/2017	461498.982	486852.4658	4	3
6					
7	t.Date	t.TotEqty	altSys.TotEqty	SysNum	posCnt
8	1/5/2017	427623.0388	125630.75	1	2
9	1/5/2017	427623.0388	50000	2	0
10	1/5/2017	427623.0388	440353.1511	3	3
11	1/5/2017	427623.0388	486634.0622	4	7
12					
13	t.Date	t.TotEqty	altSys.TotEqty	SysNum	posCnt
14	1/6/2017	444929.5103	126704.25	1	2

Alternate System Access Example

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.
 - 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:** 126
- Scope:** Block
- Used for Lookback (highlighted with a red box)
- Stepping Enabled
- Stepping Priority:** 0

New Parameter Lookback Disabling Option

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:**Description:**

TrueFalse

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.

Returns:

Function doesn't return a value. It just enables or prevents the automatic reservation of look back data bars.

Example:

```
' BEFORE SIMULATION SCRIPT SECTION
' Disable Auto Priming
test.SetAutoPriming( FALSE )
```

Example:**[Roundtable Forum Question \(link\):](#)**

In my actual code I want a 250 bar lookback on the daily data. I solved my problem by setting the Years of Priming Data to 25 (fortunately I had that much monthly data going back) which allowed scripts like Entry Orders and Update Indicators to be run on the monthly bars from the `test.startDate` forward.

CODE:

```
' Disable Auto-Priming
test.SetAutoPriming( false)
```

This will start running the entry orders script on bar one regardless of lookback parameters or indicators. So when using this, check the `instrument.bar` before accessing any indicators (they might not be ready) or lookbacks. In the case of monthly data based indicators, I sometimes create flexible function based indicators like this:

CODE:

```
' Compute an SMA of the close, using as much data as available, up to 250
sma = AVERAGE( instrument.close, min( instrument.bar, 250 ) )
```

Links:

See Also:

[Miscellaneous Functions](#)

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 532

SetChartSimulationHtml

This `test.SetChartSimulationHtml` function allows the user to insert HTML information, like a custom chart, an image file or text information into the **Stepped Parameter Summary Performance** report using Trading Blox Builder scripts.

When used, it creates an end of test tasks to automatically display a custom chart images, an image file, or scripted HTML wrapped text statements in the area below the **Stepped Parameter Summary Performance Table**.

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:

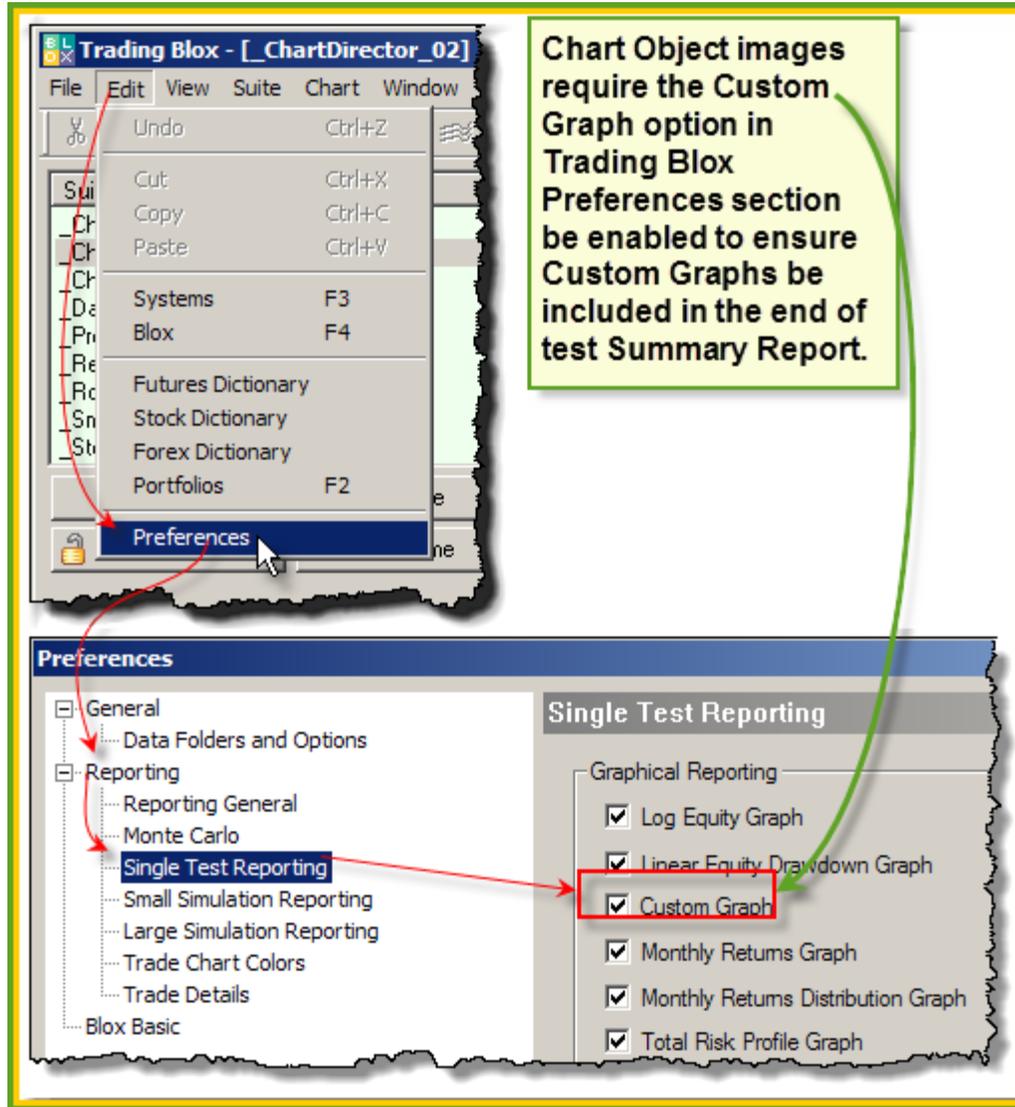
This function is placed in the **BEFORE TEST** script section.

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.

Notes:**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.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.SetChartSimulationtHtml( chartHtml1 + "<br>" + chartHtml2 )
' =====

```

Links:

[currentParameterTest](#), [resultsReportPath](#)

See Also:

Links:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 535

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:	Data Information:
sHTMLImageReference	<p>HTML Image loading reference that includes image path and full file name.</p> <p>Multiple task can be included in as a single parameter by placing a plus-sign + between each image to be loaded.</p> <p>See code example below for exact details on how to create a HTML image source reference.</p>

Notes:

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.

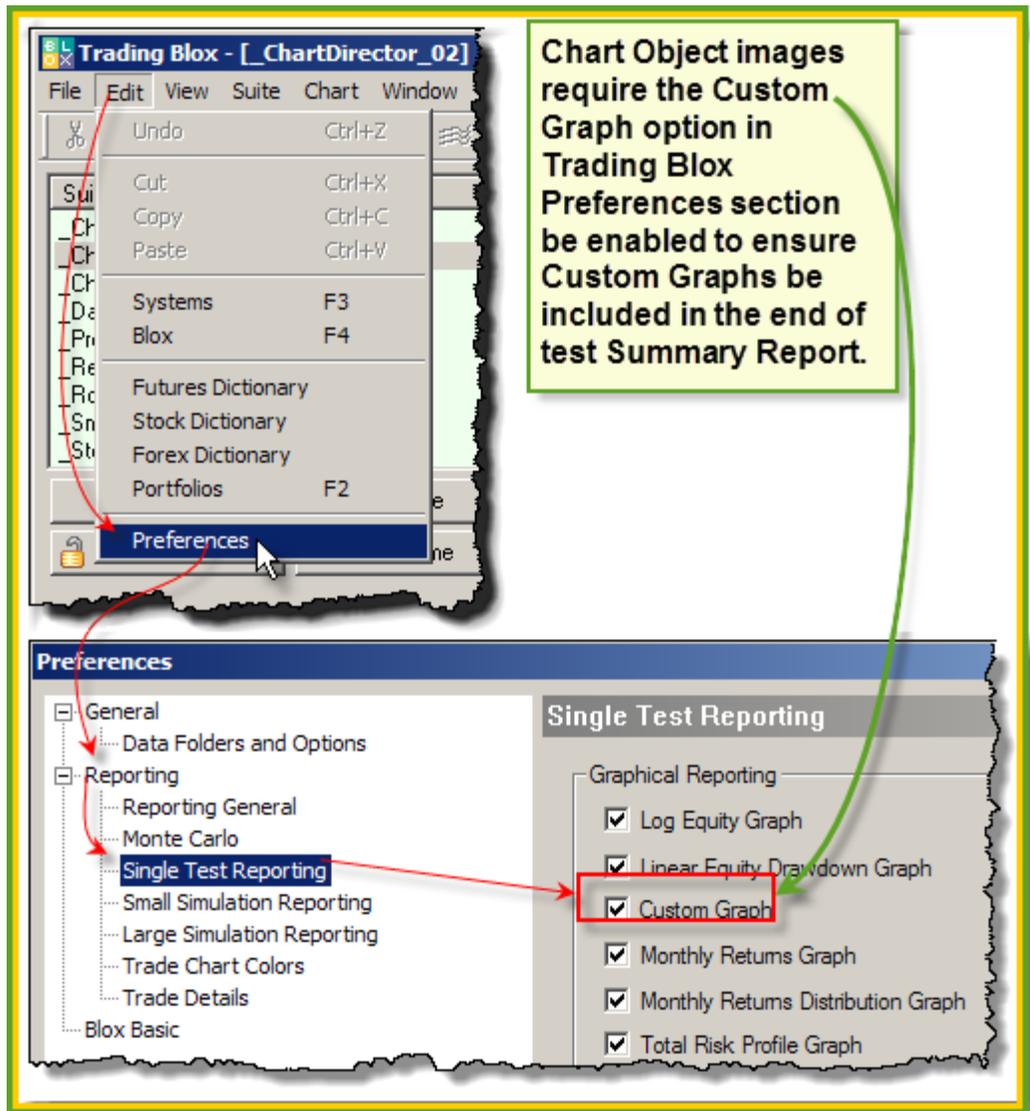
When this method is used with stepped test, the parameter test index is added to the file name created. File names created will each have an index that matches equal the number of steps in test when this method is used with multiple stepped test. When only 1-step is in a test, there will only be one 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 check mark.

Trading Blox Preferences:

Notes:



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("")
' =====
OR
' =====
' This task will load two chart images in the
' simulaiton 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](#)

See Also:

SetDisplayOrderReport

Syntax:

```
test.SetDisplayOrderReport()
```

Parameter:**Description:****Example:****Returns:****Links:****See Also:**

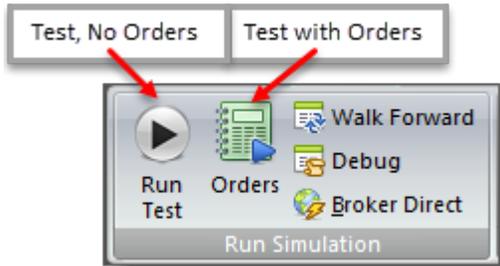
Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 720

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.

When the parameter setting is True, the test will generate orders. When the parameter setting is False a normal test will run without orders being generated.



Run Simulation Test, & Run Simulation Test with Orders.

Because the default test button settings use this True value to generate Orders, and it uses a False value to not generate orders, the same results can be created by selecting the kind of Test Simulation you want to create.

If you decide to use this function, execute this function is used in the Before Test script. This function will enable a couple of stepped test to make profile or other changes before the test run needs to generation orders.

Syntax:

```
test.SetGeneratingOrders( True/False )
```

Parameter:

Description:

True
/False

True Enables Orders, False disables orders

Example:

```
' Disable order generation
test.SetGeneratingOrders( FALSE )
```

Returns:

<None>

Links:

See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 545

SetGoodnessForWalkForward

Trading Blox Builder User's Guide topic "Walk Forward Testing" information about how that process works is explained.

Syntax:

```
test.SetGoodnessForWalkForward( goodnessValue )
```

Parameter:

goodnessValue

Description:

This function requires an entry value. See the [Measure of Goodness Index](#) section of Trading Blox Builder User's Guide.

Example:**Returns:****Links:****See Also:**

SetGoodnessToChart

This function is used to add, one or more statistical functions to a [Stepped Parameter Summary Performance](#) report multi-parameter graph. When more than one additional statistic function is to be added to a multi-parameter graph, use the information in the Example shown below.

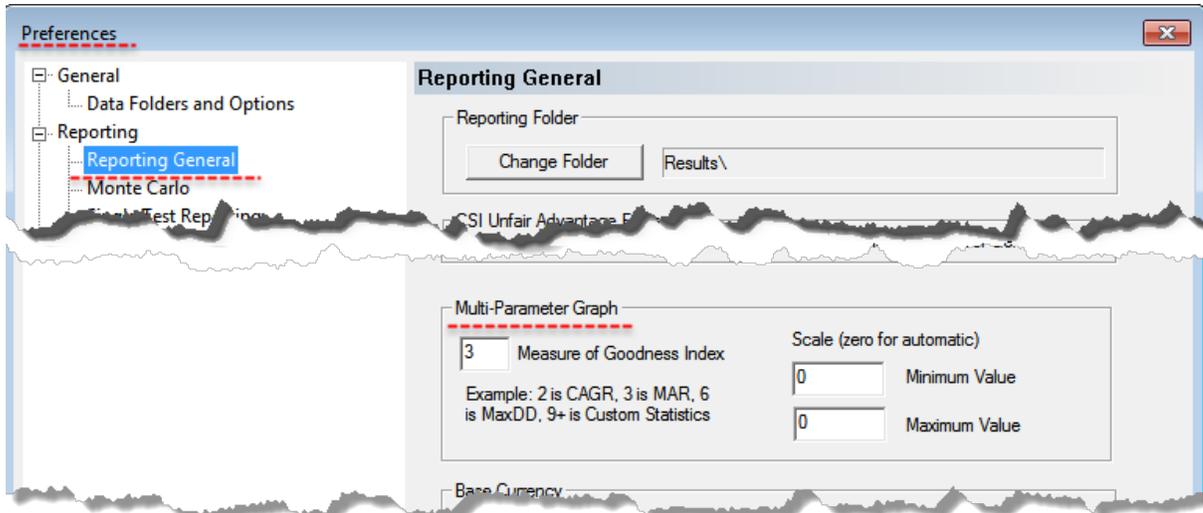
Note:

Trading Blox Builder User's Guide has more information about how this function is applied.

A list of Trading Blox built-in statistic function is available here:

Help File:	Main Menu Topic:	Topic
Trading Blox User's Guide	Test Results "	Trading Performance and Statistics

When the primary Multi-Parameter Graph statistic should be change for all test reports, change the setting in the Preference setting area:



Syntax:

```
test.SetGoodnessToChart( "statisticName1 [ ,statisticName2] [ ,statisticName3] [ , ...]" )
```

Parameter:	Description:
"statisticName"	Add one or more of the built-in statistic name inside of two quotation marks.
[,statisticName2]	When more than one statistic is to be added, separate each statistic name with a comma.
[,statisticName3]	Each added statistic is separated by a comma.

Parameter:	Description:
[, ...]"	Last statistic name must be followed by a closing quotation mark.

Example:

```
' Add the Monthly Sortino test statistic to the stepped-test
' multi-parameter graph
test.SetGoodnessToChart( "Monthly Sortino" )

' Add the multiple test statistic to the stepped-test
' multi-parameter graph, and add statistics to the Custom
' Statistics section and to the summary table
' at the top of the Stepped Parameter Summary Performance report.
test.AddStatistic( "Monthly Sortino", test.monthlySortinoRatio )
test.AddStatistic( "R3", test.rCubed )
test.AddStatistic( "My Value", 7.77 )
test.SetGoodnessToChart( "MAR, R3, Monthly Sortino, My Value" )
```

Returns:

This statistic function is added to the **After Test** script section in any of the blox included in a system. When it is executed, it doesn't return a value. If the blox you are considering doesn't have that script section listed, it can be added by right-clicking on and of the script name shown in the script name section.

Statistics added using this function are for the multiple-parameter graph area of the test results report. They are not added as a statistic item in the **Custom Statistics** section in the summary report. Multiple-Parameter Graphs only appear when a stepped test result report is created.

Added statistics are not related to the goodness measure used to create multi parameter surface charts. Unless it is also set as the default 'measure of goodness' in the Trading Blox Builder **Preferences Large Simulation Measure of Goodness field**.

This function only adds an additional measure to a multi-parameter graph.

Links:

[Miscellaneous Functions, AddStatistic](#)

See Also:

[Trading Blox User's Guide -> Test Results -> Trading Performance and Statistics](#)

SetIBPositionSynchOrderType

Use this function to set the order type used for IB position synch process.

Syntax:
<code>test.SetIBPositionSynchOrderType(orderType)</code>

Parameter	Description:
:	
orderType	Order Type position synchronize value.

Example:
Returns:

Links:
See Also:

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 724

SetMinimumTick

Syntax:

Parameter:	Description:

Example:

Returns:

--

Links:

--

See Also:

--

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 727

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:**Description:**

True
/False

Use True or False keywords, or the values, 1 = True, 0=False

Example:

```
IF test.totalTrades < tradesThreshold THEN
  ' Enable a Silent Test Run
  test.SetSilentTestRun( true )
  ' Output Message
  PRINT "Filtered test ", test.currentParameterTest, _
    " because total trades of ", test.totalTrades, _
    " were less than threshold of ", tradesThreshold
ENDIF
```

Returns:

No Return

Links:**See Also:**

SetSmartFillExit

This option will sort all the exit orders for a particular unit and fill the one closest to the open. Use Smart-Fill-Exit option when the system's exit orders requires that the best, or nearest price be reported in the **Positions & Orders Report**.

Global Parameter Properties



In a system that generates many exit orders at different prices, this [Global Parameter Properties](#) feature set to **TRUE** will limit the orders in the **Positions & Orders** report to the order that has the best protection price.

Notes:

[instrument.SetExitStop](#) function sets an **IPV** for the instrument with the best protective exit stop price. This nearest exit price is then applied the following day.

How this feature works can be seen by stepping this feature's operational state and the adding the exit price rule to the [order.SetRuleLabel](#) or [order.SetOrderReportMessage](#) function. Stepped output results the details of which orders are nearest and then get placed in the **Positions & Order Report**, and the **TradeLog.csv** report can be seen easily.

To aid in this process discovery, Trading Blox Builder provides an Entry-Exit blox named "**Donchian Smart Fills**" where multiple exit orders can be placed at the same time, and the operation of how this feature operates can be seen by using "**Donchian Smart Fills**" in system where both states of this feature are exercised.

This feature can also be enabled and disabled in scripting using this function with either a **TRUE** or **FALSE** value. It is also possible to use scripting to discover this feature's operational state by examining the [test.smartFillExit](#) property (see example).

Syntax:
<code>test.SetSmartFileExit(aTrueFalseValue)</code>

Parameter:	Description:
aTrueFalse Value	Parameter will respond to any True or False value. True = 1, False = -1

Example:

```
' ~~~~~  
' In order for multiple exit orders to work  
' correctly, Smart Fills must be set to true.  
If NOT test.smartFillExit THEN  
' Enable Smart Fill Exit Process so the nearest  
' exit order will be executed first.  
test.SetSmartFillExit( TRUE )  
ENDIF  
' ~~~~~  
OR  
' ~~~~~  
' Note that for multiple exit orders to work  
' correctly, Smart Fills must be set to true.  
If NOT test.smartFillExit THEN  
  MessageBox( "This system requires that the" _  
              + " Global Parameter Smart Fills is" _  
              + " set TO TRUE For the multiple exit" _  
              + " orders TO be processed correctly." )  
  test.AbortSimulation  
ENDIF  
  
' Another option rather than the error message,  
' is to just set Smart Fills to true behind the scenes:  
test.SetSmartFillExit( TRUE )  
  
Print "test.smartFillExit: ", test.smartFillExit
```

Returns:

```
test.smartFillExit: 1
```

Links:

[General Properties](#), [Global Parameter Properties](#), [instrument.SetExitStop](#), [test.smartFillExit](#)

See Also:

[order.SetOrderReportMessage](#), [order.SetRuleLabel](#)

SetTotalParameterRuns

Use this function in the [Before Simulation](#) script section. when executed, it will change the total parameter runs for the simulation. It must be applied in the Before Simulation script so that it can overrides the previous number of total for a simulation.

Syntax:

```
test.SetTotalParameterRuns( totalRuns )
```

Parameter:**Description:**

Parameter:	Description:
totalRuns	Max number of simulation runs for a Simulation.

Returns:

Changes the current number of simulation runs.

Example:**Links:****See Also:**

SortInstrumentList

test.SortStringArray(arrayIndex, direction, elementCount)

Syntax:
<code>test.SortInstrumentList(iMethod)</code>

Parameter:	Description:
iMethod	The number listed in the next table describes the type of sorting that will happen.

Sorts the physical instrument list that is used for the simulation loop using the method indicated:

Method Value:	Sorting Method Description:
1	Long Ranked value sorts the instrument list by the current Long Rank number order.
2	Long Ranking script assigned value will be sorted in ascending order.
3	Dictionary assigned Original Order Sort value.
4	Custom Sort Value order.
5	Alphabetical by Symbol
6	Alphabetical by Group.
	<p>Note: All market symbols are sorted regardless of whether they are primed when the sorting happens.</p> <p>Instruments for these Sort Types need a dictionary source for these methods:</p> <ul style="list-style-type: none"> 3 Order Sort Value 4 Custom Sort Value 6 Group Name

Example:
See the System.SortInstrumentList(5) topic for scripting information.
Returns:

Links:

See Also:

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 722

Test String Arrays

Note:

- **A Test-Level multi-column array was introduced in version 3.x.**

With the later releases of Trading Blox Builder, it is not recommend to create String-Series methods using this Test-Level array ability, because It has been deprecated and is only still available for compatibility.

Instead, use the BPV series in its place. BPV series are not as complicated to work with the test-level arrays, and the BPV series are not a depreciated feature that will eventually be removed. However, the Test-Level series method is only available as a test level series, whereas the normal BPV String arrays are available any place.

[String Array Details, Sorting Columns & Random](#) (Web Link)

(Blox are available in the post at above link)

Test-Level string arrays are created so that any blox within the test suite can access the information contained in the string array.

There are a lot of uses for having a string array available so that it is available to all levels of a system. In addition, columns within the string array can be sorted independently, if needed. This ability creates an opportunity to have a keyword index stored in a different column that can be sorted in ascending, or descending order. By creating a keyword with an index, various index columns can be used to access data contained in other columns of the array in different sequences. Each sequence can be controlled by how the keyword-index is sorted and created.

Understanding String Size, Row and Column Count Parameters:

Trading Blox allows a maximum of 100-columns to be created. There is also a maximum limit of 10,000 rows that can be created. Values larger than the above will generate a run-time error that will stop the script from running.

All String lengths in Trading Blox are limited to 512-Characters. This is can be adjusted in String arrays so that each column can use the maximum size, or a size value that is smaller. The only lower character limit is a size of 1-character, or a size that would safely contain the maximum number of characters you wanted to store in any of the string array's locations. In the example provided, the maximum String size is used because the example was created to support a need to store open trade and new signal record details.

When creating any string array, each character will consume a character-size memory space for each element in an array. Each row in a column of an array is a multiplier of the string size, and each column is a multiplier of the memory consumed by a column.

Still, memory space isn't a problem for most modern computer users. In the majority of cases there is more memory space than a user needs most of the time. However, as the number of instruments in a portfolio increases, and length of data grows, at some point memory could be an issue requiring the user to reduce the number of instruments, or length of data that is being loaded.

In most cases most of the expected string arrays aren't likely to be a memory space hog, but sizing the array's string elements should be something you think about when you make the decision, so you don't make the string size larger than is needed, or create more rows and columns than are required.

Creating String-Array:

Once the number of row and columns, and the size of each string array element has been decided, execute the `test.createStringArray` method shown and if the parameters used with the method are within bounds the String-Array will be ready for use:

Example - 01:

```
' ~~~~~
' STRING ARRAY ROW & COLUMN SIZE DETAILS

' Array Setup Parameters
StrRowLen = 512           '      512      Maximum String Length
Array_Columns = 2        '      100      Maximum Column Count
Element_Rows = 10        ' 10,000      Maximum Row Count

' ~~~~~
' CREATE TEST-LEVEL STRING ARRAY
Test.CreateStringArray( Array_Columns, Element_Rows, StrRowLen )
```

Storing and Retrieving String Data Information:

All arrays can be thought of as simple series of rows in a table. In Trading Blox the standard **BPV** and **IPV** series only have 1-dimension arrays to controls the number of rows in the series. If an Auto-Indexing series is selected, Trading Blox will handle the chore of deciding how many rows are needed, and it will also keep each row aligned to either the Instrument Date for IPV Auto-Indexed series, or it will align series rows to the `test.currentDate` for BPV Auto-Indexing series.

Trading Blox also allows manual indexing of numerical and string series to be created. All numerical series are of TYPE: Floating and all String series are of Type String. Numerical series will accept any decimal or integer value.

Manual indexed series requires the user to determine the number rows before it is needed to prevent an access error. This can be done by checking the need for more rows and comparing how many element spaces are available in the series. Once the difference between the need and the rows remaining is know, ,the series rows can be increased or reduced before they are needed. All manual-sized series require the programmer to maintain an index that will be the index value for each row in a series. Access can be sequential, or random when a manual index is the means of access.

All IPV and BPV series manually Index are supported by special methods for sorting and for changing the size of a manual-indexed array. See [Series](#) topic for more complete details.

String Array series are a little different than Numerical series, but the need for determining how many rows are need, and the manual index tracking for each series is about the same. Sorting is

also almost the same, with the exception that different methods are needed with String-Arrays, than those used with Numerical arrays.

Test-Level String Arrays allow the user to create String-Type series with multiple columns. This ability can provide data isolation, or information can be linked if the order of each column is not changed, or an index reference in one column will point to a reference in a different column.

Test-Level String Arrays don't have a method for finding how many rows are available in the series, or for changing the count once the series is created. This means it is important to set the count for the number of rows needed accurately, or to be generous so that you won't run out of places to store information.

Indexing for both data types is exactly the same, but the storage and retrieval of data from a String Array element requires the use of special methods. For storing information, set the index to the next available element, and then execute the following method:

Example - 02:

```
' ~~~~~
' STORE DATA IN AN ARRAY ELEMENT

' Send data in sTemp1 in Column 1 at Row# Ndx
Test.SetStringArrayElement( Column_1, Ndx, sTemp1 )
```

Retrieving information requires the element-locating index to be set to the desired location and the following method executed:

Example - 03:

```
' ~~~~~
' RETRIEVE DATA FROM AN ARRAY ELEMENT

' Get data in Column 1 at Row# Ndx, and place it in sTemp1
sTemp1 = Test.GetStringArrayElement( Column_1, Ndx)
```

Example - 04, Function:

```
' 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.

' To access to the global test level string arrays, use
```

Example - 04, Function:

```
' the following functions and properties:
test.CreateStringArray( arrayCount, elementCount, stringLength )

' To 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 )

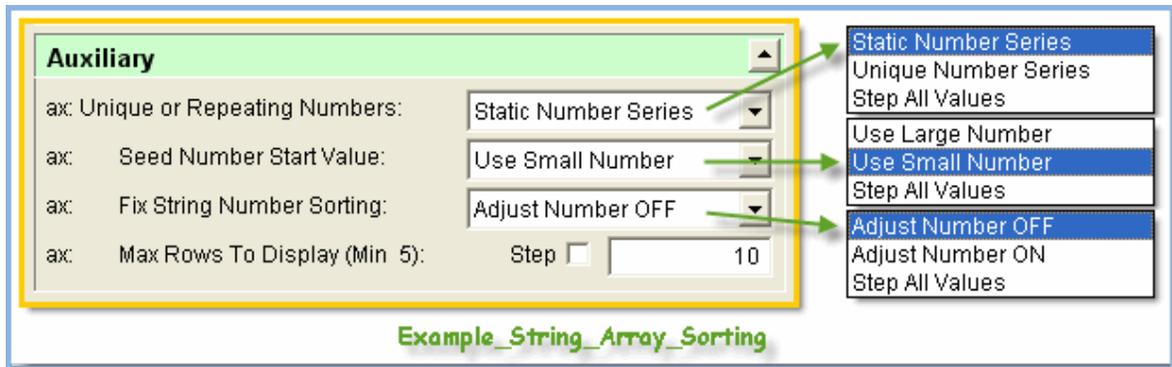
' Sets a string into the arrayIndex array at elementIndex element.
```

Links:[Test String Array Sorting](#)**See Also:**[Example_String_Array_Sorting Blox](#)

Test String Array Sorting

Example Blox Description:

Box contains a simple Test-Level String Array Example using numbers in a string format. Numbers were selected because they are the least understood when it comes to sorting strings that look like numbers. Numbers are generated using the Trading Blox Random Number generator. There are also options in this blox for changing how the numbers are sequenced, and for how they are stored in the string array.



Operation Notes:

When testing this blox, start off with the default settings to get an understanding of what it does. Once you've got a sense of how the blox operations, change the fix number option next so that the numbers will sort properly.

From here on, changes to the other parameters will help you get an understanding of how the character of the Trading Blox Random Number data series can change by changing the initial start value, or by changing the value of numeric Seed value supplied to the [RandomSeed](#) function.

When this blox is executed in a Simulation run, the only active script section is the Before Test Script. Before Test script location is the best place I've found for creating a Test-Level String array. It is also the best place for accessing selectable user parameters that can decide which scripts executes during a simulation run.

Any script section can be used to store, or access information in a Test-Level script once it has been created. There are no known restrictions on creating the Test-Level script array in other sections as long as you know it will be created before you need to store, or retrieve initialization information in the string array.

There are a lot of little script techniques available in this simple example along with the process for creating a Test-Level string array, storing data in each element in each column, sorting data in each column in either ascending, or descending orders, and for how to retrieve the values in each string element.

Operation Notes:

There are also some formatting techniques to make output more readable, or presentable for others. There are also two unique user functions. One user function is intended to return the value of the computer's time of day in the form of the number of seconds that have transpired since 12:00 AM midnight. This is useful for determining the differences between the two different times of the day, or even between days when a serialized date value is part of the calculations.

In the second function is a process for creating a unique random number seed value. To create a unique number the script function creates a date serialization process that uses an adjusted Julian date value appended to the value in the number of seconds from midnight. This unique number provides a good source for seeding the random number series with unique values so the random number generator will produce a unique series of number sequences each time the random number generator is operated.

User Parameters:	Descriptions:
Random Series Sequencing	<p>In the default setting the numbers generated will always be the same for each simulation. This setting is using a fixed seed value. Fixed seed values initialize the Random Number series so that the same sequence of random numbers is generated each time the random number generator is accessed. Having a consistent number in a random series makes trouble shooting random number driven events easier to debug, but they aren't good for "Monte Carlo" type data methods.</p> <p>A second option uses a unique number each time a series of numbers is generated. By using a unique seed value for each random number sequence the order of the numbers generated will not be the same for each simulation run.</p>
Random Number Start	<p>By default the settings use a small number to set the range of numbers that will be created for each series. This small number provides a higher probability that the range of random numbers in the sequence will be over a wider range of values. This was provided because of the limited number of numbers being generated to keep the example within tolerable reading range.</p> <p>A larger start number is available to show that when the numbers have the same number of characters in their values, then the sorting option will sort those numbers correctly as long as the character count in a string number is the same for all string numbers. However, if the number of characters in the string-number changes, the sort operation is likely to not be sorted properly.</p>
Fix String Array Sorting	<p>First Option shows how numbers stored without regard for their ASCII character representation will not always sort correctly.</p>

User Parameters:	Descriptions:
	<p>Second Option shows how to adjust for numbers that have different lengths so that when they are sorted they are in a correct ascending, or descending sequence. In the example shown, the number adjustments are using variable number of leading zero values. However spaces, underscore characters could also have been used. There are other methods for adjusting string numbers, so if the examples aren't enough information, then try some math approaches, or post a question about your idea.</p>
<p>Max Array Rows to Display</p>	<p>Trading Blox has an upper limit of 10,000 rows that can be created with a Test-Level String Array.</p> <p>However, each row will reserve memory space times that will equal the array element time the number of columns. In this example we are only using 2-columns. However, that can be changed by modifying the script code, and by adjusting how the loops handle the increased, or reduced number of column.</p> <p>For this simple example, the minimum number of rows is set to 10-rows. You get to set the row count from any number between 5, and whatever value seems reasonable to scroll through once the blox has completed.</p>
<p>Blox Use</p>	<p>No Portfolio instruments are required to run this blox.</p> <p>To test this blox, create a system name you want, or use the default system name "Chart" and attach this Blox. When this blox executes, it will generate information into the Trading Blox Log window on the Main screen of the program when the Log option under the Window menu item is selected. It will always generate and send information to the "Print Output.csv" file.</p> <p>"Print Output.csv" file is easily opened using the Main menu File selection and then scrolling down until Result selection is highlighted. With the list visible, select the Print Output option and the output will appear in an Excel spreadsheet. While the data appears easily, it will look better if you access it with the Windows Notepad. File is in the Trading Blox Results folder and will easily open with Notepad, or WordPad.</p>

The code examples show ideas about the use of String Array Sorting. The examples execute the **BEFORE TEST** script section of the blox available in the **Blox MarketPlace**. When example code is executed it will produce something similar to the details shown below:

Example:

```

' =====
' Example_String_Array_Sorting
' BEFORE TEST SCRIPT - START
' =====
' ~~~~~
' CONSTANTS
' ~~~~~
' StringPad Len Must be as Large as Largest Number Length
sNbrPad = "000000"
NbrPadLen = Len(sNbrPad)

sStrPad = "      "
StrPadLen = Len(sStrPad)
' ~~~~~
' RANDOM SEED ASSIGNMENT
' ~~~~~
' Allow Unique Random Series, or Static Series
If Random_Series = UNIQUE_NUMBER_SERIES THEN
  Seed_Number = SystemTime( 3 )'Script.Execute( "Unique_Number" )
ELSE
  Seed_Number = 1084239526
ENDIF ' Random_Series = UNIQUE_SERIES

' Initialize Random Number Series Sequence
Seed_Number = RandomSeed( Seed_Number )
' ~~~~~
' CREATE PARAMETER SELECTORS
' ~~~~~
' Next conditional statement requires a parameter to select
' the size of the number to use for creating values.
' In operation it changes the Range of Numbers so String
' Sorting Issue is Shown
If Rnd_Number_Start = USE_SMALL_NUMBER THEN
  Rnd_Start = 100 ' Small Value will generate small numbers
ELSE
  Rnd_Start = 1000 ' Generates larger, 2-digit numbers
ENDIF

' Next conditional statment uses a parameter selector to
' control how the sorting is performed.
If Pad_Number = ADJUST_NUMBER_ON THEN
  ' Show Sorting Number Fix
  sPadFix = "Fix Number Sorting"
ELSE
  ' Show Sorting Number Fix
  sPadFix = "DO NOT Fix Number Sorting"
ENDIF

' Use Maximum Row Count Value
Series_Rows = Max(5, Row_Count)
' ~~~~~
' STRING ARRAY ROW & COLUMN SIZE DETAILS
' ~~~~~
' Array Setup Details
StrRowLen = 512 ' 512 = Maximum TB String Length

```

Example:

```

Array_Columns = 100
Element_Rows = Series_Rows

' Array Column Locators
Column_1 = 1
Column_2 = 2
' ~~~~~
' CREATE TEST-LEVEL STRING ARRAY
' ~~~~~
test.CreateStringArray( Array_Columns, Element_Rows, StrRowLen )
' ~~~~~
' GIVE USER SOME INFORMATION
' ~~~~~

PRINT
PRINT "Bloc: ", block.name, "          Script Name: ", block.scriptName
PRINT
PRINT "Computer Date & Time: ", SystemDate (), " ", SystemTime()
PRINT
PRINT "String Array Creation & String Number Sorting Examples"
PRINT
PRINT "PARAMETERS:"
PRINT "      Array Rows      : ", Element_Rows
PRINT "      Random Start #: ", Rnd_Start
PRINT "      Fix # Sorting  : ", sPadFix
PRINT "      Seed Number    : ", Seed_Number
PRINT
PRINT "ARRAY DEFAULTS:"
PRINT "      String Element Size : ", StrRowLen
PRINT "      Array Column Count : ", Array_Columns
PRINT
PRINT "-----"
PRINT
PRINT "String Array Created with " _
+ AsString(Array_Columns, 0) _
+ " Columns, and " _
+ AsString(Element_Rows, 0) _
+ " Element Rows."

PRINT
PRINT "Generating Alpha-Numeric Data for Column 1 Rows:"
' ~~~~~
' CREATE DATA FOR ARRAY'S FIRST COLUMN ROWS
' ~~~~~
For Ndx = 1 TO Element_Rows
' Generate a random number for this row
sRandom_Num = AsString(Random( Rnd_Start, 10000 ) * 0.01, 2)

' USER CAN CHANGE HOW NUMBER SORTING WORKS
If Pad_Number = ADJUST_NUMBER_ON THEN
' Build an Initial Row Location Identifier
sRow = Left(sStrPad, StrPadLen - Len(Ndx)) _
      + AsString(Ndx, 0)

' Build the data in a simple COMMA Delimited Record
sTemp1 = Left(sNbrPad, NbrPadLen - Len(sRandom_Num)) _

```

Example:

```

        + sRandom_Num + "," + " in Row " + sRow

    ' Show User the Simple Record
    PRINT sRow, " " + sTemp1
ELSE
    ' Build an Initial Row Location Identifier
    sRow = AsString(Ndx, 0)

    ' Build the data in a simple COMMA Delimited Record
    sTemp1 = sRandom_Num + "," + " in Row " + sRow

    ' Show User the Simple Record
    PRINT Ndx, " " + sTemp1
ENDIF

    ' Send the Record to the Array in Column 1, Row of Ndx
    test.SetStringArrayElement( Column_1, Ndx, sTemp1 )
Next ' Ndx
' ~~~~~
' SEND MORE INFORMATION
' ~~~~~
PRINT
PRINT "Generating Alpha-Numeric Data for Column 2 Rows:"
PRINT
' ~~~~~
' CREATE DATA FOR ARRAY'S SECOND COLUMN ROWS
' ~~~~~
For Ndx = 1 TO Element_Rows
    ' Generate a random number for this row
    sRandom_Num = AsString(Random( Rnd_Start, 10000 ) * 0.01, 2)

    ' USER CAN CHANGE HOW NUMBER SORTING WORKS
    If Pad_Number = ADJUST_NUMBER_ON THEN
        ' Build an Initial Row Location Identifier
        sRow = Left(sStrPad, StrPadLen - Len(Ndx)) _
            + AsString(Ndx, 0)

        ' Build the data in a simple COMMA Delimited Record
        sTemp2 = Left(sNbrPad, NbrPadLen - Len(sRandom_Num)) _
            + sRandom_Num + "," + " in Row" + sRow

        ' Show User the Simple Record
        PRINT sRow, " " + sTemp2
    ELSE
        ' Build an Initial Row Location Identifier
        sRow = AsString(Ndx, 0)

        ' Build the data in a simple COMMA Delimited Record
        sTemp2 = sRandom_Num + "," + " in Row" + sRow

        ' Show User the Simple Record
        PRINT Ndx, " " + sTemp2
    ENDIF

```

Example:

```

' Send the Record to the Array in Column 1, Row of Ndx
test.SetStringArrayElement( Column_2, Ndx, sTemp2 )
Next ' Ndx
'
' ~~~~~
' SEND MORE INFORMATION
' ~~~~~
PRINT
PRINT "-----"
PRINT
PRINT "Sorting Array Elements in the Column #1 in Descending Order:"
PRINT "Sorting Array Elements in the Column #2 in Ascending Order:"
PRINT
PRINT "-----"
'
' ~~~~~
' SORT EACH STRING ARRAY COLUMN IN DIFFERENT DIRECTIONS
' ~~~~~
test.SortStringArray( Column_1, 0 , Element_Rows )
test.SortStringArray( Column_2, 1, Element_Rows )
'
' ~~~~~
' SHOW WHAT HAS BEEN ACCOMPLISHED
' ~~~~~
PRINT
PRINT "Sorted Alpha-Numeric Data for Both Array Columns:"
PRINT
PRINT "Index  Column 1          |          Column 2"

For Ndx = 1 TO Element_Rows
' Get Record from Column 1, Row #Ndx
sTemp1 = test.GetStringArrayElement( Column_1, Ndx)

' Get Record from Column 2, Row #Ndx
sTemp2 = test.GetStringArrayElement( Column_2, Ndx)

' Show User Results of Sorted Columns
sRow = Left(sStrPad, StrPadLen - Len(Ndx)) _
      + AsString(Ndx, 0)
PRINT sRow, " " + sTemp1, " | " + sTemp2
Next ' Ndx
PRINT
PRINT "-----"
'
' =====
' BEFORE TEST SCRIPT - END
' Example_String_Array_Sorting
' =====

```

Returns:

```

Blox: ,Example_String_Array_Sorting,          Script Name: ,Before Test,
Computer Date & Time: ,2014-03-04, ,1:47:38 PM,
String Array Creation & String Number Sorting Examples,
PARAMETERS: ,

```

Returns:

```

Array Rows      : ,10,
Random Start #: ,100.000000000,
Fix # Sorting  : ,DO NOT Fix Number Sorting,
Seed Number    : ,0.000000000,

```

```

ARRAY DEFAULTS:,

```

```

String Element Size : ,512,
Array Column Count  : ,100,

```

```

-----,
String Array Created with 100 Columns, and 10 Element Rows.,

```

```

Generating Alpha-Numeric Data for Column 1 Rows:,

```

```

1, 55.33, in Row 1,
2, 59.69, in Row 2,
3, 71.81, in Row 3,
4, 84.59, in Row 4,
5, 60.67, in Row 5,
6, 85.94, in Row 6,
7, 54.94, in Row 7,
8, 84.88, in Row 8,
9, 42.94, in Row 9,
10, 62.73, in Row 10,

```

```

Generating Alpha-Numeric Data for Column 2 Rows:,

```

```

1, 64.94, in Row1,
2, 39.05, in Row2,
3, 44.32, in Row3,
4, 30.45, in Row4,
5, 89.29, in Row5,
6, 6.61, in Row6,
7, 96.41, in Row7,
8, 27.99, in Row8,
9, 38.96, in Row9,
10, 48.29, in Row10,

```

```

-----,
Sorting Array Elements in the Column #1 in Descending Order:,

```

```

Sorting Array Elements in the Column #2 in Ascending Order:,

```

```

-----,
Sorted Alpha-Numeric Data for Both Array Columns:,

```

Index	Column 1	Column 2,
1,	85.94, in Row 6,	27.99, in Row8,
2,	84.88, in Row 8,	30.45, in Row4,
3,	84.59, in Row 4,	38.96, in Row9,
4,	71.81, in Row 3,	39.05, in Row2,
5,	62.73, in Row 10,	44.32, in Row3,
6,	60.67, in Row 5,	48.29, in Row10,
7,	59.69, in Row 2,	6.61, in Row6,
8,	55.33, in Row 1,	64.94, in Row1,
9,	54.94, in Row 7,	89.29, in Row5,
10,	42.94, in Row 9,	96.41, in Row7,

Links:
See Also:

Available Downloads:	Description:
Traders Round Table's Blox Marketplace: Example: String Array Details, Sorting Columns & Random	There are two example blox (version 3.x & 4.x) available in Topic:
Example_String_Array_Sorting.tbx	TB4 - Example_String_Array_Sorting
Test-Level String Array Series.pdf	Information Shown Above
Example_String_Array_Sorting.tbx	TB Ver: 3.2.x Compatible String Array Creating, Access, Sorting, and Random Number details

Edit Time: 3/21/2024 10:41:42 AM

Topic ID#: 615

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.
- Positive value adjustments increase the expense amount of the Other Expense sub-account total.
 - Negative value adjustments reduce the expense amount of the Other Expense sub-account total.

Returns:

See example.

Example:

```
' Negative value adjustments reduce the Other Expense total.
test.UpdateOtherExpenses( -25000 )

' Positive value adjustments increases the Other Expense total.
test.UpdateOtherExpenses( 25000 )

' 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:

[CapitalAddsDrawsTotal](#), [OtherExpenses](#)

See Also:

Edit Time: 9/11/2020 4:48:30 PM

Topic ID#: 641

SetMarginEquity

Enter topic text here.

11.5 Test Statistics

Test statistics are designed to be scripted 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.

Example:

```
' Each property can be used as value reported using a
' Print or other output type of function:
Print "Goodness Measure ", test.goodness
OR
' As a value assigned to a variable:
value = test.goodness
```

Note:

Some statistics calculate the value when used, so use with care as this can be an issue that effects performance.

Statistic Name:	Description:
annualGeometricReturn	Annual geometric return
annualizedDailyReturnSD	Annualized Standard Deviation of the Daily Returns
annualizedMonthlyReturnSD	Annualized Standard Deviation of the Monthly Returns
annualizedMonthlySharpeRatio	Annualized monthly Sharpe Ratio
annualReturnDownsideDeviation	Downside Standard Deviation of the annual return
annualReturnStandardDeviation	Standard deviation of the annual returns
annualSharpeRatio	Annual Sharpe Ratio
annualSortinoRatio	Annual Sortino Ratio
averageAnnualReturn	Average annual return
averageClosedDrawdown	Average closed equity drawdown percent
averageDailyReturn	Average daily return
averageLongestDrawdown	Average longest drawdown
averageLossPercent	Average percent loss of the losing trade
averageMarginEquityRatio	Summary Margin to Equity Ratio (See User Guide:

Statistic Name:	Description:
	Margin to Equity Ratio: $\text{MarginEquityTotal} / \text{TotalTradingDays}$)
averageMaxDrawdown	Average maximum drawdown
averageMonthlyReturn	Average monthly return
averageOpenDrawdown	Average total equity drawdown percent
averageRiskPercent	Average percent risk per trade
averageTradePercent	Average percent profit per trade
averageWinPercent	Average percent win of the winning trades
calmarRatio	Calmar Ratio
closedDrawdownStandardDeviation	Standard Deviation of the closed equity drawdown percent
dailyGeometricReturn	Daily geometric return using calendar days
dailyGeoSharpeRatio	Daily geometric Sharpe Ratio
dailyReturnDownsideDeviation	Downside Standard Deviation of the daily return
dailyReturnStandardDeviation	Standard deviation of the daily returns
dailySharpeRatio	Daily Sharpe Ratio
dailySortinoRatio	Daily Sortino Ratio
earnedInterest	Total earned interest
expectationRatio	Sometimes known as Expectancy, this statistic shows the suite's end of test Expectancy Ratio of how much one expects to gain for every amount bet or risked on a given trade. Numbers greater than '0.0' are winning systems, less than '0.0' are losing systems.
goodness	Return will be from the statistic assigned in Preferences' Reporting General Multi-Parameter Goodness field . Value display is the statistic current value.
longestOpenDrawdownMonths	Longest total equity drawdown in months
losingMonths	Total number of losing months
lossCount	Total losing trades
marginInterest	Total margin interest
marRatio	MAR ratio

Statistic Name:	Description:
maxClosedDrawdown	Max closed equity drawdown percent
maxClosedMonthlyDrawdown	Max closed equity monthly drawdown percent
maxOpenDrawdown	Max total equity drawdown percent
maxOpenMonthlyDrawdown	Max total equity monthly drawdown percent
modifiedSharpeRatio	Modified Sharpe Ratio
monteCarloConfidenceDrawdown	
monteCarloConfidenceDrawdown2	
monteCarloConfidenceDrawdown3	
monteCarloConfidenceDrawdownLength	
monteCarloConfidenceDrawdownLength2	
monteCarloConfidenceDrawdownLength3	
monteCarloConfidenceMAR	
monteCarloConfidenceReturn	
monteCarloConfidenceRSquared	
monteCarloConfidenceSharpe	
monthCount	Total number of months
monthlyReturnDownsideDeviation	Downside Standard Deviation of the monthly return
monthlyReturnStandardDeviation	Standard deviation of the monthly returns
monthlySharpeRatio	Monthly Sharpe Ratio
monthlySortinoRatio	Monthly Sortino Ratio
netProfit	Net profit
openDrawdownStandardDeviation	Standard Deviation of the total equity drawdown percent
percentProfitFactor	Profit factor percent (win percent / loss percent)

Statistic Name:	Description:
profitFactor	Profit factor (win/loss)
rar	RAR - Risk Adjusted Return.
rCubed	R-Cubed
robustSharpe	Robust Sharpe Ratio
roundTurnCount	Number of round turns
rSquared	R-Squared
totalCarry	Total cost of carry
totalCommissions	Total commission
totalLossDollars	Total dollars from losing trades
totalSlippage	Total slippage
totalTrades	Total trades not including zero size trades
totalWinDollars	Total dollars from winning trades
walkForwardGoodnessValue ()	This is the same as test.goodness - Return will be from the statistic assigned in Preferences' Reporting General Multi-Parameter Goodness field . Value display is the statistic current value
walkForwardGoodnessName	Returns the GoodnessName text.
winCount	Total winning trades, including break even
winningMonths	Total number of winning months

11.6 Trade Properties

Test level closed trade details of all the instruments from all of the systems in the test suite is available from these properties.

Each record of trade information is identified by the test object property: `test.tradeSystem[x]`

Value of '`[x]`' is the processing loop's index that increments as the information is processed. In the examples below, the loop starts at '1' so the earliest, or oldest trade record is at the top of the list table that is created by each example. When more than one system is in a suite, trade records from all the systems will appear in the table. Their position in the table will be based upon their trade entry date. System number identifies with of the many systems in the suite created the trade record details.

Properties:	Description:
averageTradeDuration	Average trade duration of all trades in the test, computed using the tradeDaysInTrade property
savedTradeCount	Number of trades saved by the WF process from one OOS test to the next.
tradeBarsInTrade[]	Number of bars between entry and exit
tradeCommission[]	Total trade commission expense.
tradeCount	Number of prior trades including zero size trades. Used to index the following properties:
tradeCustomValue[]	Custom value as set through scripting
tradeDaysInTrade[]	Number of days between entry and exit (includes weekends and holidays)
tradeDirection[]	Direction as a description of LONG or SHORT text.
tradeDollarsPerPoint[]	Dollars per point on the entry day
tradeEntryBPV[]	Entry BPV of the instrument
tradeEntryDate[]	Entry date
tradeEntryFill[]	Entry fill price
tradeEntryOrder[]	Entry order price
tradeEntryRisk[]	Entry risk as a percent of entry day trading equity
tradeEntryStop[]	Initial entry day stop, if used
tradeEntryTime[]	Entry time
tradeExitDate[]	Exit date
tradeExitFill[]	Exit fill price

Properties:	Description:
tradeExitOrder[]	Exit order price
tradeExitTime[]	Exit time
tradeMaxAdverseExcursion[]	Maximum Adverse excursion of the trade
tradeMaxFavorableExcursion[]	Maximum Favorable excursion of the trade
tradeMinFavorableExcursion[]	Minimum Favorable excursion of the trade
tradePositionReferenceID[]	Position Reference value.
tradeProfit[]	Closed out profit including Slippage and Commission
tradeProfitPercent[]	Profit as a percent of entry day trading equity
tradeQuantity[]	Quantity in shares or contracts
tradeRuleLabel[]	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:

Examples shown report the first to the last trade created by the suite.

Properties listed with a '[]' following them require an index numerical value. Index values cannot be less than one, or more than the total number of trades reported by: `test.totalTrades`

Most Recent Trade Reported First:

Example - 1:

```

' 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 - 1:

```

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 earliest Trade information will appear first:

Example - 2:

```

' 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 = 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 - 2:

```

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 [Trading Blox Website Documentation Section](#) for the current PDF, Help File, or Web Based versions of these manuals.

A lot of information about Trading Blox can be found on its [Traders' Roundtable Forum](#) website. Access to the forum is free. If you register and have a problem, send an email to customersupport@tradingblox.com and explain the problem you are having accessing the Forum.

Edit Time: 9/11/2020 4:48:26 PM

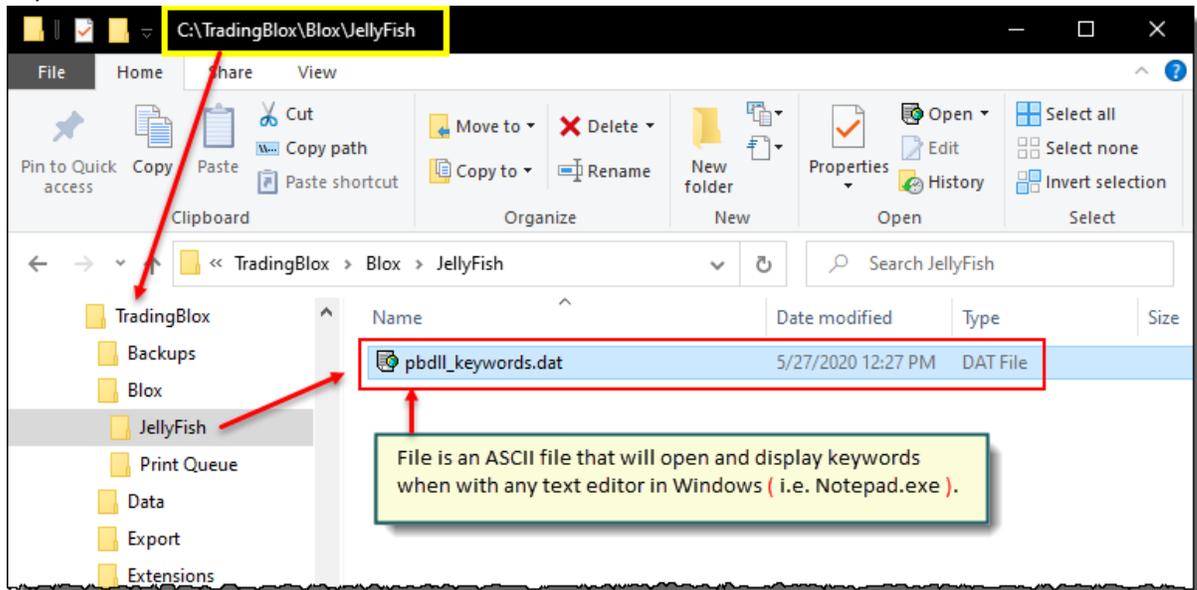
Topic ID#: 212

Section 1 – Trading Blox Keywords

Trading Blox Builder Keywords are at the heart of building trading system components, and how a test will execute.

Trading Blox Builder **Basic Keyword** abilities are used in scripting when the user needs more than basic common programming structures to complete a task.

Each new Trading Blox Builder version will get an updated current list of Trading Blox Builder Basic keywords to use:



Trading Blox Keyword Name List

The `pbdll_keywords.dat` file is a text file with a ".Dat" suffix.

To use the KeyWord list, download the file and use a text editor like the Windows Notepad, or free [Notepad++](#) editor.

Section 2 – The Life of a Test

Each suite will always have a tab section that allows the user to determine the settings for how a suite will test the systems selected to perform in a suite. **Test Start** and **End of Test** parameter settings are explained and displayed in the **Trading Blox Builder User Guide** section named **Simulation Parameters**. Reading and understanding that section will be helpful to understanding this topic, and in understanding how a test is controlled.

1) Test Start Date:

The specified earliest **Start Date** of the test is determined by the value the user enters into the **Simulation Parameters - Start Date** parameter field. The actual **Start Date** of a test is determined by the system requirements and the data record dates of the instruments in the selected portfolio for a system. For example, if the system requires look-back calculations to prime its indicators, each instrument in the system's portfolio will need provide the number of records required so that instrument's indicators can be primed. Once an instrument has primed its indicators, the **Simulation Parameters - Start Date** parameter value will allow the entry orders of a system to determine if an order to enter can be created. If an instrument's prime data is earlier than the **Simulation Parameters - Start Date** parameter value, and the system's rules for creating an entry order will test each of the instruments to determine in an entry order can be created. The actual Start Date displayed in the Chart's display for each instrument will always be the earliest data allowed that meets the **Simulation Parameters - Start Date** parameter value. The earliest trade date, will always be the earliest date an instrument allows the system to test a primed instrument and the conditions found for the first entry date that meets the system's logic to create an entry order.

2) End Test Type Timing:

Timing for when a test ends is determined by the user's selection of how the test should end. Selections for ending a test can be to **"Use All the Data"** available, or **"Use a Date"** value entered, or it can be a **"Fixed Length"** that limits the number of instrument records in the test.

How to use **"End Test Type"** timing is described in the **Simulation Parameters - Test End Type** parameter description information.

When a test ends using the **"Run Test"** option, the system will close all open positions and calculate the summary results for the test. When a test ends using the **"Orders"** option, the system does not close any open position that didn't have an order to close that position on the date the test ends. This difference of not closing open positions, will create different information in the trade log, but the open positions will be displayed in the **"Orders"** tab near the **"Chart"** tab.

By placing the open positions in the open positions and open orders report, the active position information and the orders for the next trade date can be found.

3) Indicator and System Information Priming:

On the Start Date, if you have enough data in an instrument, and the value that is displayed in the **Preferences Data and Folder** section there the **"Years of Priming Data"** field is located, all instrument with enough data ahead of the **"Test Start Date"** the system's indicators will be able to be primed and ready for the system to check to see if an order can be created. For instruments where there not enough data, such as when you set the **"Test Start Date"** near, or

even before the beginning of the data for any of the instruments, then each instrument will be enabled to start trading after that instrument has processed enough records to allow the the system's calculation to be primed.

The priming for each indicator is determined by the overall prime bars required by a system. The prime bars are determined by adding the max required bars for indicator priming to the max required bars to create a look-back required option that determines how many instrument records are required by a system.

For example, if you have a simple moving average indicator that uses a parameter with value 20, that means the indicator will need that many records for priming. When there are other indicator parameters that also require a number of records for look-back priming, the number of records in the next look-back requirement will add the number of records in the next parameter look-back to the previous number of look-back priming requirement. For example, with the first example needing 20-look-back records, and the second look-back example needing 10-look-back records, the system will require a total priming look-back requirement of 30 bars so the system can be primed and made ready to generate orders.

Here is an example, where a `simpleMovingAverage` is defined as needing 20 records or bars to prime this calculation. When a second indicator, then enabled look-back parameter is a lookback parameter with value 10:

```
simpleMovingAverage[ lookbackParameter ]
```

Section 3 – Test Results

Forum Questions that appear more frequently are listed in the topics of this section.

Forum Topic:	Details:
Two different results	<p>Q: I am trying to understand how it's possible to get two different results from two tests using the same parameters.</p> <p>A: There are many parameters involved, so replicating test results requires exact parameter settings, identical blox and systems, as well as identical portfolios and data sets. Normally you would need to copy the exact data set, and use the same exact suite and Trading Blox preferences settings.</p> <p>Any small change in the data, or the parameter can cause all sorts of small changes that result in a large difference. Something about butterflies.</p> <p>Based on just the results, it's not possible to predict what the difference might be, but start by looking at the first few trades in each test, and see if they start at the same time, same instruments, same position size, and same profit and loss. If not, investigate why.</p>

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 84

Section 4 – How Stops Work

There are a few different things to know about how Trading Blox handles Stops.

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) are the basic `On_STOP` or `At_LIMIT` orders. The `exitStop`, which is available to all these orders, is a protective Exit-Price Stop-Order. Let's say the broker enters long at \$11 with an `exitStop` of \$10. Should the position's price drops below \$10 on the day of entry, the position will automatically be exited. But check the Trading Blox Builder **User's Guide -> Global Suite Simulation -> Entry Day Retracement Percent Parameter** setting value for details on whether an entry day stop will be exited based on the low, or the close of the day. Note: When Protective Stops placed in this manner, they are only good for the day of entry. If you want them to create an exit price order after the day of entry, you will need to create a new protective price order. Usually, this is done in the Exit Orders Script section that only runs when the instrument has an active position. The need to create a new protective price order for each price-bar is required to make it easier to change the protective price without having to remove the previous protective price.

Are protective Stop-Price orders required?

No. An Exit Stop-Price is optional for all Broker orders. Some reversal systems are always in the market and don't use stops. However, remember that position risk calculations for each instrument are based upon the use of a protective Stop-Price order. If you have no stops, Trading Blox assumes undefined/Unlimited Risk. Using Blocks like the Fixed Fractional Money Manager, which is calculated required a risk amount to create an entry position order quantity amount. When there is no protective price, the logic in the Fixed Fractional sizing will assume unlimited risk and will assign a zero quantity to the entry order. In most cases, a zero quantity entry order will be canceled. Canceled order don't create a trading position.

My position was not exited, and its protective stop price for my order went below the exit price on the next bar.

The protective stop as placed with an entry order is saved with the instrument, but the protective exit-order only placed for the day of entry for a new position. 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`. However, you must use a new broker order every day so an active protective exit order is active to "hold" your protective exit-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. Open up their Exit Order script sections to see how they are being used.

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 Exit Orders and Adjust Stops script are the best places to keep protective exit orders active. All new Exit and Entry orders will be active for the next price bar. Protective Exit Stop-Price orders are then used in the end-of-day risk and other calculations. We usually place all entry broker orders in the entry script, and exit broker orders in the exit script, and protective price orders will also work well when updated in the Adjust Stops script section.

What if I have multiple units?

Use the same function - you can enter a different stop for each unit:

```
instrument.SetExitStop( unitNumber, stopPrice )
```

An example of how you might want to handle protective-exit price stops with multiple units is available in the blox, **Turtle Entry Exit -> Exit Orders** script section.

Section 5 – 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	Run to end of the displayed page.

Keys	Use-In	Operation:
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.

Edit Time: 9/11/2020 4:48:28 PM

Topic ID#: 567

Section 6 – TradingBlox.ini

Trading Blox Builder uses the application's default values that are listed in the TradingBlox.ini file.

For more understanding about default settings, click this next link:

[Trading Blox Default Parameter Settings](#)

Section 7 – Removed Keywords

The following **variables** have been removed:

```
entryPrice  
protectStop  
entryRisk  
unitSize  
canFillOrder  
newPosition  
canAddUnit
```

FILTER constant

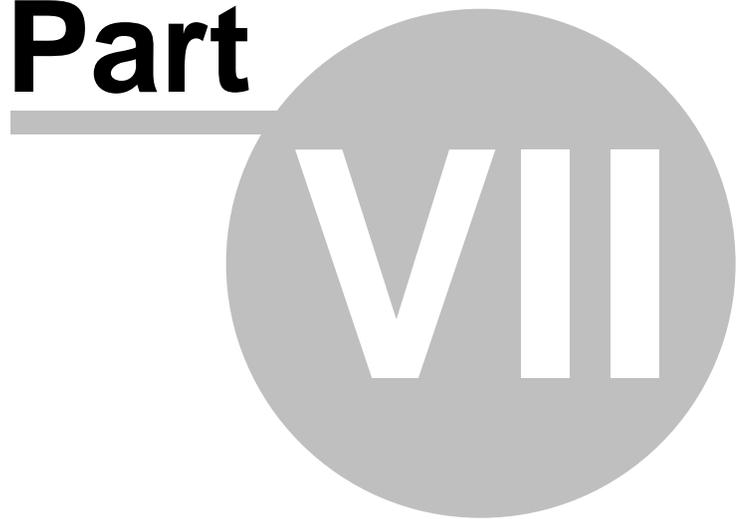
```
instrument.forexBaseRate  
instrument.forexQuoteRate  
instrument.group  
test.equityRisk (use test.currentRisk)  
system.SortOrdersByInstrumentLongRank
```

Edit Time: 3/21/2024 10:41:40 AM

Topic ID#: 723

Troubleshooting

Part



Part 7 – Troubleshooting

When a problem seems to have happened with Trading Blox Builder, there are some simple methods that often can resolve most of the types that have happened.

Click on this next link that will open your default browser and show you the **Troubleshooting** topic in the Trading Blox Builder User's Guide: [Trading Blox User's guide Troubleshooting methods:](#)

Edit Time: 3/21/2024
10:41:42 AM

Topic ID#: 728

Index

- A -

AbortParameterRun 685
AbortSimulation 1217
AbortTest 685
abortTestPending 1171
Account for Contract Rolls 1179
Account for Forex Carry 1179
addLine 685
addLineSeries 685
AllowAllTrades 994, 998
AllowLongTrades 994, 996
AllowShortTrades 994, 997
Alternate Order Object 1003, 1032, 1035, 1036, 1037, 1038, 1040, 1042, 1043, 1045, 1046, 1047, 1048, 1049, 1050, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1060, 1061, 1062, 1063, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1072
AlternateBroker 718
AlternateOrder 1003, 1032, 1035, 1036, 1037, 1038, 1040, 1042, 1043, 1045, 1046, 1047, 1048, 1049, 1050, 1052, 1053, 1054, 1055, 1056, 1057, 1058, 1060, 1061, 1062, 1063, 1065, 1066, 1067, 1068, 1069, 1070, 1071, 1072
alternateSystem 718
AND 679
ank Instrument 124
ASCII 539
ASCIIToCharacters 541
Automatic Context 350, 965
Auto-Priming 1198

- B -

Bar 674, 713
Bar Indexing 713

bars 526
barsSinceEntry 685
base currency 1179
baseCurrency 1171
baseCurrencyBorrowRate 1171
baseCurrencyLendRate 1171
Basic Money Manager 66
Before Trading Day 124
Block 90
Block Object 378
block.systemIndex 730
blockName 1032
Blox
 Working with 90
Boolean 273
Breakpoint 687
Broker
 Entry Orders 737
 Exit Orders 766

- C -

capitalAddsDrawsTotal 1171
Charge Incentive Fee 1179
Charge Management Fee 1179
Chart 176
Chart Plotting 176
CHARTNOVALUE 200, 798
clearingIntent 1032
ColorBackground 195
ColorCrossHair 195
ColorCustom1 195
ColorCustom2 195
ColorCustom3 195
ColorCustom4 195
ColorDownBar 195
ColorDownCandle 195

ColorGrid 195
ColorLongTrade 195
ColorRGB 195
ColorShortTrade 195
ColorTradeEntry 195
ColorTradeExit 195
ColorTradeStop 195
ColorUpBar 195
ColorUpCandle 195
Comments 205
Commission per Contract 1179
Commission per Stock Share 1179
Commission per Stock Trade 1179
Commission per Stock Value 1179
Commodity Channel Index 590
Common Questions 1246
 How Stops Work 1251
 Shortcut Keys 1253
 The Start and End Dates 1248
Comparison 683
Constants 200
continueProcessing 1032
Correlation Functions 905
 Add Closely Correlated 908
 Add Loosely Correlated 909
 Reset Closely Correlated 906
 Reset Loosely Correlated 907
Correlation Properties 910
currentDate 321, 1171
currentDay 292, 685, 1171
currentDrawdown 685
currentOpenEquity 685
currentParameterRun 685
currentParameterTest 685, 1171
currentPositionProfit 685
currentPositionQuantity 685
currentPositionRisk 685

currentPositionUnits 685
currentTime 322, 1171
Custom Function 1104
customSortValue 992
customValue 1032

- D -

Data Function
 Add Commission 931
 GetDateTimeIndex 940
 GetDayIndex 941
 Price Format 942
 Real Price 943
 Round Tick 944
 Round Tick Down 945
 Round Tick Up 946
Data Functions 929
dataLoadedBars 685
Date Functions 306
 DateToJulian 309
 DayMonthYearToDate 310
 DayOfMonth 311
 DayOfWeek 312
 DayOfWeekName 313
 DaysInMonth 315
 JulianToDate 317
 Month 319
 MonthName 320
 SystemDate 321
 SystemTime 322
 WeekNumberISO 325
 Year 327
dayNumber 685
Debugger 687
Defined Elsewhere in Another Block 90
deliveryMonth 685
Dema 590
DenyAllTrades 994, 1001
DenyLongTrades 994, 999
DenyShortTrades 994, 1000

Dominant Cycle 590
 Dominant Cycle Highest 590
 Dominant Cycle Lowest 590
 Dominant Cycle Phase 590
 Drawdown Reduction Amount 1179
 Drawdown Reduction Threshold 1179

- E -

Earn Dividends 1179
 Earn Interest 1179
 Ehlers Lead Sinewave 590
 Ehlers Nonlinear Ma 590
 Ehlers Sinewave 590
 Ehlers Zero Lag Ema 590
 Email Manager
 EmailConnect 865
 EmailConnectSSL 866
 EmailDisconnect 868
 EmailSend 869
 EmailSendHTML 871
 EncloseInQuotes 549
 EnterLongOnStopOpen 685
 EnterLongStopOpenOnly 685
 EnterShortOnStopOpen 685
 EnterShortStopOpenOnly 685
 Entry Day Retracement 1179
 Entry Orders
 EnterLongAtLimit 750
 EnterLongAtLimitClose 762
 EnterLongAtLimitOpen 742
 EnterLongOnClose 755
 EnterLongOnOpen 738
 EnterLongOnStop 748
 EnterLongOnStopClose 758
 EnterLongOnStopOpen 741
 EnterShortAtLimit 753
 EnterShortAtLimitClose 764
 EnterShortAtLimitOpen 746
 EnterShortOnClose 757

EnterShortOnOpen 740
 EnterShortOnStop 749
 EnterShortOnStopClose 760
 EnterShortOnStopOpen 744

Entry Scripts
 Entry Order Filled 180
 Entry Orders 166

entryRisk 1032

Equity Manager 1158

equityDrawdown 685

executionType 1032

Exit Orders

 ExitAllUnitsAtLimit 777
 ExitAllUnitsAtLimitClose 784
 ExitAllUnitsAtLimitOpen 771
 ExitAllUnitsOnClose 780
 ExitAllUnitsOnOpen 768
 ExitAllUnitsOnStop 774
 ExitAllUnitsOnStopClose 782
 ExitAllUnitsOnStopOpen 770
 ExitUnitAtLimit 778
 ExitUnitAtLimitClose 785
 ExitUnitAtLimitOpen 773
 ExitUnitOnClose 781
 ExitUnitOnOpen 769
 ExitUnitOnStop 775
 ExitUnitOnStopClose 783
 ExitUnitOnStopOpen 772

Exit Scripts

 Exit Order Filled 179
 Exit Orders 165

- F -

FALSE 200, 273, 1198

FAMA 590

feesIncentiveAccrued 1171

feesIncentiveTotal 1171

feesManagementAccrued 1171

feesManagementTotal 1171

File Functions

 CopyFile 336

File Functions

CreateDirectory 337
 DeleteFile 338
 EditFile 339
 FileExists 343
 MoveFile 359
 OpenFile 362
 OpenFileDialog 363
 SaveFileDialog 366

FileManager

Close 875
 EndOfFile 879
 OpenAppend 881
 OpenRead 883
 OpenWrite 885
 ReadLine 888
 WriteLine 890
 WriteString 892

fillPrice 1032

Filter Portfolio 124

FindString 550

Fixed Fractional Money Manager 66

Floating 273

Forex Trade Size 1179

forexDataPath 1171

FormatString 551

FRIDAY 200

futuresDataPath 1171

futuresMonth 685

- G -

General Functions

BuildDividendFiles 367
 ColorBackground 386
 ColorCrossHair 386
 ColorCustom1 386
 ColorCustom2 386
 ColorCustom3 386
 ColorCustom4 386
 ColorDownBar 386
 ColorDownCandle 386

ColorGrid 386

ColorLongTrade 386

ColorShortTrade 386

ColorTradeEntry 386

ColorTradeExit 386

ColorTradeStop 386

ColorUpBar 386

ColorUpCandle 386

FileVersion 367

FileVersionNumerical 367

GetRegistryKey 367

License Name 377

LicenseName 367

LineNumber 367

LoadUnadjustedClose 386

LoadVolume 386

Message Box 380

MessageBox 367

NumberOfExtraDataFields 386

PlaySound 367, 385

Preference Items 386

ProcessDailyBars 386

ProcessMonthlyBars 386

ProcessWeekends 386

ProcessWeeklyBars 386

ProductVersion 367

ProductVersionNumerical 367

RaiseNegativeDataSeries 386

SetRegistryKey 367

YearsOfPrimingData 386

generatingOrders 685

GetField 560

GetFieldCount 562

GetFieldNumber 564

GetNumberField 537, 560

GetStringField 537, 560

Getting Started 2

Adding Money Management 66

Creating a System 4

Tutorial 4, 66

What are blox? 3

Global Parameters 1158, 1179

Global Suite System 718, 1116

group 722
 Group Properties 947
 GSS 718, 1116, 1194

- H -

Historic Volatility 590
 Historical Trade Properties 949

- I -

IBCancelOpenOrder 1011
 IBConnect 1011
 IBDisconnect 1011
 IBOOrderNumber 1011
 Ignore Test Positions 1179
 Increment Test Start 1179
 index 1144
 Indexing 713
 Indicator 176
 Indicators 658

- Accessing 672
- Basic 660
- Calculated 663
- Custom 670
- Invalid Items 663
- Valid Items 663

 Instantaneous Trendline 590
 Instantaneous Trendline Alternate 590
 Instrument 273
 Instrument Data Properties 221
 Instrument Loading 350, 951, 965

- Load By Long Rank 956
- Load By Short Rank 957
- Load External Data 958
- Load Symbol 952

 Instrument Type 894
 instrument.inPortfolio Property 916
 instrumentCount 685, 1171

instrumentExists 722
 instrumentList 1171
 Integer 273
 IPV 160, 162, 165, 166, 169, 176, 178, 179, 180,
 181, 183, 184, 185, 187
 isBuy 1032
 isEntry 1032
 isWholeExit 1032

- K -

Kaufman Adaptive Moving Average 590
 keyword 713
 Keywords 713

- Abort 702, 1186, 1187
- Assert 702
- Break 702
- DO 699
- ELSE 706
- End 1186, 1187
- ENDIF 706
- ENDWHILE 711
- FOR 703
- IF 706
- LOOP 699
- NEXT 703
- Stop 702, 1186, 1187
- THEN 706
- UNTIL 699
- WHILE 699, 711

- L -

Laguerre Moving Average 590
 LeftCharacters 565
 Left-to-Right Data-Type Counting 1078
 Len 574
 Leverage 1171, 1179
 limitPrice 1032
 LineNumber 378
 Load IPV From File 350, 965

LoadPortfolioInstrument 685
 LoadSymbol 685
 LoadUnadjustedClose 200
 LoadVolume 200
 Local 90
 LONG 200
 LowerCase 566

- M -

MAMA 590
 Management Fee 1179
 marketOrdersValue 1151, 1171
 Mathematical Comparison 683
 Mathematical Functions 393
 Absolute Value 395
 Arc Cosine 396
 Arc Sine 397
 Arc Tangent 398
 Arc Tangent XY 399
 Average 400
 CAGR 402
 Correlation 405
 Correlation Log 407
 Cosine 409
 Degrees to Radians 410
 ema function 411
 Exponents 412
 Hypotenuse 415
 IfThenElse 416
 IsUndefined 417
 Log 418
 Max 419
 Min 420, 426
 Radians to Degrees 422
 Random 423
 RandomDouble 424
 RandomSeed 425
 Round 426
 Sine 430
 Square Root 431
 Standard Deviation 431
 Standard Deviation Log 434
 sumValues 436
 Tangent 438
 Max Percent Volume Per Trade 1179
 MiddleCharacters 567
 Minimum Futures Volume 1179
 Minimum Slippage 1179
 Minimum Stock Volume 1179
 mod 679
 Momentum 590
 MONDAY 200
 Money 273
 Money Manager Scripts
 Unit Size 169
 Multi-Money Manager 66

- N -

name 722, 1171
 ncentive Fee 1179
 noStopPrice 1032
 NOT 679
 NumberOfExtraDataFields 200

- O -

Objects 713
 alternateBroker 732
 AlternateOrder 1003
 alternateSystem 1116, 1194
 Block 722
 Broker 732
 Email Manager 863
 FileManager 873
 Instrument 894
 Name 722
 Order 1003
 Script 1073
 ScriptName 722
 System 1116
 Test 1157

On Balance Volume 590
 openEquity 685
 OpenOrderMargin 1151
 Operators 679
 OR 679
 Order Context 1134
 Order Functions
 Reject 1013
 SetClearingIntent 1011
 SetCustomValue 1016
 SetFillPrice 1017
 SetLimitPrice 1011
 SetOrderReportMessage 1020
 SetQuantity 1022
 SetRuleLabel 1024
 SetSortValue 1026
 SetStopPrice 1029
 SetTimeInForce 1011
 Order Generation Equity 1179
 Order Generation Equity High 1179
 Order Object 1134
 Order Properties 1032
 orderExists 722
 orderGenerationBar 685, 1171
 orderGenerationTest 1171
 orderPrice 1032
 orderReportMessage 1032
 orderReportPath 1171
 Orders
 Entry Orders 737
 Exit Orders 766
 orderType 1032
 OtherExpense 1158
 OUT 200

- P -

Pay Margin on Stocks 1179
 Percent 273
 Percent R 590
 Percent Rank 590
 PI 200
 Placing Orders 732
 Plotting 176
 Portfolio Manager 124
 Portfolio Manager Scripts
 Filter Portfolio 162
 Rank Instruments 160
 position 1032
 Position Functions 974
 Set Exit Limit 975
 Set Exit Stop 976
 Set Unit Custom Value 978
 Position Properties 981
 positionInstruments 685
 Positions & Orders Report 1217
 Positon Adjustment Functions 788
 Adjust Position At Limit 793
 Adjust Position On Close 790
 Adjust Position On Open 791
 Adjust Position On Stop 792
 Preferences 200, 1237
 Price 273
 primeStart 1171
 Priming 176
 priorityIndex 685
 ProcessDailyBars 200
 processingMessage 1032
 ProcessMonthlyBars 200
 ProcessWeekends 200
 ProcessWeeklyBars 200
 Purchase Equity 1158

ParameterCount 1104

Parameters 229

- Q -

quantity 1032

- R -

RaiseNegativeDataSeries 200

Range 590

Rank Instrument 124

Ranking Functions 986

 Set Long Ranking Value 988

 Set Short Ranking Value 990

Ranking Properties 992

rankInstruments 685

Rate of Change 590

referenceID 1032

Registry Keys

 GetRegistryKey 376

 SetRegistryKey 389

RemoveCommasBetweenQuotes 568

RemoveNonDigits 570

ReplaceString 571

resultsReportPath 1171

RightCharacters 573

Risk Manager Scripts

 Adjust Instrument Risk 185

 Can Add Unit 169

 Can Fill Order 178

 Compute Instrument Risk 184

 Compute Risk Adjustment 184

 Initialize Risk Management 184

Roll Slippage 1179

RTrim 576

ruleLabel 1032

- S -

SATURDAY 200

Scope 90

script.SetStringReturnValue 685

scriptName 722, 1032

Scripts 148

 After Instrument Day 187

 After Simulation 188

 After Test Script 187

 After Trading Day 187

 Basic Scripts 107

 Before Instrument Day 162

 Before Simulation 150

 Before Test 160

 Before Trading Day 162

 Entry Blox Scripts 126

 Exit Blox Scripts 127

 Money Manager Blox Scripts 127

 Portfolio Manager Blox Scripts 124

 Risk Manager Blox Scripts 128

 Update Indicators Blox Scripts 129

 Working with 104

Selector 273

Sell Stock Split Remainder 1179

SendToFXCM 1032

Series 273, 292

Set Test Duration 1179

SetAlternateSystem 718

SetCustomSortValue 992

SetReturnValue 685

SetSeriesColorStyle 195

SetSeriesSize 510

SetSeriesValues 512

SetSmartFillExit 1217

SetStringReturnValue 685

setxAxisLabels 685

SHORT 200

Simons Historic Volatility 590

Simulation 90

Simulation Loop

 Comprehensive 113

- Slippage 1179
 - Smart Fill Exit 1179
 - smartFillExit 1171, 1217
 - sortInstruments 685
 - sortValue 1032
 - Standard ASCII Table 545
 - StartingEquity 1158
 - Statements 697
 - Assignment 698
 - DO 699
 - Error 702
 - FOR 703
 - IF 706
 - Print 708
 - VARIABLES 273
 - WHILE 699, 711
 - Stepping Priority: 12
 - stockDataPath 1171
 - stopPrice 1032
 - STRING 273, 537, 588
 - String Functions
 - ASCII 539
 - ASCIIToCharacters 541
 - Concatenate 537
 - FindString 550
 - FormatString 551
 - GetField 560
 - GetFieldCount 562
 - GetFieldNumber 564
 - LeftCharacters 565
 - LowerCase 566
 - MiddleCharacters 567
 - RemoveCommasBetweenQuotes 568
 - RemoveNonDigits 570
 - ReplaceString 571
 - RightCharacters 573
 - StringLength 574
 - TrimLeftSpaces 575
 - TrimRightSpaces 576
 - TrimSpaces 577
 - ucase 578
 - StringLength 574
 - stringParameterCount 1104
 - summaryResultsPath 1171
 - SUNDAY 200
 - symbol 1032
 - symbolWithType 1032
 - System 90, 722
 - Correlation 448
 - Correlation Log 450
 - Portfolio Instrument Access 1125
 - System Functions 1122
 - System Properties 1144
 - systemBlockName 1032
 - systemCount 1171
 - systemIndex 722
 - systemName 1032
 - Systems
 - Working with 86
- T -
- Tema 590
 - Test 90
 - CreateStringArray 1222
 - Equity Properties 1158
 - General Properties 1171
 - GetStringArrayElement 1222
 - SetStringArrayElement 1222
 - SortStringArray 1222
 - String Array Functions 1222
 - Test Statistics 1237
 - Trade Properties 1241
 - Test Functions 1183
 - Abort Simulation 1186
 - Abort Test 1187
 - Add Statistic 1188
 - GetSteppedParameter 1191
 - Set Alternate System 1194
 - Set Auto Priming 1198
 - Set Generating Orders 1209
 - SetSilentTestRun 1216
 - Update Other Expenses 1235

Test Starting Equity 1179
testEnd 1171
testStart 1171
testTotalEquity 685
Thread Count 1179
threadCount 1171
threadIndex 1171
THURSDAY 200
Time Frame 674
Time Function
 Hour 315
 Minute 318
Time Functions
 TimeDiff 324
timeIncrement 1171
timeInForce 1032
timeStamp 1171
today 685
totalEquity 685
totalInstruments 685
TotalMargin 1158
totalParameterRuns 685
totalParameterTests 685, 1171
totalPositionProfit 685
totalPositionRisk 685
totalPositions 685
totalPositionSize 685
totalUnits 685
Trade Always on Tick 1179
Trade Control Functions 994
 Allow All Trades 998
 Allow Long Trades 996
 Allow Short Trades 997
 Deny All Trades 1001
 Deny Long Trades 999
 Deny Short Trades 1000
Trade Control Properties 1002
Trade Futures on Lock Day 1179
TradeLog 1217
tradeOrder 685
Trading Blox 119
 Auxiliary Blox Reference 129
 Entry Blox Reference 126
 Exit Blox Reference 127
 Money Manager Blox Reference 127
 Portfolio Manager Blox Reference 124
 Risk Manager Blox Reference 128
Trading Blox Architecture
 Blox 80
 Indicators 80
 Parameters 80
 Process Flow 111
 Scripts 80
 Simulation Loop 112
 Suites 80
 Systems 80
 Trading Objects 80
 Units 80
 Variables 80
 Working with Systems, Blox and Scripts 84
Trading Equity Base 1144, 1179
Trading Objects 713
tradingBars 685
TradingBlox.ini 176
Trend Vigor 590
Trim 575, 577
TrimLeftSpaces 575
TrimRightSpaces 576
TrimSpaces 577
TRUE 200, 273, 1198
True High 590
True Low 590
True Range 590
TUESDAY 200
TYPE 273, 586
Type Conversion Functions 579
 AsFloating 580
 AsInteger 581

Type Conversion Functions 579
 AsString 543, 584
 IsFloating 586
 IsInteger 587
 IsString 588

- U -

UCase 578
 UNDEFINED 90, 200
 unitBarsSinceEntry 685
 unitCustomValue 981
 unitNumber 1032
 Update Indicators Scripts
 Update Indicators 176
 UpperCase 578
 Use Broker Positions 1179
 Use Capital Adds Draws 1179
 Use Pip-Based Slippage 1179
 Use Start Date Stepping 1179
 Used for Lookback 1198
 User's Guide Generation Orders 1003

- V -

VADI 1158
 Variable Name Rules 208, 221
 Variables 202
 Average 400, 445
 Block Permanent Variables 208
 Cross Over 458
 GetSeriesSize 464
 Highest 465
 HighestBar 467
 Instrument Permanent Variables 221
 Lowest 469
 LowestBar 471
 Median 475
 Naming Variables 262
 RegressionEnd 476
 RegressionSlope 478

RegressionValue 480
 RSI 482
 Scope 265
 Series Functions 400, 431, 434, 439, 445, 458, 464, 465, 467, 469, 471, 475, 476, 478, 480, 482, 497, 509, 512, 516, 519, 522, 524, 526, 528, 530, 532, 534
 Series Indexing 460
 SetSeriesColorStyle 497
 SetSeriesSize 509
 SetSeriesValues 512
 SortSeries 516
 SortSeriesDual 519
 Standard Deviation 431, 522
 Standard Deviation Log 434, 524
 Sum 526
 Swing High 528
 Swing High Bars 530
 Swing Low 532
 Swing Low Bars 534
 The VARIABLES Statement 273

- W -

Walk Forward Optimization Days 1179
 Walk Forward OSS Days 1179
 walkForwardStatus 1171
 WEDNESDAY 200
 Week 674
 Weighted Moving Average 590

- X -

xAxis 685

- Y -

YearsOfPrimingData 200
 yesterday 685

- Z -

ZScore 590