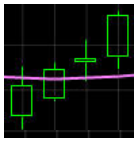
StockFinder®

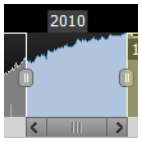
RealCode Reference

2nd Edition

StockFinder 5

Updated February 2010





WORDEN

REALCODE REFERENCE

Worden Brothers, Inc.
www.Worden.com

Five Oaks Office Park
4905 Pine Cone Drive
Durham, NC 27707

REALCODE REFERENCE

www.Worden.com
Worden Brothers, Inc,
Five Oaks Office Park
4905 Pine Cone Drive
Durham, NC 27707
Customer Service: 1-800-776-4940
Technical Support: 1-919-408-0542
Fax: 1-919-408-0545

Table of Contents

What's new for RealCode in StockFinder version 5.0

For StockFinder version 5.0 there are many exciting changes to the RealCode language. One of the major changes to version 5 is the ability to import indicators and conditions into your code directly from the Library without first adding them to a chart. Additionally you can un-link indicators and condtions that you have referenced on a chart. This allows you to save your indicators and conditions even if they reference other items on the chart. This is a giant leap forward for RealCode developers as you no longer need to save a Chart if your code references other indicators/code. This also means you no longer need an indicator or condition to exist on the chart for you to call it in your code. You can now import indicators and conditions into your code directly from the Library without first adding them to the chart.

Version 5 adds some additional child functions. Child functions, like moving average and stochastics can be applied to Price, Volume, or any imported indicator. They can also be chained together to produce child of child calculations. For TeleChart users these will feel more like the PCF syntax that is available in TeleChart. For more information see Chapter 5.

Some additional TimeFrame (now called Bar Interval) functions have been added to help you determine the active bar interval of the chart. See Chapter 5.

You can now reference price data for any symbol and/or bar interval from directly within your RealCode. This allows you to compare the active symbol with another, or the active bar interval with a different one.

Additionally, you can now discover the context in which your code is running allowing you make programming decisions based on a set of conditions. When running on a chart, the chart start and end date are available as well as the number of bars visible. There is also a new directive to instruct RealCode to re-calculate when the zoom or scroll changes on the chart. When running as a market indicator (index) there are new properties to define the start and end of the list as well as the number of items in the list. For more information, see chapter 8.

For advanced users, there is now a global memory object you can access to share data with other RealCode calculations.

The RealCode help and code-complete engine has been upgraded to have improved functionality and to document more of the functions that are available to you. Integrated F1 help directly in the editor will open a new RealCode API class reference. The class reference is a companion to this document and should be used to reference all of the classes and functions available to you.

There are also more places to use your RealCode in StockFinder 5. With the changes to Scans and Sorting in StockFinder, your conditions and indicators can be used much more easily outside the Chart itself.

In addition to the actual changes in StockFinder and RealCode, there has also been a terminology change. Rules are now called Conditions. This will not effect any previous source code but all references to Rules from previous documentation is now the equivilant of Conditions in version 5.0.

Chapter 1- Introduction

This book introduces the RealCode™ programming Language. RealCode is based on the Microsoft Visual Basic.Net framework and uses the Visual Basic (VB) language syntax. RealCode is compiled into a .net (pronounced dot net) assembly and run by the StockFinder application. Unlike the scripting languages that some other applications use, RealCode is fully compiled and runs at the same machine language level as the StockFinder application itself. This gives you unmatched performance, running at the same speed as if we (the developers of StockFinder) wrote the code ourselves, with the flexibility of "run-time development and assembly" that you get by writing your own custom code (Whew, that's a mouthful!),

This book covers only the RealCode™ language and implementation and does not cover the basics of the StockFinder application. It is assumed you understand how to run the program, add indicators, create charts and scans and save or load your Layouts.

This book does not fully cover all the classes and functions available to you via the RealCode API or the .net 3.5 framework. The full RealCode class api is available online at http://www.stockfinder.com/realcodeapi/ and contains all the properties and methods you can call from your code.

The first part of this book covers the introduction to programming and covers the basics of VB.net development. If you're new to programming or new to VB.net this chapter will go over the fundamental data types, constructs and flow control needed to program in VB.net It is by no means a complete reference for VB.net. If you are new to software development or to programming in general, and you wish to get into deeper RealCode programming, I highly recommend getting a few VB.Net books and try some programming outside of StockFinder. For basic RealCode calculations this book should be enough to get you started.

If you're an experienced developer and/or understand the concepts of variables, flow control, conditionals and loops you can skip over the introduction to VB and move on to the RealCode fundamentals.

THE CODE EXAMPLES PROVIDED IN THIS BOOK ARE FOR EDUCATIONAL PURPOSES ONLY. THE EXAMPLES IN NO WAY SUGGEST AN ENDORSEMENT, TRADING STRATEGY OR GUARANTEE ANY SORT OF RETURN ON INVESTEMENT. THE CODE EXAMPLES ARE PROVIDED AS-IS.

About the Author

Ken Gilb is the Chief Software Engineer for Worden Brothers, Inc. His work at Worden Brothers includes "big picture" architectural overview as well as "down and dirty" code writing. He's a self proclaimed code monkey. Quote "I write code and I love it".

Additional Resources

The RealCode Class Reference API is available in the RealCode Editor (via the help button) or online at http://www.stockfinder.com/realcodeapi/

In addition to this User Guide and the Class Reference, you can use many existing Microsoft Visual Basic.Net references, Websites, blogs and forums.

Microsoft Online References:

- MSDN: http://msdn.microsoft.com – Microsoft Developers Network. API reference, training, examples.
- Microsoft VB.net new to Development: http://msdn2.microsoft.com/en-us/vbasic/ms789097.aspx

Worden Brothers, Inc

- Class Reference API: http://www.stockfinder.com/realcodeapi/
- Worden website: http://www.Worden.com – market commentary, discussion forums.
- StockFinder website: http://www.StockFinder.com - Videos & manuals
- Worden Discussion Forum: http://www.worden.com/training - Active forums with Worden Trainers, Worden developers and customers.

What types of RealCode can you create?

You can create the following RealCode items:

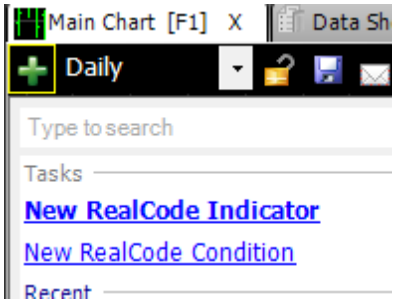
- Indicators (plots on chart or WatchList column)
- Conditions (true/false Conditions to sort, scan, filter, color or backtest)
- Paint Brush (indicator coloring based on code)
- Scans (via RealCode Conditions or Indicators applied to a Scan)
- WatchList Condition Lights (via RealCode Conditions or Indicators applied to a WatchList)
- Sorts (via RealCode Indicators applied to a WatchList column)
- BackScanner entry or exit Condition. (RealCode Condition applied to entry/exit Condition)

Indicators, Conditions and Paint Brushes are all created for and owned by a Chart. Indicators and Conditions can also be added to any WatchList to perform a scan, sort or filter. You can treat a RealCode indicator or condition like any other indicator or condition in that it can be used in other conditions, List Calculations, Scans, Sorts, back testing etc. Anything you can do with a regular indicator/condition in StockFinder can also be performed with a RealCode Indicator /condition.

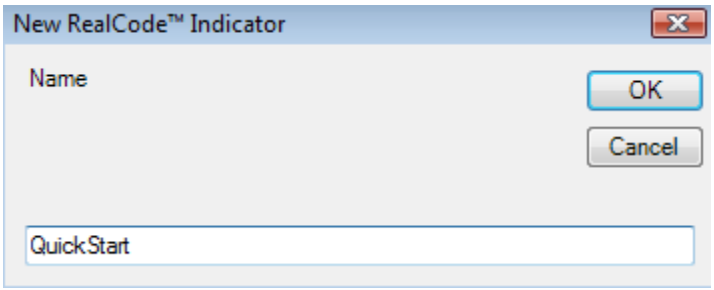
Quick Start

Nothing beats a code example, so let's start with something basic. We're going to make a RealCode indicator that will plot the closing value of a security (stock symbol). Then we'll modify it to show you the high, the net change and finally the percent change.

1. Click on the *Add Indicator +* button at the top-left of a chart.
2. Click on *New RealCode Indicator*



3. Type *Quickstart* as the name for the new RealCode indicator and click Ok



4. Type the following line of code into the editor

```
Plot = Price.Close
```

5. Click *Apply*
6. Click *Ok* to close the editor.

You should now have an indicator on your Chart that represents the closing prices for the selected symbol (the data will depend on the symbol selected). Let's modify the code to plot the High of the day instead of the close. Right click on your RealCode indicator and choose *Edit RealCode.*

Change the code in the window from *Price.Close* to *Price.High.* You should now have a line that looks like this:

```
Plot = Price.High
```

Click *Apply* and *Ok* to close the editor window. Your plot is now showing you the high of the selected symbol for every date on the chart. Let's modify the code one more time, but instead of

showing the high or close, we're going to show the daily net change. Right click on your indicator and choose *Edit RealCode.*

Change the existing code in the editor to the following:

```
Plot = Price.Close – Price.Close(1)
```

Click *Apply* and close the editor window. Your indicator is now showing you the daily net change for the selected symbol. In the code above `Price.Close` is equal to the close for the currently calculating bar or the close today for the latest bar. `Price.Close(1)` is the previous day's close, or the close for yesterday for the latest bar. By subtracting the current close by the previous day's close, we get the net change or the amount the stock went up or down from the previous day.

We could also have used the built in function *NetChange* to perform the same calculation. Right-click on your RealCode indicator and choose *Edit RealCode*. Change the existing code to:

```
Plot = Price.NetChange
```

Hit *Apply* and close the editor. You will notice your plot doesn't change from the previous example as they both return the current bar net change of price.

Let's make one last change and we'll be finished with this lesson. Right click on your indicator and choose, *Edit RealCode*. Change the code in the editor to:

```
Dim netChange As Single = price.close - price.close(1)
Dim percentChange As Single = netChange / price.close
plot = percentChange * 100
```

We've broken the code up into 3 lines for clarity; you could also type the above as:

```
plot = ((price.close - price.close(1)) / price.close)  * 100
```

Both are mathematically the same but the 3 lines reads a bit easier, so let's use this for our example. On line one we store the net change calculation into a *variable* named `netChange`. A variable is simply the place in memory to store data (or in this case a specific numeric value). For now, let's ignore the rest of the syntax on this line, but you can note that the `netChange` variable has the same formula as the first example assigned to it.

On the second line, you'll notice we divide our previously calculated value (`netChange`) and divide it by the current closing price. This gives us the percent change and we store this in a variable named `percentChange` (how appropriate!). At this point in time we could type `Plot = percentChange` on a new line and return the current value, but this would be a decimal number (0.0 to 1.0) . Most people like to think of percentages as values from 1-100 so we simply multiply the `percentChange` variable by 100 and set that to the Plot output. (the `*`symbol is multiply)

The previous code was to introduce you to the math and programming concepts involved with RealCode. If you want to calculate the net change or percent change in RealCode, you can simply call the netChange or percentChange functions on price:

```
Plot = Price.NetChange()
```

```
Plot = Price.PercentChange()
```

That's the end of our quick start, if you're lost, don't worry we'll go into more detail about what is going on in the next few chapters. If you're new to programming and programming concepts or you simply want a primer on programming with Visual Basic.net, read on to Chapter 2. If you understand variables and conditionals and want to dive in to the meat, skip ahead to Chapter 3.

Chapter 2 - A Visual Basic.Net Programming Primer

If you are not familiar with VB.net or with programming in general, the next few sections will go over basic programming principals. This is by no means a substitute for a Visual Basic.Net book or online reference but should give you enough basic information to get started in RealCode.

Fundamental Data Types

All programming deals with data: generating data, manipulating data, and consuming data (or some combination of the three). The most common and fundamental data types you will deal with in RealCode are numbers (integer, single, decimal) text (string, char), Booleans (true/false), Colors (pretty colors) and Dates (not an evening out on the town, calendar dates). There are many more data types available in the .net framework (hundreds of them. Look at the size of that thing, its huge!). But for the basic understating of RealCode you will need to know the following:

- Integer (whole numbers, like parameter values) example: 1,2,3
- Single (fractional numbers like stock prices) example: 1.25, 55.73
- Boolean (true or false, used for Conditions) example: True, False (brilliant!)
- Date (date and time) example: 5/12/1975 13:30:06
- String (text) example: “Hi”, “Hello!”, “StockFinder is fun”
- Color (duh) example: Color.Red, Color.Lime, Color.Coral

Identifiers (variables)

Identifiers are unique names you give to variables of a specific data type. Variables are placeholders in memory for specific types of data. If you remember you high school Algebra then you already understand variables in formulas like the Pythagorean Theorem ($a_2 + b_2 = c_2$) where a, b and c are all variables. In VB.net we declare variables of a specific type (integer, single, string, color) to tell the computer what type of information we want to store in the variable. To declare a variable in VB.net we use the `Dim` and `As` keywords along with the variable name (identifier) and data type. Variable names must not contain a space but may contain mixed case alpha numeric symbols and underscores. Examples:

```
Dim averagePeriod as Integer
Dim net_Change as Single
Dim stockSymbol as String
```

In the above example we have declared 3 variables: `AveragePeriod`, `NetChange` and `StockSymbol`. By default, they do not contain any value, or rather, they contain a default value. Numeric values are always 0 by default while the String data type contains a special value called `Nothing`[1]

VB.net is not a case sensitive language, so naming a variable averagePeriod can also be typed as AveragePeriod or AVERagePeRiOd (in case your caps lock key is acting funny) anywhere else in code and it represents the same variable that you declared with the `Dim` keyword. Variable names must be unique (you cannot have two variables with the same name).

[1] Nothing is the equivalent of Null in other programming languages like C++ or C#

Conditions are used to branch your code down one path or another. In VB.Net you use a combination of keywords: `If`, `then`, `else`, `elseif`, `end if`. Example:

```
If expression then
   Do something
Else
   Do something else
End if
```

So, for example, if you wanted to test if an integer variable named `period` is greater than zero you would use the following:

```
If period > 0 then
   Plot = 1
Else
   Plot = -1
End if
```

The `ElseIf` keyword lets you test another condition in your If/Then statement:

```
If period > 0 then
   Plot = 1
Elseif period < 0 then
   Plot = -1
Else
   Plot = 0
End if
```

You can have as many `ElseIf` conditions before your final Else branch.

If you have many `If/Elseif` statements to compare you can keep your code looking clean and easier to read by using a `Select Case` statement:

```
Select Case period
   Case < -5
        … do something…
   Case > 5
        … do something…
   Case 0
        … do something…
   Case else
        … do something…
End Select
```

There are a few different types of operators in VB.net, but the most basic ones you will deal with are the mathematical and logical operators. The mathematical operators are:

+	Addition
-	Subtraction
*	Multiplication
/	Floating point division (returns a fractional value)
\	Integer division (returns a whole number)
Mod	Modulus division (returns the remainder)
^	Exponent (to the power of)
=	Assignment and comparison operator
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
&	String concatenation (combines two strings)

Logical operators are used to test if a statement's conditions are true or false. They are used to combine multiple conditions in If/Then statements:

- `And` - both conditions must evaluate to True
- `AndAlso` – if the first condition is false, it will not evaluate the second condition.
- `Or` - One of the 2 conditions must evaluate to True.
- `OrElse` – if the first condition is true it will not evaluate the second condition
- `XOr` - results to True ONLY if 1 of the 2 conditions is true
- `Not` - results to True if the conditions result to False

Logical operators are used in If then statements like so:

```
If price.close > price.high and price.close > price.open then
      ….
End if
```

The equal sign (=) performs two roles. It is used as an assignment operator, (to assign a value to a variable) and it is also used as a comparison operator (to test if two values are equal). When used as a comparison operator (for example in an IF statement) it will return a Boolean value. When used as an assignment operator, it can be combined with other math operators to use the target of the assignment in the calculation. For example: `Count += 1` is the equivalent of: Count = Count + 1

All math calculations follow the order of operations (parenthesis, exponents, multiplication, division, addition, subtraction) so you if you need to perform some addition or subtraction before a multiplication you can group your calculation with parenthesis like so:

Dim X as Single = (5 + 5) / 10

Looping

Looping is at the heart of every RealCode class. Behind the scenes we're actually performing a loop for you before we call your code. If you need to perform your own loops you could use one of the following statements:

For Each Looping Statement:

```
For each item as string in ListOfItems
   … do something…..
Next
```

For Next Loop:

```
For i as integer = 1 to 100
    … do something 100 times
Next i
```

Do While Loop:

```
Do
… some operation
While someVariable = true
```

Performing a loop can become an expensive operation. Since your code is called for every bar of the calculation, your loop will perform x times the loop count. So if you're loop is 1 to 100, you're going to perform 100xnumber of bars. This can really slow down your calculation. If you notice a calculation taking significant amount of time, you may wish to try to "unroll" the loop or cache your data and perform the loop calculation on the first or last bar. See the Cyclical Charts example of caching data to limit the number of looping operations performed.

Arrays

Arrays are a special data type that is a container of multiple data elements. You can think of an array as a bucket of data that stores 1 to x number of items in the variable. Each item is referenced by a numeric index. Think of an array as a spreadsheet with one column and many rows of data. The rows are accessed by selecting the row number (array index). Most tabular data is stored in arrays. All pricing and volume data along with every other indicator or condition in the system is stored in multiple arrays.

An example of an array of strings would be:

```
Dim myArray(3) as String
```

The above array myArray will hold 4 values. Arrays always start at index 0, so when defining an array in VB.net you define the upper bound of the index or in the case above, 3.

Confused? Let's try this approach. You want to store the four following strings in an array: "Rubber","Baby","Buggy","Bumpers". Your array will have rubber at index 0, Baby and index 1, buggy at

index 2 and bumpers at index 3. 0-3 is four elements. 3 is the upper bound of your array so you declare it with 3. The other way you can think of this is: arrays bounds are declared as one less than the number of elements in the array. 4 elements minus 1 element = 3.

```
Dim MyArray(3) as string
myArray(0) = "Rubber "
myArray(1) = "Baby "
myArray(2) = "Buggy "
myArray(3) = "Bumpers"

for each item as string in myArray
   debug.writeline (item)
next
```

The result of the above code would be: Rubber Baby Buggy Bumpers.

Reserved Words and Special Characters

VB.net has many other reserved words (words you cannot use as a variable name) and other special characters that mean different things. The list is too many to go into this overview. A good beginning vb.net book or some references on the web should lead you down the many more options available to you.

One special character to note is the '(single quote apostrophe) character. This starts a comment and the compiler ignores any text on the same line after this character. It is good for adding documentation to your code (documentation in code is good for when you review your code 3 months later and try to remember what the heck your code is doing. Do it for any complex calculation or for anything ambiguous).

Another set of keywords that are useful is `Exit Function`. Adding these two words to your code will stop evaluating the remaining lines of code for the current bar and exit your RealCode immediately. It will then advance to the next bar and call into your code again. This is helpful when you do not wish to wrap your code into a bunch of `If` statements to isolate specific conditions that would normally cause all other calculations to cease.

Chapter 3 - RealCode Fundamentals

RealCode is the name of the StockFinder programming language and is based on the Microsoft Visual Basic syntax. At its core, RealCode is a compiled Visual Basic.net Class. Classes are modular code that can be assembled with other classes to create a new algorithm and perform a calculation.

All RealCode items inherit a special class `BaseScriptingTemplate`. Using the RealCode API Reference you will notice that `BaseScriptingTemplate` inherits another class named `Block`. Blocks are compiled code that are connected together in a block diagram to perform a calculation. Every plot, condition, scan or sort in the StockFinder application is performed through a series of Blocks in a Block Diagram.

When you write RealCode, StockFinder compiles it for you and creates a block diagram to perform your calculation. The diagram is always visible to you (simply Edit your RealCode item and click on the *block diagram* link) but you really don't need to know the details of what happens in a diagram to understand how to write effective RealCode. The Block diagram is simply the plumbing for getting the data into and out of your calculations.

All you really need to know is that your code is compiled (which means it's fast. Have I mentioned the word *compiled* enough?) and that the .net framework and all the hundreds of pre-built classes and methods are available to you in your RealCode. If you're familiar with .net you can access any of the namespaces by simply typing them into your code editor[2] .

Whether creating a RealCode Indicator, Condition or Paintbrush, you need to understand how StockFinder evaluates your code to create your calculation. There are two ways to create a RealCode item. The traditional (and easier way) is to write the body of a function (method) that is called for every plot, color or condition that will appear on the chart. An advanced way, called RealCode classes, lets you write an entire class (not just a single function) to perform your calculation. RealCode Classes are covered in chapter 7.

The Price Chart:

All RealCode is based on the Chart that owns (displays) the RealCode. If your code is not running on the chart (say it is only being listed in the WatchList) then your code runs in the "theoretical" chart that would be displayed if your indicator were on a chart.

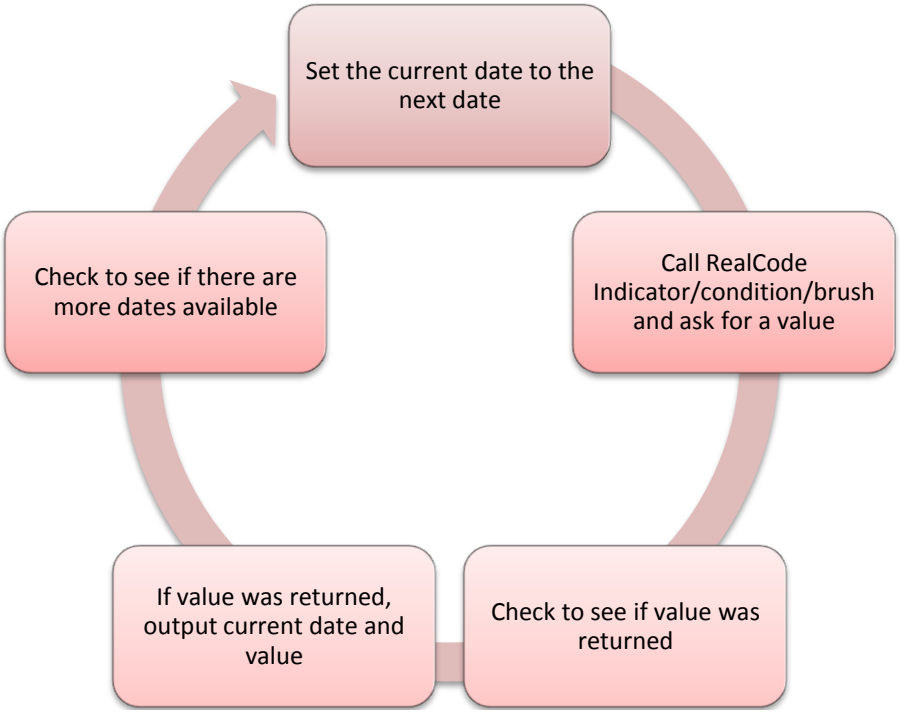
When StockFinder calls your RealCode it prepares all the Pricing and Volume data for the current symbol and the selected bar interval.

Starting at the oldest date "bar" (Open/High/low/Close values) for the selected bar interval, StockFinder calls your RealCode asking you to provide a value for the current bar. It then stores the resulting value and advances to the next date, calling your code once more for the second value. This repeats until the last date of data and it then constructs the appropriate line, paintbrush or scan based on the values you set in code.

[2] System.IO and System.Net are excluded for security.

Essentially every bar (date) will become the current bar as it loops through the Pricing and Volume Data. Because each bar is defined by the selected Bar Interval the "current" bar could be a day, month, year, hour or minute or any other custom bar interval provided. This means if your Chart or WatchList column is set to 1 minute each price bar represents one minute of trading data and your code will be called for a value for every minute of data starting with the oldest and ending with the latest.

The cycle below shows the context that your RealCode is calculated:



The main point you need to understand when writing RealCode is that your code is executing for the "Current Bar". Current Bar will represent a date on the Price history. All functions and calculations you perform are in relation to the current bar. If you need the previous close value (for say a net or percent change calculation) you would ask for data 1 bar ago. Most of the functions available to you in RealCode allow you do specify what bar (ago) to return.

The series of tables below shows how StockFinder calls your RealCode. The top row of the table shows the dates for a specific price chart, The second row shows the stock price for each date. The third row represents the values your RealCode will return.

	Price					
Date	4/1/2009	4/2/2009	4/3/2009	4/4/2009	4/5/2009	4/8/2009
Close	• 51.70	• 52.37	• 57.80	• 56.78	• 59.07	• 69.80
RealCode Value	• ?	• ?	• ?	• ?	• ?	• ?

Before we call your code, you have no values set

	Price					
Date	4/1/2009	4/2/2009	4/3/2009	4/4/2009	4/5/2009	4/8/2009
Close	• 51.70	• 52.37	• 57.80	• 56.78	• 59.07	• 69.80
RealCode Value	• 75	• ?	• ?	• ?	• ?	• ?

After we call your code for the first time, your value is set for the first bar of the indicator

	Price					
Date	4/1/2009	4/2/2009	4/3/2009	4/4/2009	4/5/2009	4/8/2009
Close	• 51.70	• 52.37	• 57.80	• 56.78	• 59.07	• 69.80
RealCode Value	• 75	• 77.12	• 78.04	• 77.89	• ?	• ?

We continue to call your code until all the values have been set for each date of the price chart.

Each of the 3 RealCode items has a different keyword to set the value for the current bar

RealCode Item	Return Value Keyword	Return Value Data Type
Indicator	Plot =	Single
Condition	Pass (or Fail)	Boolean
Paint Brush	PlotColor =	Drawing.Color

Lets' say you wanted a basic condition that gives you a buy signal every time price closes above the previous bar's high. To reference pricing data for the current bar you simply call `Price.Close` (see Quickstart examples for more details) . To reference the High value for the previous bar you simply include a parameter value with the number of bars in the past you wish to get the value for: `Price.High(1)`. To write this simple condition it would simply look like this:

```
If Price.Close > Price.High(1) then Pass
```

A Plain English Translation: If the close price of the current bar (today) is greater than the High 1 bar ago (yesterday) then my condition passes for the current bar.

When your code is called (executed), you can access the symbol, volume, bar date, and pricing data (including the open, high, low and close prices). When getting the pricing and volume data you can get the values for the current bar or any previous bars by using an offset index when calling for the pricing or volume data as seen in the example above.

Working with Price and Volume

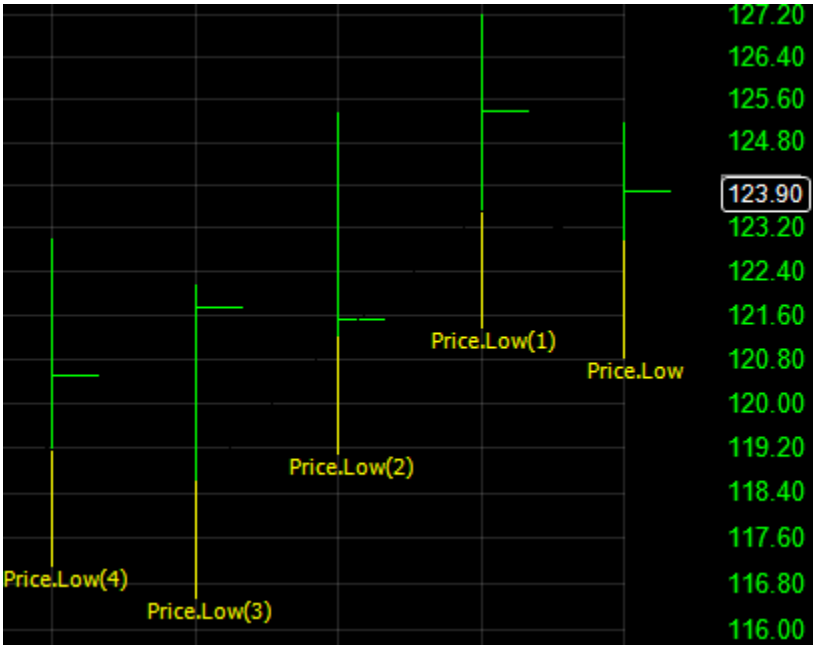
Price and volume for the current symbol and bar interval are always available to you in RealCode.

Volume in StockFinder is reported in 100 share intervals, Therefor when checking for volume in RealCode a value of 1 = 100 shares, a value of 1000 is 100,000 shares.

Price(0)	Returns the close/last value for the current bar
Price.Last	Same as above
Price.Value	Same as above
Price.Open	Returns the open price for the current bar
Price.High	Returns the high value for the current bar
Price.Low	Returns the low value for the current bar
Price.Close	Same as price.last
Price.Open(1), Price.High(1), Price.Low(1),	Returns the open, high and low values 1 bar ago
Volume.Value	Returns the volume (reported in 100 shares) for the current bar
Volume.Value(1)	The Volume (reported in 100 shares) for the previous bar

Looking Backwards

There are a lot of reasons why you may need to look for previous price, volume or indicator values. Some basic built in functions all use look-back (bars ago) parameters: NetChange, PercentChange,New High/Low, Moving Up/Down, Crossovers, Greater/Less than. Using previous values is a core concept you need to understand.



In the picture above, the current bar is the far right bar on the chart. If you were to ask for Price.Low(1) it would be the low for the previous bar. Price.Low(2) would be to bars ago. The chart above should help you visualize what the barsAgo parameters are used for in any of the RealCode calculations.

Advanced Tip:

If there is not enough data to calculate your indicator/condition (because you are calling for data x bars ago) RealCode will not output a value until there is enough data for your code to calculate. You can

help speed this process up and increase performance by doing a check at the top of your code that will exit if the current bar is less than the largest number of bars back you need to evaluate.

What else can you do?

There are also many child indicator functions you can call (moving average, exponential average, stochastics, RSI, etc) that also can use the look-back parameter to get data from previous bars.

Additionally, your RealCode can access the other Conditions and Indicators in the Library or on the Chart. So besides being able to get the values for Price and Volume data, you can get the value of any other indicator or condition. The values are automatically matched up for you when calling your code, so all the same offset calls work for getting any of the values. This means your RealCode can be calculated using the values of the hundreds of pre-defined indicators in the Indicator Library, or any custom RealCode indicators you create.

You can also reference any saved indicator or condition from the Library, including the hundreds of defaults that come with StockFinder, or any indicators or conditions you or another user creates.

You can also declare *UserInput* variables that are inputs into your code and are provided at "run time"(when StockFinder is getting a value for the current bar) to further customize your calculation. An example of a *UserInput* variable might be a period, a color or a numeric value to test against. *UserInput* variables can be set using the QuickEdit feature of the Indicator or Condition. You can access QuickEdit by simply left clicking on the Indicator or right clicking on a condition and choosing edit.

Lastly, when creating a Paint Brush, you can also access the data of the line you're painting. If your paintbrush is applied to something other than price data, your paintbrush can use the data of the indicator you're painting to perform its calculations.

The actual Price History pane does not need to be visible on the chart (nor the volume pane) for the data to be available to you in RealCode. Just know that your calculation will be called for the active symbol and for every bar for the selected bar interval as if the price data was displayed on the chart.

You do not have to produce a value every time you are called. Doing so will create your own custom bar interval. The frequency at which you output the data is entirely up to you (though most calculations will want a value for every bar to match the bar interval of the chart). If you mix your own custom bar interval with another indicator that is on a longer or shorter bar interval, the chart will automatically adjust to add more dates to accommodate the longer bar interval.

Your RealCode calculations will always use the maximum data available (as defined by the number of bars to use in the Data Manager) except when used in a WatchList column or scan. When used as a WatchList column, you only need to produce one value (the most current). When used as a condition light you need only 50 values (the last 50 true/false). When your indicator or condition is used in this manner on the WatchList, StockFinder automatically detects the minimum bars needed to calculate and limits the data to that number for increased performance. Because of the auto detecting feature you should only return values when you intend to and should set `Plot = Single.NaN` (for Indicators) and should not call pass until your condition has enough data to pass.

RealCode Properties and Methods

Besides all the built in .net classes, methods and functions, RealCode has added some additional properties and methods. These are always available in the RealCode editor. There is a class library available directy from within the RealCode editor (you can hit F1 on any keyword to open the help directly for that keyword) or online at http://www.stockfinder.com/realcodeapi/

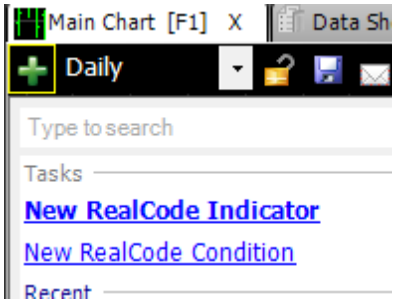
While there are hundreds of classes and functions you can use, we will only cover some of the more common ones here:

Methods:	Data Type	Description
Price.Open	Single	Returns the Open value for the currently calculating bar
Price.High	Single	Returns the High value for the currently calculating bar
Price.Low	Single	Returns the Low value for the currently calculating bar
Price.Close	Single	Returns the closing price for the currently calculating bar
Price.Last	Single	Same as price.close
Price.Open(x), Price.High(x), Price.Low(x), Price.Close(x)	Single	Returns the open/high/low or close for x number of bars ago (x is an integer value)
Volume.Value	Single	The Volume value for the current bar (reported in 100 shares)
Volume.Value(offset)	Single	The volume value for x number of bars ago.
Me.CurrentDate	Date	Returns the date and time for the currently calculating bar
Me.CurrentSymbol	String	Returns the symbol for the currently calculating bar
DateValue(offset)	Date	Returns the date for offset number of bars ago
MyValue(offset)	Single	Returns a value you have already set for the given offset
Me.isFirstBar	Boolean	Returns true if calculating for the first bar of the calculation
Me.isLastBar	Boolean	Returns true if calculating for the last bar of the calculation.
Me.Log.Info(txt)	String Parameter	Call to write to the Info Debug Log
Price.AVG(20)	Single	The 20 bar simple moving average of price
Price.STOC(12,2)	Single	A 12, 2 stochastics of price.
Price.MaxHigh(50)	Single	The highest high in the last 50 bars
Volume.MinValue(40)	Single	The lowest volume in the previous 40 bars.

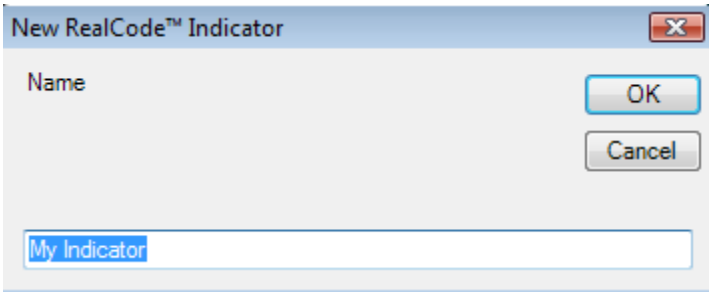
Table 1 –Common RealCode Properties and Methods. This is not a complete list. Please refer to the API Class documentation.

RealCode Indicators

To create a RealCode Indicator, click the Add Indicator button on the chart and select *New RealCode™ Indicator*



Provide a name for your Indicator and Click Ok to open the editor:



RealCode indicators are created in their own pane but you can Overlay or Merge them with a different pane by dragging your indicator to a different pane.

To edit an existing RealCode Indicator Right click on the indicator legend or the indicator itself and choose *Edit RealCode* from the menu.

When creating a RealCode Indicator, you are writing the code to produce a plot (line) on a chart. The result of calling your code will display a value on the chart for every date for the selected bar interval. This means if the chart is on a minute chart, your code will be called for every minute starting with the oldest minute and progressing to the newest minute of price history. If the bar interval is set to monthly, your code will be called for the oldest month and progress a month at a time calling your code for every month of price history available for the active symbol.

To plot a value for the current bar, you simply need to call `Plot =` in your code. Example:

```
Plot = 0

Plot = Price.Close

Plot = (price.high + price.low + price.close) / 3
```

The first example above will simply return a flat line at 0. The second example will simply return the close price for the active symbol for every bar, essentially a price history plot with the line style selected. The third example returns the average trading range for the current symbol (this type of calculation is used in pivot points).

As long as you call `Plot = value` in your code your indicator will plot that value on the chart. If you do not type Plot = in your code, or if the path in your code fails to set Plot to a value or if you set `Plot = Single.NaN` it will not plot a value for the current bar.

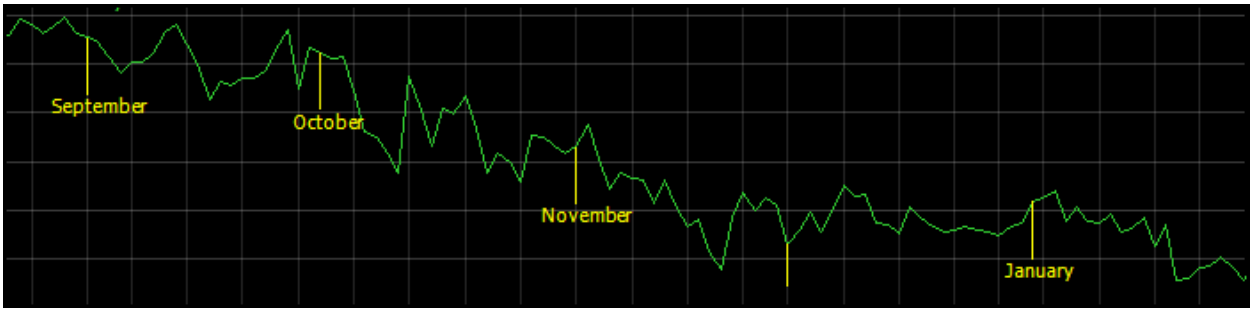
RealCode indicators also get a special property they can set to create a custom Label.

A custom label is any text you wish to appear on the chart or with a data export of your RealCode indicator. Custom labels also use the current bar methodology like the `Plot = value` syntax.

The following is an example of a custom indicator that draws the month name as a label on the first bar of the month:

```
plot = price.close

If price.DateValue.month <> price.datevalue(1).month  Then
   label = price.DateValue.tostring("MMMM")
End If
```

The code checks for the month of the current date with the month of the previous bar, and if they're not the same (meaning it's a new month) it will set the Label property to the month name (`.tostring("MMMM")` formats the current date value with the month name). The code above produces a chart in the image below:



The display of the custom labels can be set using the Indicator editor. The size, position and color can all be changed from the Indicator editor. See the StockFinder manual for more information on chaning the display of your RealCode custom labels.

RealCode Conditions

To create a new RealCode condition: click on the *Add Indicator/Condtion* button at the top of the Chart and select *New RealCode Condition*. To edit an existing RealCode condition, right-click on the condition "bubble" and choose *Edit RealCode*. See Figure 1 for an example of a Condition Bubble.

Figure 1 - RealCode rule bubble in red at bottom of pane

Like an indicator, your RealCode condition runs in the context of the current charts symbol and Bar interval. By default, RealCode conditions are in the fail (false) state so you only need to program the RealCode condition when it passes. Conditions can be used to paint indicators, scan a WatchList, Sort on a Condtion light in the Watchlist or perform a trade condition in BackScanner.

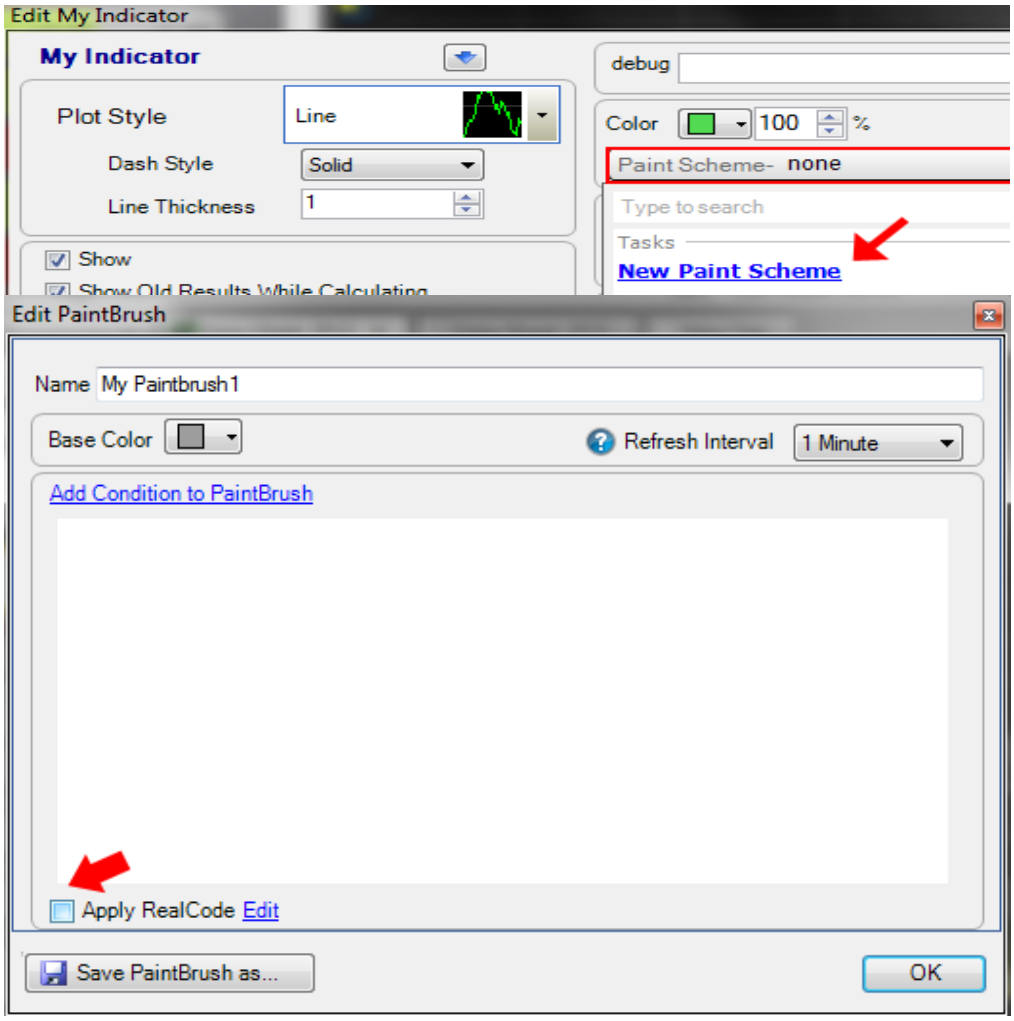
When your condition becomes true, you simply need to call the `Pass` method. Example:

```
If price.close > price.close(1) then Pass

If Volume > Volume(1) then Pass

If Price.High = Price.Last then Pass
```

The first example will pass if the current close is greater than the previous close. It will put a true marker on the current bar. The second example passes if the current volume is greater than the previous bar's volume. The last example will put a true marker on the chart if the price closed at the high of the day. If your code has already made a call to `Pass`, but you need to change the result simply use the `Fail` keyword at any point in your code.

RealCode Paint Brush

You can paint your indicators by your RealCode Conditions, but sometimes you might want some special logic to perform your painting. RealCode paintbrushes are part of the Paint Scheme that is applied to your indicators. To create a RealCode Paint Brush, Left click on the indicator you wish to paint and select New Paint Scheme on the Paint Scheme drop down . Check the *Apply RealCode* checkbox, this will open the RealCode Paintbrush Editor.



When creating a RealCode Paint Brush you are writing code to color an indicator on the chart. Paint Brushes are applied to an existing indicator on a Chart. A Paint Brush is similar to a RealCode Condition in that they both test for a Boolean value. Unlike a Condition, a Paint Brush can return different values (colors) for multiple Boolean tests. The result of calling your Paint Brush code will assign a color the currently calculating bar. To set a color in code, set the `PlotColor` variable equal to the color you wish to apply. Example:

```
If price.close > price.close(1) then
   PlotColor = Color.Lime
ElseIf price.close < price.close(1) then
   PlotColor = Color.Red
Else
   PlotColor = Color.White
End if
```

In the example above, we assign one of three colors to the plot. The first line checks for a positive gain (current price greater than the previous bar's price). If it's an up bar, it sets the `PlotColor` to `Color.Lime` (Lime is the bright green on the price chart). The third line checks for the opposite of the first line, that the close is lower than the previous bar's close. If this is true it sets the `PlotColor`

to `Color.Red`. The fifth line is simply the last possible combination: it closed the same as the previous bar (it was neither down nor up). In this case it will set `PlotColor` to `Color.White`.

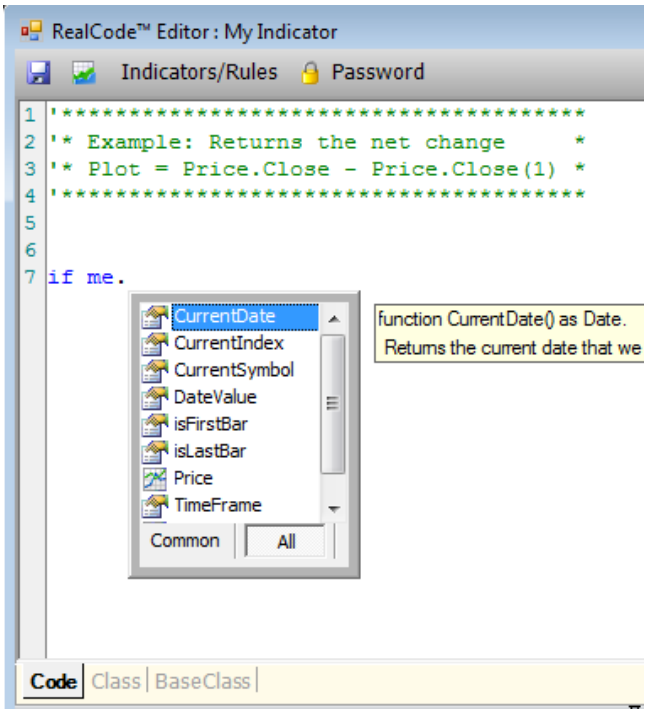
Colors can also be assigned from UserInput variables. This allows you to change the output color of your Paint Brush without having to edit your code.

Saving your RealCode

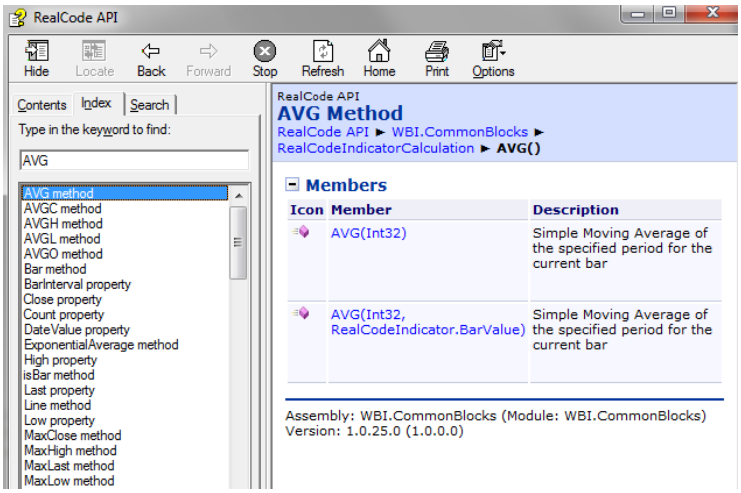
You can save RealCode Indicators and Conditions by clicking the Save button on the editor or right clicking on the item and choosing save. To save a Paint Brush, open the Paintbrush editor and choose the save button. If your RealCode references any indicators or conditions on the chart, then those indicators/conditions will be embedded into your code so it will continue to work when you open your RealCode at a later time.

Chapter 4 - The RealCode Editor

Whenever you create or edit RealCode you will use the RealCode editor. It is a text editing tool that behaves much like Microsoft Visual Studio. When typing in the editor, you will receive code tips (similar to MS Intellisense™) to help you complete your code. For instance when typing in the editor, you can type `Me.` (don't forget the period) to see all the properties and methods you can call on the Me object. To see everything available to you for pricing data, type `Price.` (don't forget the period) and the list of properties and methods from the `Price` object will pop up (Figure 2). You can select an item from the list by continuing to type the name of the item, or you can use the up/down arrow keys to select the item on the list and hit tab to finish the word. The code tips are a great tool for learning all the properties and methods that are available for you to call as well as speeding up your development time by typing code for you. The code tips are separated into the most commonly used but you can display all properties/methods by clicking on the All tab. You can also invoke the Code tips popup by hitting ctl+spacebar.



You can access the RealCode API documentation from the Editor by hitting F1 at any time. For instance, if you need help with the `AVG` function you can type `Price.AVG` and hit F1 to pull up the help for the `AVG` function. This will list the functions on the left hand side. Double clicking on a topic will pull up the documentation for that topic and allow you to browse the help and code samples.



By default, the editor will re-compile your code every time you change lines of code or select a new line with your mouse cursor; essentially every time you hit the *enter*, *up* and *down* arrow keys or click on a line with your mouse. The compile process takes place in the background and any errors it encounters will display in the error list at the bottom. Sometimes, the editor will highlight a problem area of code with a red squiggly line under a word at or near the problem. Figure 3 shows the RealCode editor with an error on line 3. Notice the red squiggly line on the word `Then`. The error in the error list says we're missing a comma, ')' or a valid expression. We forgot to close our parentheses for the `netchange`(1). Putting our mouse over the highlighted word `Then` shows us a tool tip with what the problem is. You can double click on the error in the list to take you to take you right to the line with the problem.

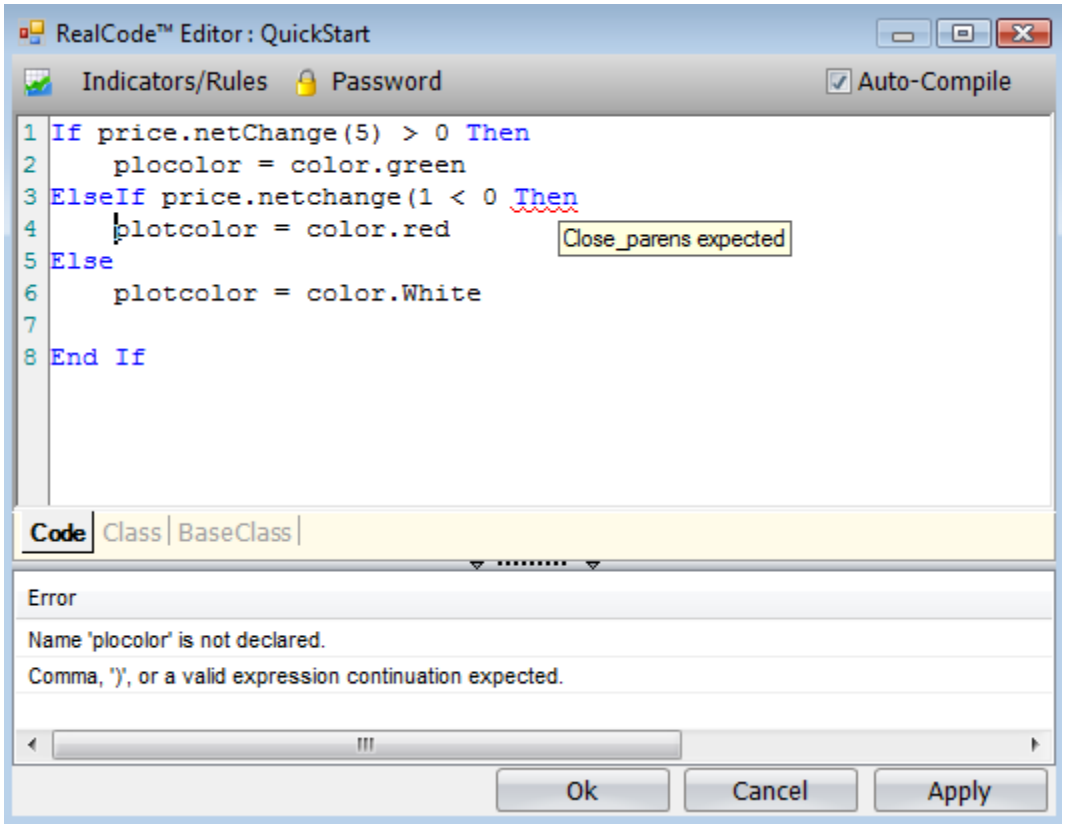


Figure 3 - RealCode Editor with an error

If you need more room for editing your code, you can collapse the bottom error list by clicking on the splitter between the text input and the error list. Also, you can resize the editor window or double click the header to maximize the code window.

Clicking the *Apply* button will compile your code and update the RealCode item with your changes. Hitting *Apply* allows you to see the results of your edits to verify the results. When you are finished editing the code, you can close the editor window with the *Ok* button or hitting the X in the upper right corner.

Chapter 5 – RealCode Functions

RealCode includes some built in functions (calculations). These are not standard vb.net functions, they have been provided by the StockFinder compiler as commonly calculated methods. All of these functions work on the built in Price and Volume objects as well as any indicators you import into your code. This chapter in no way is a complete overview of all the functions available to you. Please refer to the RealCode API from the RealCode Editor in StockFinder (press F1) or online at http://www.stockfinder.com/realcodeapi/

Function	Return Type	Description
NetChange	Single	Returns the net change from the previous bar
NetChange(NetChangePeriod,BarsToOffset)	Single	Returns the net change for the specified period and offsets x bars ago with the BarsToOffset. NetChange(1,0) is equivalent to NetChange()
PercentChange	Single	Returns the percent change from the previous bar
PercentChange(PercentChangePeriod,BarsToOffset)	Single	Returns the percent change for the specified period and offsets x bars ago with the BarsToOffset. PercentChange(1,0) is equivalent to PercentChange()
MaxOpen(period),MaxHigh(period), MaxLow(period), MaxClose(period)	Single	Returns the max value (open,high,low or close) in the last x number of bars (period).
MinOpen(period), MinHigh(period), MinLow(period), MinClose(period)	Single	Returns the min value (open, high low or close) in the last x number of bars (period)
TradeRange	Single	Returns the High minus the low for the current bar
TradeRange(offset)	Single	Returns the High minus the low for the offset number of bars ago
Me.BarInterval	TimeSpan	Returns the currently selected BarInterval as a TimeSpan object.
Indicator.AVG(period)	Single	Returns a simple moving average value for the specified period, for the currently calculating bar. Can be applied to Price,Volume or any imported indicator
*Indicator.*MovingAverage(period)	Indicator	Returns the indicator for the moving average for the specified period. Can be used to chain multiple calculations.
*Indicator.*XAVG(Period)	Single	Returns the exponential moving average value for the specified period for the currently

		calculating bar. Can be applied to Price, Volume or any imported indicator
*Indicator.*STOC(period,%k)	Single	Returns the stochastics value for the specified period for the currently calculating bar. Can be applied to Price, Volume or any imported indicator
Indicator.RSI	Single	Returns the RSI value for the specified period of the currently calculating bar.
Me.isDaily, Me.isMinute, Me.isHourly, Me.isMonthly, Me.isWeekly, Me.isYearly	Boolean	Returns true if your code is set to the specified bar interval

NetChange & PercentChange Functions

`NetChange` - Returns the net change from the close of the previous bar to the close of the current bar. You can also provide it two parameters to change the bars and period for the net change. The following two lines of code return the net change from the previous bar to the current bar (net change today if on a daily chart):

```
Plot = Price.Close – Price.Close(1)

Plot = Price.NetChange()
```

You can also call `NetChange()` with two parameters. The first parameter changes the period for the net change. The two lines below calculate the net change from today to two bars ago (two day net change if on a daily chart).

```
Plot = Price.Close – Price.Close(2)

Plot = Price.NetChange(2,0)
```

The second parameter for net change offsets both bars. Instead of calculating for the current bar, you could find the next change for the previous bar. The following two lines of code both provide the previous bar net change (yesterday's net change on a daily chart)

```
Plot = Price.Close(1) – Price.Close(1 + 1)

Plot = Price.NetChange(1,1)
```

`PercentChange` returns the percent difference from the previous bar to the current bar. It also accepts the same parameters as `NetChange`.

Max/Min Open,High,Low and Close

`MaxHigh (MaxOpen,MaxLow,MaxClose)` – all return the highest value for the given bar over a period. The following two code examples both return 10 bar high (10 day high if on a daily chart):

Calculate the Max High for price over the last 10 bars, using a loop

```
Dim calculatedMax as single = single.minValue
For i as integer = 0 to 9
   calculatedMax = System.Math.Max(calculatedMax,Price.High(i))
Next i
Plot = calculatedMax
```

Calculate the Max high for price over the last 10 bars using the MaxHigh function

```
Plot = Price.MaxHigh(10)
```

MinHigh (MinOpen,MinLow,MinClose) – all return the lowest value for the given bar over a period. The following two code examples both return the 10 bar low (10 day low if on a daily chart)

Calculate the Min Low for price over the last 10 bars, using a loop

```
Dim calculatedMin as single = single.MaxValue
For i as integer = 0 to 9
   calculatedMin = System.Math.Min(calculatedMin,Price.Low(i))
Next i
Plot = calculatedMin
```

Calculate the Min Low for price over the last 10 bars using the MinLow function

```
Plot = Price.MinLow(10)
```

All of the Max and Min functions also accept a second optional parameter to get the Max/Min value with an offset. To get the 10 bar high for yesterday's bar you would call the example below:

```
Plot = Price.MaxHigh(10,1)
```

TradeRange Function

TradeRange returns the High minus the Low for the current bar. The following two lines of code are equivalent:

```
Plot = Price.High – Price.Low

Plot = Price.TradeRange
```

`TradeRange` can also take a parameter to offset the bars for the calculation. The following two lines of code return the previous bar trading range (yesterday trading range if on a daily chart)

```
Plot = Price.High(1) – Price.Low(1)

Plot= Price.TradeRange(1)
```

Bar Interval Functions

Version 5 includes some new helper functions to help identify the bar interval for your calculation.

Bar Interval functions

isMinute	Returns true if chart is set to 1 minute
isHourly	Returns true if chart is set to 1 day
isDaily	Returns true if set to 1day
isWeekly	Returns true if set to 7 days
isMonthly	Returns true if set to monthly
isQuarterly	Returns true if set to quarterly
isYearly	Returns true if set to Yearly
BarIntervalisDays(numOfDays)	Returns true if the bar interval is set to the specified number of days
BarIntervalIsMinutes(numOfMinutes)	Returns true if the bar interval is set to the specified number of minutes
BarIntervalIsYears(numOfYears)	Returns true if the bar interval is set to the specified number of Years

Code Samples:

```
If isMinute then
   ' Perform a minute calculation
ElseIf isDaily then
   ' Perform a daily calculation
Elseif isMonthly then
   ' Perform a monthly calculation
End if
```

BarInterval Property

If you need to know more than just a true/false about the current BarInterval the BarInterval function will return a .net TimeSpan structure that represents the currently selected BarInterval . The following code is an example of how to use the TimeSpan object to determine the number of minutes or days for the current BarInterval:

```
If me.BarInterval.Days > 0 then
   ' Daily or higher
   Select Case me.BarInterval.TotalDays
        Case  1
             ' daily chart
        Case 2
```

```
                ' 2 day chart
    End Select
Else
    ' Intraday
    Select Case me.BarInterval.TotalMinutes
         Case  1
                ' 1 Minute chart
         Case 2
                ' 2 Minute chart
    End Select
End if
```

For more information on the TimeSpan structure see the MSDN documentation at **http://msdn.microsoft.com/en-us/library/system.timespan.aspx**

Child Functions

There are a series of child functions available to call on any RealCode indicator. They are named Child functions because they calculate on an existing set of data (Price, Volume). They data they use in their calculation is the Parent data. These are more commonly known as indicators like Moving Average, Stochastics, Relative Strength Index and Wilders RSI. Any indicator you import via the Library or Chart can call these child functions to get the derivative data.

Indicator.AVG & *Indicator*.MovingAverage(20)

A standard Simple moving average for the specified period

```
Plot = Price.AVG(30) ' 30 Bar Simple Moving Average
Plot = Volume.AVG(20) ' 20 bar Simple Moving Average of Volume
```

The `AVG` function will return a *single* data point. `AVG` is the shorter form of calling `MovingAverage`. You can use the longer version of you need to "chain" multiple child indicators together. For instance you want a moving average of a moving average:

```
' Plot the 5 bar moving average of a 40 bar moving average
Plot = Price.MovingAverage(40).AVG(5)
```

You can also use the look-back parameter to get previous bars of data. The code below will return the previous (yesterday on a daily chart) bars 20 period moving average:

```
Plot = Price.AVG(20,1)
```

Indicator.XAVG & Indicator.ExponentialAverage

`XAVG` and `ExponentalAverage` use the exponential moving average calculation. These work in the same way as the `AVG` and `MovingAverage` functions:

```
Plot = Price.XAVG(30) ' 30 Bar Exponential Moving Average
Plot = Volume.XAVG(20) ' 20 bar Exp Moving Average of Volume
```

Indicator.STOC & Indicator.Stochastics

`STOC` and `Stochastics` use the exponential moving average calculation. These work in the same way as the `AVG` and `MovingAverage` functions:

```
Plot = Price.STOC(30,3) ' a 30,3 Stochastics
Plot = Price.Stochastics(30,3).AVG(5) ' A 5 bar moving average of a 30,3 Stochastics
```

Indicator.RSI & Indicator.RelativeStrengthIndex

A relative strength index for the specified period and average period

```
Plot = Price.RSI (15,5) ' a 15,5 RSI
```

More Functions, Properties and Methods

For more documentation on the classes, functions properties and methods available to you in RealCode, see the RealCode API class reference: http://www.stockfinder.com/realcodeapi/

Chapter 6 - Accessing External Indicators, Conditions and Parameters.

A RealCode calculation can read the values of the existing *Indicators* and/or *Conditions* (including other RealCode Items). These existing Indicators and/or Conditions can be from the Library, your computer or from a Chart in your layout.

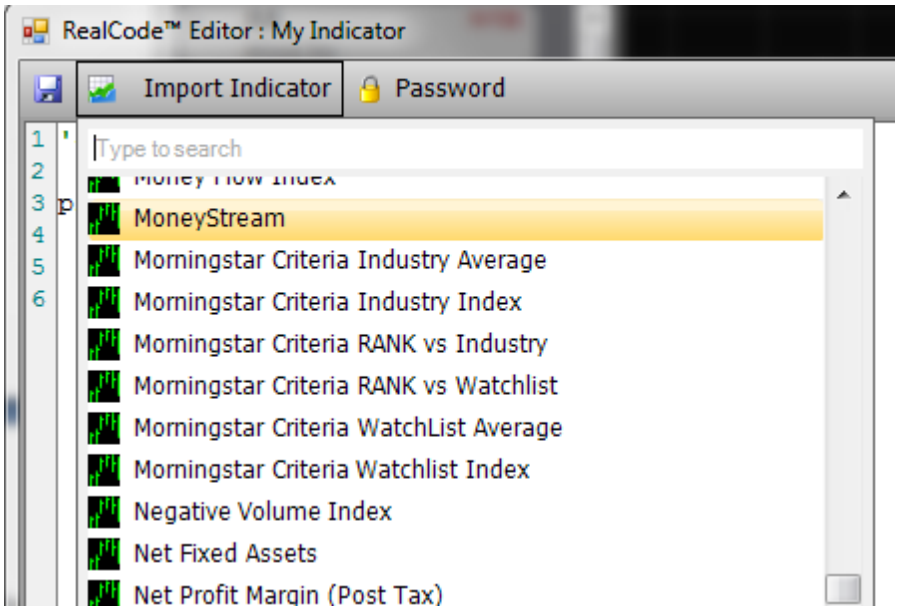
You can also define *User Input Variables* that can be assigned at run time from the Edit option. This is a really great feature of RealCode. This means you can daisy chain calculations to create more complex and aggregate data and use variable parameters for performing your calculations.

External data is referenced through a *Directive*. A RealCode directives start with the comment symbol (') and then the pound symbol (#). By adding a directive you instruct the RealCode editor to write some background plumbing code to ensure that you can read the values of an external item. These directives are a StockFinder RealCode™ exclusive feature and not a part of the .Net framework.

Importing Indicators

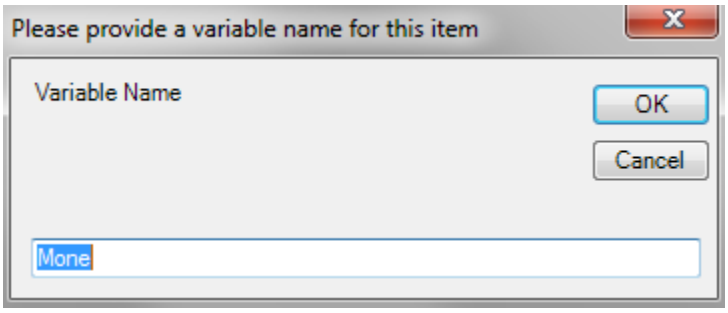
Referencing an Indicator in your RealCode allows you to leverage the existing Indicator Library (or your own custom RealCode Indicators) and perform aggregate calculations or conditional testing. Any indicator values in the Library or on your Chart can be accessed by your RealCode.

The easiest way to access an Indicator is to add it from the Import Indicator button and choosing the Indicator or Condition to import into your code:



After you select the indicator or condition to import into your code, you will be asked to provide a variable name for the indicator. You will not be able to change the variable name once chosen.

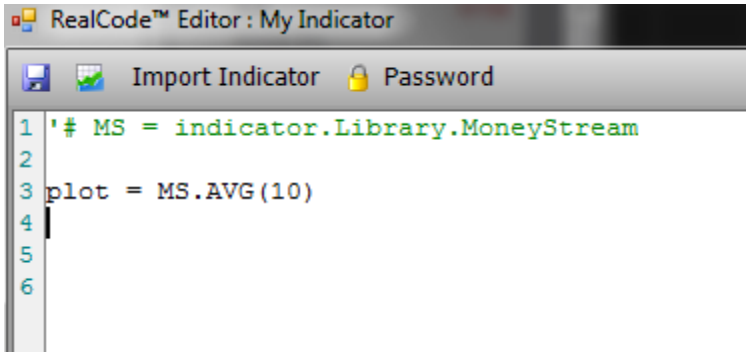
Type the variable name when prompted and choose *Ok.*



In previous images, if we were to choose MoneyStream from the Library and gave it a variable name of MS it would add the following line of code:

```
'#MS = indicator.Library.MoneyStream
```

This line of code starts with our input directive `'#` and then declares a variable name for the indicator. In the above example MS is the variable name we gave to MoneyStream Once we have referenced the item, we can use it just like the built in Price and Volume variables and can call any of the built in functions (Max/Min, netchange etc).



The MoneyStream Indicator can now be accessed using `MS.Value` to get the value at the current bar, or `MS.Value(x)` to get the value x number of bars ago. The Value property returns a Single data type. You can access any of the Open, High, Low, and Last properties for an indicator (if applicable) by using the variable name. You can also call child functions just like the Price and Volume objects. In the screen shot above, we are plotting the 10 bar moving average of MoneyStream.

Importing Indicators/Conditions from the Chart

In addition to being able to import indicators and conditions from the Library, you can also reference the items on your chart, creating a link to those values. You might import indicators or conditions from the chart, instead of the library, because you want the values to stay synchronized when you change a parameter on the chart. Say you import a moving average to your RealCode from the chart. If you update the moving average period (or change the average type from simple to exponential) your RealCode will automatically recalculate on the new moving average values.

Importing a condition or indicator from the chart by dragging the indicator or condition directly into the RealCode Editor and dropping it on the code. This would create a new line of code similar to below:

```
'# MA = chart.MovingAverage.3
```

The MA is the variable name assigned to your indicator and can be renamed to any name you like (unline indicators/condtions imported from the library). This variable will now behave just like any other Indicator object and can be called for it's .Value, .MaxHigh, AVG etc.

When saving a RealCode item that references indicators and/or conditions on the chart, those references will become embedded into the RealCode and behave as if you added them from the library. When you load a RealCode item that referenced a chart indicator, it will no longer be linked to the original item on the chart. Any parameters for the original indicator can be set by editing the RealCode indicator properties.

Chapter 7 - RealCode Classes

So far, all the examples have been using the default Code tab in the editor. The code tab handles looping and calling your code for each bar on the price chart. Sometimes your calculation might be more complex or might need to control the enumeration of the prices. This is where RealCode classes fit in.

A class is an object oriented concept and is a standard programming construct. For more information about classes, view the .net Framework reference at http://msdn.microsoft.com . For our purpose writing your code as a class gives us the following advantages:

- User functions. You can write functions in your class to perform the same calculation more than once
- Inheritance – your class can inherit another class that contains existing reusable logic
- Custom Looping – You can override the default RealCode behavior and enumerate the price data using your own custom loop.
- Custom Timeframing – while this was possible without classes, it's much easier with the custom looping.
- "Global" variables without the need for static keyword – you can define class level member variables that do not need to be managed like the static variable examples in later chapters.
- Private Classes – you can write your own classes to encapsulate a custom data set in your calculation.

Fundamentally, there is nothing different about writing RealCode as a class or as a function body. When writing RealCode, the compiler generates a class for you behind the scenes. You can switch back and forth between the class and the function body by using the tabs at the bottom of the RealCode Editor:

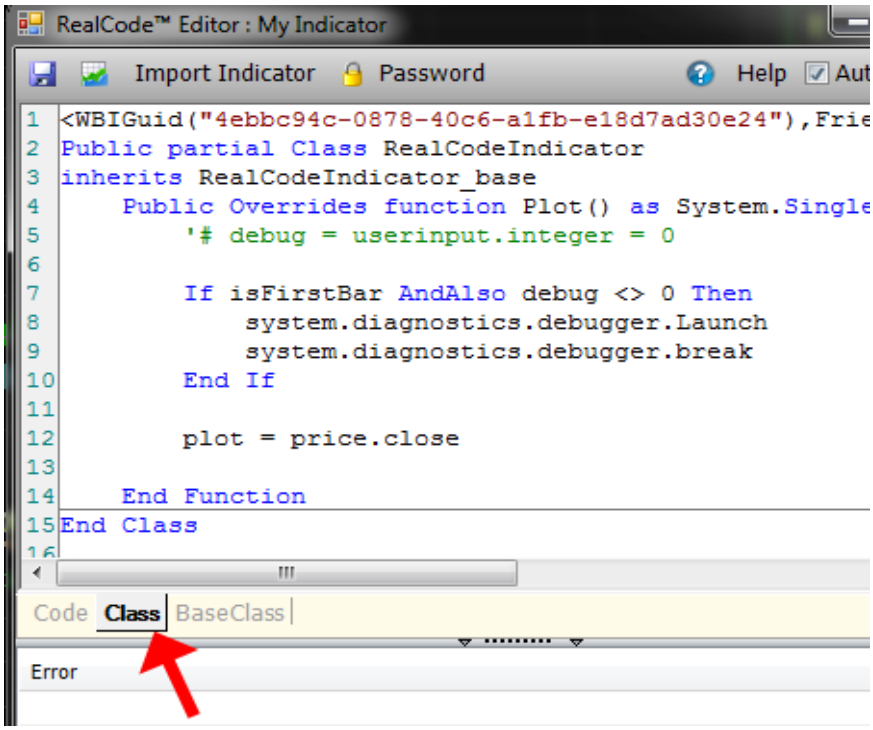


Figure 4- The RealCode class tab

When writing a RealCode class you simply need to fill in the default method that is stubbed out when you open the Class tab. For an indicator that method is named Plot. By default, RealCode classes behave the same as standard RealCode: your method will be called once for every index in the Price history. You can override this default behavior and do all your own looping by creating a new constructor at the top of your class.

Manual Looping (AutoLoop = false)

Manual looping is the process of changing the default RealCode behavior of calling you once for every date on the price chart to calling you once for the active symbol. This can improve performance because you can decide how much (if any) to loop. The drawback of this method is that your inputs will not be interpolated (synchronized) and you will not be able to use any of the built in child functions, lookback functions and most of the built in helper functions.

Additionally, when autoloop = false, you can no longer use the return keywords for your item (plot, plotcolor or pass). Instead you will need to construct your own output by calling AddToOutput (see examples below)

When performing your own looping you will want to call the `Price.Bar` object to get the raw array data. Unlike the default `Price.Open` object where an index parameter is the number of bars ago, `Price.Bar.OpenValue` is a 0 based index, where 0 is the first Value in the array and `Price.Count - 1` is the last bar in the array. In pseudo code:

```
Price.Open(0)  <> Price.Bar.OpenValue(0)
```

Price.Bar.OpenValue(0) is the first open value in the data array while Price.Open(0) is a moving target, it will always equal the current open value when used in the traditional AutoLoop method.

When performing your own price looping, you can call the `AddToOutput` method to create your indicator. `AddToOutput` has two overrides to create your indicator, one to output a single value and one to output bar data.

```
Mybase.AddToOutput(dateValue,Open,High,Low,Close)

Mybase.AddtoOutput(dateValue,Value)
```

Although you are writing your own class, you cannot write any imports statements at the top of the class, nor should you modify the first or second line of the class. Doing so will result in your class not compiling correctly. When editing a RealCode class you can edit the lines between the inherits RealCodeIndicator_base and End Class. See the sample class below that performs our own looping to output the price data:

```
   Public partial Class RealCodeIndicator
   inherits RealCodeIndicator_base
      Sub New
           MyBase.AutoLoop = False
      End Sub
      Public Overrides function Plot() as System.Single
           For i As Integer = 0 To price.bar.count - 1
                Me.addtooutput(price.bar.datevalue(i), _
price.bar.value(i))
           Next

      End Function
   End Class
```

***Note**: When AutoLoop is set to false, you cannot use many of the built in methods and properties. Any property or method that uses lookback or barsAgo parameters will not be valid in the manual looping context. If you are looping over the Price.Bar data (or the bar data for any imported indicator, or any other RealCodeIndicator object) the lookback parameters, AVG functions, isFirst,isLast are not valid because the class is not managing the looping for you. In fact many of the functions are simply not needed as you are controlling the index of data array. Additionally, the need for static variables (as used in many examples in this reference) are not needed as your function will only be called once.*

Chapter 8 – Getting Deeper into RealCode

Writing with RealCode is pretty straightforward. Your code will be called (executed) for every bar that is to be drawn, painted or tested (Indicator, Paint Brush or Condition). If your code requires a bit more complexity than a simple mathematical calculation this chapter will help you solve more advanced RealCode problems.

Cumulative Indicators

When writing a cumulative indicator (like and Advance/Decline or Exponential moving average) you will need to add the *'#CUMULATIVE* directive at the top of your code. Cumulative indicators are indicators that change based on their starting value, or need the entire data set to perform the calculation. Using the '#CUMULATIVE directive will instruct the compiler to always use as much history as available. When your code is used in a WatchList, the software attempts to use the least amount of data as possible to perform your calculation. This performance enhancement cannot be used on cumulative indicators.

Variables and Scope

Scope is a programming term used to define what parts of your code can access different variables. By default, every variable "falls out of scope" when the end of your code is reached. This means that the values of your variables are "reset" to their default value and do not persist between calculations. You can override the default scope of a variable by adding the `static` keyword. Static tells the compiler to keep the value in memory between calls to your code. Example:

```
Static AdvanceCount as integer

If Me.isFirstBar then
   AdvanceCount = 0
Else
   If price.Close > price.Close(1) then
         AdvanceCount = AdvanceCount + 1
   Else
         AdvanceCount = 0
   End if
End if

Plot = AdvanceCount
```

The above code outputs a line that counts the number of bars in a row that the stock went up. If the stock does not go up then it resets the counter back to 0. Because by default our variables fall out of scope at the end of our code, we need to declare our counter variable `AdvanceCount` with the `Static` Keyword. This will preserve the current value of the variable so you can access it at the next time your code is called.

When using Static variables, if you need to reset your value back to a default, test the Me.IsFirstBar (or Me.IsLastBar to clean up any static variables and reduce the amount of memory used). The second and third line of code test for the first bar of the calculation and reset our counter back to 0.

***NOTE**: When using Classes you do not need to use static variables to keep values between calls to your code as you can create a class member variable. The previous example can also be represented in a class as the following:*

```
Public partial Class RealCodeIndicator
inherits RealCodeIndicator_base

      Private AdvanceCount As Integer =0

      Public Overrides Function Plot() As System.Single
            If isFirstbar then
                  AdvanceCount = 0
            End if

            If price.NetChange > 0 Then
                  AdvanceCount += 1
            Else
                  AdvanceCount = 0
            End If

            Plot = AdvanceCount
      End Function
End Class
```

Where in the world am I running?

Sometimes you might need to know information about the chart or the list in which you're calculating for. When your code is visible on a chart you might want to know the number of bars that are visible or the chart start/end dates. When running as part of a Market Indicator you might want to know how many symbols are in the list and when you're on the first or last symbol. Version 5 adds some additional RealCode functions to access this data:

`Me.ActiveChart`	Returns an object representing the active chart
`Me.isListCalc`	Returns true if running as a market indicator
`Me.ActiveList`	Returns an object representing the market indicator calculation

The `ActiveChart` object will give you access to the following properties:

`BarsVisible`	The number of bars visible on the chart, set by the Zoom factor
`ZoomEndDate`	The far right date on the chart, or the latest date visible
`ZoomStartDate`	The far left date on the chart or the first date visible

By default, your RealCode will only be called when a symbol, bar interval or parameter is changed. There is a new directive you can add to your code to have it re-calcluate when the zoom/scroll changes on the chart:

```
'#RECALC_ON_ZOOM_SCROLL
```

Adding this directive will cause your RealCode to recalculate on every zoom or scroll change. NOTE: this may significantly reduce performance especially if your RealCode uses a lot of resources or cpu.

Debugging

With the introduction of the Custom Label property in RealCode Indicators (see chapter 3) debugging your RealCode has become easier than in previous versions of StockFinder.

Using the Label property in combination with the Export Data feature of StockFinder you can attach any text value to every value you plot with your indicator. This is by far the easiest way to include some debugging information with your RealCode.

For example, lets say you're having trouble with some logic in your RealCode. You have a variable counter that you want to see the value for on every iteration of your code. You could output the counter's value in the Label property and then show that label on the chart or export your indicator and view the counter values. The following code illustrates an example where you can show your internal variables in the Label property.

```
Static myCounter As Integer
If isFirstbar Then
   myCounter = 0
End If

If price.netchange > 0 Then
   myCounter += 1
Else
   mycounter = 0
End If
plot = price.close * myCounter
Label = MyCounter
```

The second option for debugging is to write to the debug log. The debug log can be written to by using the `Me.Log` property. You can view the debug log from the *Help* tab. The Log property has a few sub properties to choose which log style to write to. The log style is simply a matter of what type of message you want to write. You can write to the Error, Warning or Info logs. Examples:

- `Me.Log.Info("This is my log text")`
- `Me.Log.Warn("This is my warning test")`
- `Me.Log.Err("This is my error message")`

The debug log is a good place to put single lines of text once per call (say on the first or last bar of your RealCode). If you want to write to the debug log on every call to your code, you should consider using the Label property instead.

Additionaly, if you have Microsoft Visual Studio installed, you can use it's built in just-in-time debugger . This will allow you to set breakpoints and step through your code line by line. To use Visual Studio's debugger you simply need to add the `Break` statement anywhere in your code. If you add it to the top and test for the first bar, it will break into your code the first

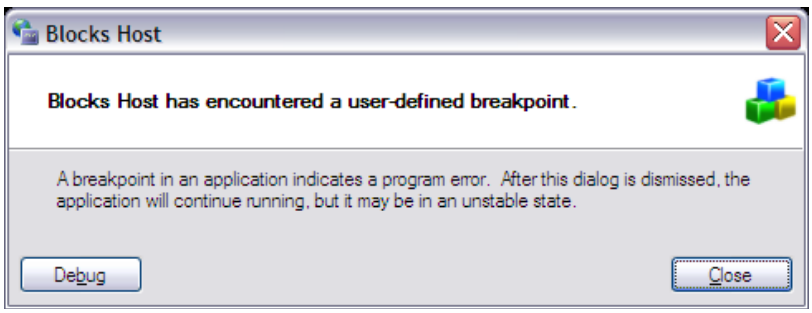
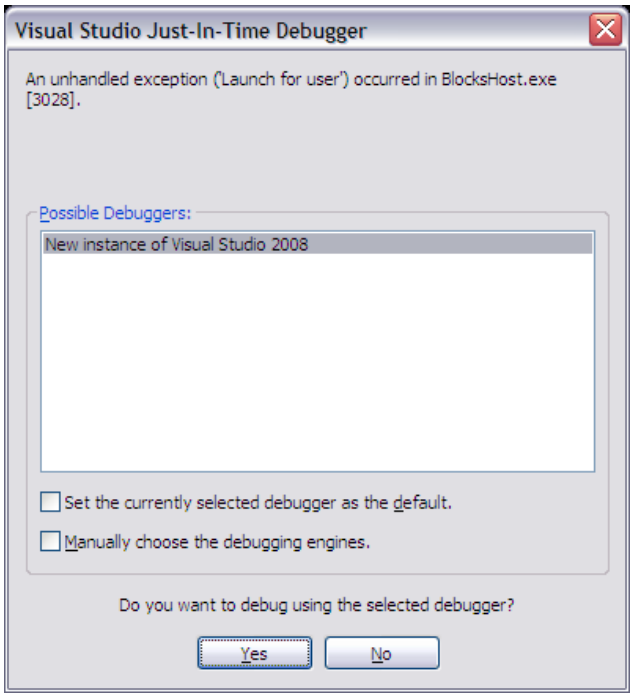


Figure 5 User-Defined Breakpoint

time it's called. Example:

```
If Me.isFirstBar Then
   Break
End If
```



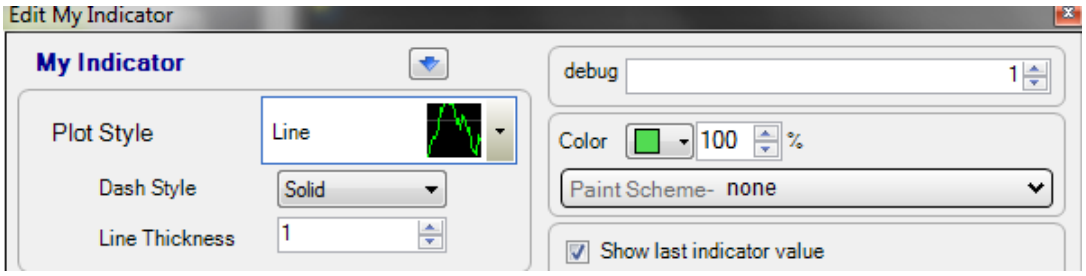
When the debugger hits that line of code it will prompt you to debug . Click the debug button. You will then be prompted for the Just-in-time debugger to use. Choose *New Instance of Visual Studi o*

If you get the message "No source code for the current location" you may need to hit F8 or F11 (depending on your visual studio configuration) to step into the next line of code. Continue to hit F8 or F11 until your code appears in the debug window. You should now be attached to the Visual Studio Debugger and can use all the built in features for debugging your code.

If you want to be able to turn your debugging on or off without recompiling the code, you could use the following code snippet:

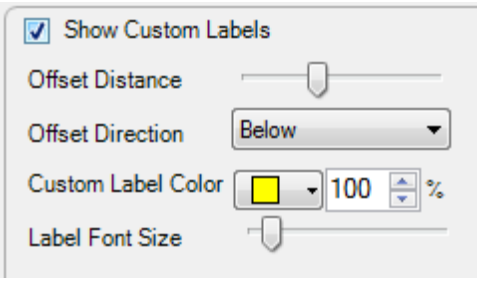
```
'# debug = userinput.integer = 0
If Me.isFirstBar andalso debug <> 0 Then
   Break
End If
```

The code sample above adds a user input variable that allows you to change the value of the debug variable. When you want to launch a debugging session, change the debug value to any other value. When you are finished debugging, change it back to 0 to keep it from stopping on your breakpoint.

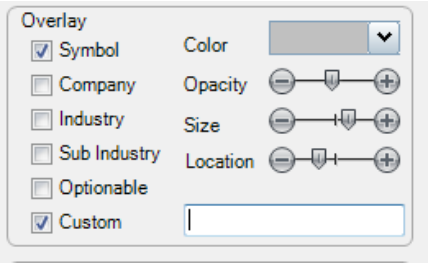


Change the debug userinput variable to 1 to lauch the debugger in your RealCode. Change it back to 0 when you no longer want to debug your code.

Labels and Chart Overlay Text



As mentioned in the debugging section, you can add labels to a RealCode indicator to draw text on specific bars of the chart. To apply a label for the current bar, simply set the Label = property to the value you want to appear. To get the labels to appear, you need to ensure that the Show Custom Labels option is checked on your indicator editor (left click on your RealCode indicator on the chart to show the editor. See chapter 3 for examples and screen shots of using the Label property.



In addition to the Label propery, you can set a chart overlay text value from code that will appear on the overaly of the Chart (custom overlay needs to be checked from the Chart Properties: righ click on the chart and choose chart properties, check Custom Overlay). This allows you to add any text or numeric value to your chart.

```
8  If isLastBar Then
9      Me.ChartOverlayText = "Last:" & price.close
10 End If
```

Add the word Last: and the current close value to the chart overlay

Reading Price data for a Different Symbol and/or Bar Interval than the active chart

By default, your RealCode is calculating under the context of an active symbol. You can read the active symbol by calling the CurrentSymbol property. All of the calls to the Price object will return your values for the current symbol. You may wish to include or compare pricing data from different symbols,

or even different bar intervals than the one you're calculating. Version 5 of StockFinder adds a new method to return pricing data for different symbols and bar intervals.

Example of how to check the current symbol

```
If me.CurrentSymbol = "MSFT" then
   ' We're calculating for Microsoft
End if
```

Below is a basic example of how to plot the daily price data for QQQQ no matter what the Active symbol is for the chart:

```
Static altPrice As WBI.CommonBlocks.RealCodeIndicator
If Me.isFirstBar Then
      altPrice = Me.PriceData("QQQQ")
End If
plot = altPrice.value
```

The key method in the above example is the `Me.PriceData` method. This will return a `RealCodeIndicator` object that has all the pricing data for the specified symbol. This `RealCodeIndicator` object is the same as the built in `Price` object, allowing you to call all the built in methods including all the child calculations, lookback parameters and properties and even the bar object for use during manual looping. Anything you can do with the Price object, you can do with the returned RealCodeIndicator object. Additionally, this object is interpolated (synced) with the current index for the RealCode calculation, meaning that calling .high, .close or any other property will return the value for the same bar as as the Price object.

In addition to getting daily data, you can specify a bar interval to use for your pricing data. Below is a sample that returns 5 minute data for QQQQ:

```
8  Static altPrice As WBI.CommonBlocks.RealCodeIndicator
9  If Me.isFirstBar Then
10     Dim tf As New MinuteTimeFrameProvider
11     tf.nummins = 5
12     altPrice = Me.PriceData("QQQQ", tf)
13
14 End If
15 plot = altPrice.value
```

In the screenshot above, we are plotting the 5 minute QQQQ data. Line 8 in the screenshot defines our static variable altPrice that will hold on to the QQQQ data object. Line 9 test for the first bar in our calculation. On line 10 we define a MinuteTimeFrameProvider object. This object represents all of the minute intervals. On line 11 we set the NumMinutes property to 5, meaning we want each bar interval to be 5 minutes. We can set *NumMins* to any whole number allowing us any custom minute interval. On line 12 we call the PriceData function asking for QQQQ data and passing in the tf variable which is defined as our 5 minute bar interval.

Below are the available TimeFrameProvider objects to set the bar interval of your custom pricing data:

DailyTimeFrameProvider	NumDays	NumDays = 1 for daily
MinuteTimeFrameProvider	NumMins	NumMins = 1 for Minute
HourlyTimeFrameProvider	NumMins	NumMins = 1 for hourly
YearlyTimeFrameProvider	NumYears	NumYears = 1 for Yearly

Note: Any variable you create with the PriceData method will be disposed after the last bar has been calculated. If you are storing these in static variables (as in the previous sample) you must call PriceData on the first bar of the calculation otherwise you will get a null reference exception.

Global Variables/Memory

Version 5 of StockFinder adds global variables to RealCode. This allows you to share data between different RealCode items. Global variables should be used with care, and only in cases where there is no other way to implement your solution. Using global variables does involve some overhead and will slow down your RealCode calculation some. They should be used sparingly.

Good reasons to use global variables include:

- Caching long or slow calculation results to reuse in other indicators
- Storing data from a calculation that cannot be accessed directly by importing the indicator or condition to another RealCode
- Storing data that cannot be calculated in another item without duplicating work

Bad reasons to use global variables:

- Storing entire results of an indicator/condition into a global variable instead of referencing the indicator/condition in another RealCode item.

Candidates for global variables:

- Roll your own Money Management solution for conditions and backtesting
- Record prices or values when conditions pass to read in other conditions
- Save the most recent value of a long calculation (a cumulative indicator, market index, or some other resource intensive call)

The global variable object is a .net HashTable. A hashtable is like an array with a unique key for the index. You can store any value or object into a hashtable and reference it by a key you provide.

Because hashtables use unique keys to save their values, it's important that you do not use the same key name in different RealCode to store different values.

Below is a sample indicator that is storing the last price at which it crosses it's own 20 bar moving average:

```
8  If price(0) > price.AVG(20) And price(1) < price.avg(20, 1) Then
9      pass
10     Me.setSharedHashValue("PriceCrossUpMA20", price(0))
11 End If
```

The key line above is line 10. This is calling the `setSharedHashValue` method and providing a key "PriceCrossUpMA20".

To read the value set in the above condition, we simply call getSharedHashValue and provide the same key:

```
8  Static crossValue As Single
9  If isFirstBar Then
10     ' get the cross up value on the first bar
11     crossValue = Me.getSharedHashValue("PriceCrossUpMA20")
12 End If
13 ' check to see if the current bar is below the last cross up value
14 If price(0) < crossValue Then
15     pass
16 End If
```

The code sample above gets the PriceCrossUpMA20 value and then checks to see if the current price is below that value.

The two conditions above will work as written in a Filter, where conditions are evaluated one at a time, one after the other. They will not work in a multi-threaded context like two conditions on a WatchList sort, painting or a backtest.

The keen developer (or one who has worked with multithreading before) will notice that there is no mechanism for the second condition (the reader) to ensure that the first condition (the writer) has actually written a value to the global memory. This is ok for conditions used in Filters but in any other context (or when using multilple indicators on the same or multiple charts) you will need to implement a publish/subscribe model. Luckily for you, we implemented a simple solution for this so you do not have to roll your own.

To modify the above example and turn it into a publish/subscriber model we simply need to add 3 calls, 2 to the publisher (writer) and 1 to the subscriber(reader).

```
7  If isFirstBar Then
8      Me.NotifyDataChanging("PriceCrossUpMA20")
9  End If
10 If price(0) > price.AVG(20) And price(1) < price.avg(20, 1) Then
11     pass
12     Me.setSharedHashValue("PriceCrossUpMA20", price(0))
13 End If
14 If isLastBar Then
15     Me.NotifyDataReady("PriceCrossUpMA20")
16 End If
```

In the screenshot above, we hav e modified our first condition to include a check for the first bar (line 7). We notify any subscribers that we are about to change the data with the key "PriceCrossUpMA20". On line 14-16 we call `NotifyDataReady` with the same key "PriceCrossUpMA20".

Subscribers simply need to wait for the data ready call before reading the value out of the shared hash. We do this by calling `WaitForDataReady`:

```
8  Static crossValue As Single
9  If isFirstBar Then
10     ' Wait for the data to be ready. Time out after 2000 MS (2 seconds)
11     If WaitForDataReady("PriceCrossUpMA20", 2000) Then
12         ' get the cross up value on the first bar
13         crossValue = Me.getSharedHashValue("PriceCrossUpMA20")
14     Else
15         ' if WaitForDataReady returns false, our timeout was reached
16         crossValue = -1
17     End If
18
19 End If
20 ' check to see if the current bar is below the last cross up value
21 If crossvalue > 0 AndAlso price(0) < crossValue Then
22 '   pass
23 End If
```

Well, it is slightly more complex than calling `WaitForDataReady`. Calling this function will pause your calculation until `NotifyDataReady` is called with the same key, or until your timeout parameter is reached. In the above example we pass in a timeout of 2000 milliseconds (2 seconds) . If `NotifyDataReady` is called before our 2 seconds is reached, `WaitForDataReady` will return true and we are safe to get the new value, if it returns false, then our timeout was reached and we need to make sure we don't perform any calculations on the crossValue variable. I accomplish this by setting it to a negative number and then test to make sure it's positive before performing any other conditions.

That's it! You can store any value in the setSharedHashValue call including arrays, collections or any other .net object or primitive you choose. All the calls are thread safe and do not need to be synchronized. You may need to adjust the WaitForDataReady timeout parameter if your publish calculation takes a long time to finish.

Chapter 9 - Code Examples

This chapter has some basic code examples to use in RealCode.

THESE EXAMPLES ARE FOR EDUCATIONAL PURPOSES ONLY. THE AUTHOR DOES NOT RECOMMEND THAT YOU USE ANY SUCH CODE AS TRADING SIGNALS, STRATEGIES OR TO RECOMMEND A BUY OR SELL OF ANY SECURITY. THE FOLLOWING EXAMPLES ARE ONLY TO DEMONSTRATE THE DESIGN AND IMPLEMENTATION OF REALCODE.

Calculating the Net/Percent Change for a specific number of bars

Level: Beginner
Concepts Used:

- RealCode Indicator
- Price Offset
- User Input Variables

NOTE: This method of calculating the net or percent change is obsolete with the built-in Price.NetChange an Price.PercentChange functions. Please see chapter 5 for more details. It has been included for educational purposes only.

This example will calculate the net change between the current bar and X number of bars ago. The period for the change will be specified by a user-supplied variable from QuickEdit.

Net Change:

```
'# period = userinput.integer = 1
plot = price.Close - price.close(period)
```

Percent Change:

```
'# Period = UserInput.Integer = 1
plot = ((price.close - price.close(period)) / price.close)  * 100
```

Plotting the number of up/down bars in a row

Level: Beginner
Concepts Used:

- RealCode indicator
- Static Variables

This code example counts the number of consecutive bars in a row that the stock has moved up (or down) and plots that number as an indicator. If there is no change between bars the counter remains the same. This uses a static variable for the up bar counter.

```
Static UpCount As Integer

If me.IsFirstBar then
     UpCount = 0
```

```
End If

Dim netChange As Single = price.Close - price.Close(1)

If netchange > 0 then
     UpCount +=1
Elseif netchange < 0 then
     UpCount = 0
End if
Plot = UpCount
```

Checking for volume surge at the last hour of the trading day

Level: Intermediate
modiConcepts used:

- RealCode Condition
- Reading Indicator Values from a Chart
- UserInput Variable
- Checking for a Specific Date/Time

This little code example uses the Volume Surge indicator and a minute to chart to test for volume surge in the last hour of the trading day (and the last half hour in example 2). By default the volume surge must be 2 times it's average volume. Set the SurgeAmount variable in QuickEdit to change it from 2x to any other value. The key lines of code to check for the hour of the trading day are `Me.CurrrentDate.Hour`. This property returns the current hour (in 24 hour format) for the currently calculating bar.

To create this indicator add *Volume Surge* to your chart and set the chart bar interval to 1 minute. Click on *RealCode Editor – Condition*. Paste the following code into the editor:

```
'# VS = indicator.VolumeSurge
'# SurgeAmount = UserInput.Single = 2
If Me.CurrentDate.hour >= 15 AndAlso VS.Value >= surgeamount Then
   pass
End If
```

Here is a modified version of the above that checks only the last half hour of the trading day:

```
'# VS = indicator.VolumeSurge
'# SurgeAmount = UserInput.Single = 2
If Me.CurrentDate.hour >= 15 AndAlso VS.Value >= surgeamount Then
   If Me.currentdate.hour = 16 Then
        pass
   ElseIf Me.currentdate.minute >= 30 Then
        pass
   End If
End If
```

Million Shares Traded per Bar

Level: Intermediate
Concepts Used:

- Static Variables
- UserInput Variables
- Custom Timeframing
- Bar (OHLC) Output

In this code example we build the open/high/low and close for every 1 million shares traded. The 1 million is using a UserInput so you that value can be set to any number from QuickEdit to change the calculation (change the value to 5 for 5 million shares traded for example). The key to this code is that besides setting the plot value we also set `HighValue`, `LowValue` and `OpenValue`. Setting these values along with `Plot` will build a bar instead of a line. When building an indicator, setting the Plot equal to Single.NaN (not a number) will cause the indicator to skip the current bar. This way we accumulate the high and low values until we pass our million shares mark and finally output the open,high,low and last values.

To create this indicator create a new line chart and add a new RealCode indicator from the RealCode Editor menu. Remove the Price History Pane. Be sure to change the line style of your plot from *Line* to *OHLC*, *HLC* or *Candlestick* (left click on plot and change *plot style* from the dropdown)

```
'# Million = UserInput.integer = 1
Static o As Single
Static h As Single
Static l As Single
Static volCount As Integer

If Me.isFirstBar Then
   o = price.open
   h = price.high
   l = price.low
   volcount = volume.value
   plot = Single.nan
   Exit Function
End If
If Single.IsNaN(o) Then
   o = price.open
End If

h = System.math.max(h,price.high)
l = System.Math.min(l,price.Low)
volcount += volume.value

If volCount  > Million * (10000) Then
   plot = price.Last
   highValue = h
   lowValue = l
   openValue = o
```

```
      volCount = 0
      h = Single.MinValue
      l = Single.maxvalue
      o = Single.NaN

   Else
      highValue = Single.NaN
      lowValue = Single.NaN
      openValue = Single.NaN
      plot = Single.NaN
   End If
```

Creating Cyclical Charts: Average Monthly Percent Change

Level: Intermediate/Advanced

Concepts Used:

- Static Variables
- Custom Timeframing
- Custom Date Output
- Arrays

This code example creates a cyclical chart (based on months) and outputs the Average percent change for a stock for every month of the year. This plot is designed to be on it's own Chart with the Price history pane removed. This plot will output 12 values (one for every month) using the first date of the month on the date scale. Improve the look of this chart by changing the Line Style to Bar, zoom all the way out and scroll all the way back so you can see all 12 months. For the best performance you should set the bar interval to daily. The sample code uses a special collection object called A Generic List. The generic lists are stored in an array with a length of 12 (one for each month of the year). Generic lists are dynamic arrays that are strongly typed (you specify the type when declaring the variable). For more information on Generic collections see the .net Framework API. This code loops through the pricing data detecting when the data for the month has changed. When it detects a new Month it calculates the percent change from the first close of the month and adds it to a Generic.List for that month. This continues until the last bar is reached and we go average all the percent changes for every month and output the results.

```
   Static monthChanges(11) As System.Collections.generic.list(Of
Single)
   Static PrevMonth As Integer
   Static FirstClose As Single

   plot = Single.nan
   If isFirstBar Then
      prevMonth = currentDate.month
      FirstClose = Price.Close
      For i As Integer = 0 To 11
           monthChanges(i) = New System.Collections.generic.list( Of
Single)
      Next i
```

```
   ElseIf Not Me.islastbar Then
      ' not the first or last bar. record the net change from the
monthly open
      If CurrentDate.Month <> PrevMonth Then
            Dim monthPercentChange As Single = ((price.close(1) -
FirstClose) / FirstClose) * 100
            monthChanges(prevMonth - 1).add(monthPercentChange)
            prevmonth = currentdate.month
            firstclose = price.close
      End If

   Else
      ' Last bar of the calculation. Average each month
      For i As Integer = 0 To 11
                  Dim avg As Single = 0
            Dim outDate As Date = New Date(Date.now.year -1 ,i + 1,1) '
set it to the first day of the month for this year
                  For j As Integer = 0 To monthChanges(i).count -1
                        avg += monthChanges(i).item(j)
                  Next
                  avg /= monthchanges(i).count
                  Me.addtooutput(outdate,avg)
      Next

   End If
```

Simulating an Alert with RealCode Conditions

Level: Advanced

Concepts Used:

- IsLastBar
- Undocumented Functions
- Playing a wave file

This code example is a Hack (technical term for doing something that the program never intended to do) that has been posted on the Worden.Com forums by one of the developers. The original post can be found here: http://www.worden.com/training/default.aspx?g=posts&t=28507

WARNING: The following is not supported by Worden Brothers, Inc. proceed at your own risk!

Start by adding a new RealCode Condition to your Chart. As part of your conditional testing to see if your Condition passes add a check for `Me.IsLastBar = true` this will only perform the test if it's on the last bar (for a performance boost, you can perform this check first and short circuit the rest of your conditions).

Below is a code example for testing if the net change is > 0 and it's the last bar. If so it plays a wave file.

```
If Me.isLastBar AndAlso Price.NetChange > 0 Then
```

```
    pass
    LokiStatic.PlaySound("c:\alert.wav")
End If
```

Replace the `c:\alert.wav` with the path to any valid wave file on your computer. You should replace the net change condition (`Price.close > price.close(1)`) with something that is not a very common condition, otherwise your alert will fire over and over again when we start scanning for this Condition. Click *apply* and close the code editor window. This will now play the wave file anytime you browse to a symbol where the Condition is true for the last bar.

To make this alert fire for any symbol in your WatchList, drag the Condition to your WatchList and check the column header to add it to the active scan (there should be a flag next to the column header if it's part of the scan). It should now play the sound for every symbol that passes your Condition. Again, make sure your Condition is something that happens infrequently for a list of symbols otherwise your alert will fire for every symbol in your list that passes and will play the sound over and over again.

If you want a popup Alert every time it passes you can add this line of code after the PlaySound line:

```
    Me.ShowMessage("Net Change alert on" & Me.CurrentSymbol)
```

Alternatively if you want to log every alert that passes (in case you're away from your desk) you can use the following line with/instead of popping up a message or playing a sound:

```
Me.log.info("Alert passed on " & Me.currentsymbol & " with a price of
" & price.close)
```

Again, make sure your alert Condition is rare, otherwise you will get many popup windows for every symbol that passes and will have to hit OK for every symbol (if you're using the ShowMessage call).

You can also use an existing Condition and drag it into your alert Condition to create an alert when the existing Condition passes.

```
  '# SC = condition.ScanCondition.3

  If Me.isLastBar AndAlso sc.Value Then
    Me.log.info("Alert passed on " & Me.currentsymbol & " with a
price of " & price.close)
    pass
  End If
```

Using a RealCode Class to create your own price history manually

This example is purely an educational one, in that it gives you nothing that you don't already have available to you in StockFinder by default. This example will loop through the stock price to output an open,high,low and close value. It shows you how to enumerate the price array from start to finish and to create your own data.

If you understand this basic code, you should be able to modify it to do some of the previous examples (Monthly cyclical charts would be a good candidate to re-write as a class and do your own price history enumeration)

When re-producing the example, you do not need to provide the first two lines (`Public partial class`) or the last line of the class (`end class`). You should simply insert the `Sub New` and the `Overrides Plot` methods.

```
Public partial Class RealCodeIndicator
   Inherits RealCodeIndicator_base
   Sub New
        MyBase.New
        MyBase.AutoLoop = False
   End Sub
   Public Overrides Function Plot() As System.Single

        For i As Integer = 0 To price.count -1
             MyBase.AddToOutput(price.bar.datevalue(i),price.bar.Op
             enValue(i),price.bar.highValue(i),price.bar.lowvalue(i
             ),price.bar.value(i))
        Next

   end function
End Class
```

The key to the code above is the constructor, sub new. In the constructor we set the `AutoLoop` property to false. This tells the base class to call our plot function once and we will provide all the data for our calculation.

Once `AutoLoop` is set to false, we can enumerate the price history by using the `Price.Count` property and calling `Price.Bar` to get the raw data arrays for the price history.

You can also import other indicators from the chart and call their `Bar` and `Line` properties to get the raw values for the indicator.

There is one important thing to not about performing a custom loop with imported indicators. They will not be interpolated and the price and indicator array indexes will probably not match in dates. This means you will need to perform your own interpolation or find the indexes that match on your own. As a general Condition all indicators on the chart should be in order from oldest bar to latest, so index `0` will be the oldest value and index `Count - 1` will be the newest.

Chapter 10 – Code Samples for StockFinder Workbook

The code samples in this chapter are RealCode implementations of the exercises from the StockFinder 5 Workbook.

Exercise 1 – Unusually High Trade Range

Goal: To isolate stocks with a higher than normal trading range on the most recent price bar.

Solution: We perform our own simple moving average of the Price trade range. Compare our current trade range with our moving average

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition:Above Average Trade Range
'|*****************************************************************
'# avgPeriod = userinput.integer = 30
Static sum As Single
Dim TradeAvg As Single
If isFirstBar Then
    sum = 0
End If
If currentindex = avgPeriod - 1 Then
    ' compute the average for the first 30 bars (or whatever avgPeriod is set to)
    For x As Integer = 0 To avgperiod - 1
            sum += price.TradeRange(x)
    Next
ElseIf currentindex > avgperiod - 1 Then
    ' update our average, add the current trade range and subtract the trade range from 30 bars ago
    sum = sum + price.TradeRange - price.TradeRange(avgperiod)
    tradeavg = sum / avgperiod
    ' check to see if the current trade range is above average
    If price.TradeRange > tradeavg Then
            pass
    End If
End If
```

Exercise 2 – Above Long and Short Averages

Goal: To isolate stocks above their 20 and 50 bar moving averages

Solution: Use a user input variable for the short (20) and long (50) period averages. Use the built in AVG function to get the moving average values

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition:Above short and Long averages
'|*****************************************************************
'# longPeriod = userinput.integer = 50
'# shortPeriod = userinput.integer = 20
If price(0) > price.AVG(shortPeriod) AndAlso price(0) > price.avg(longPeriod) Then
    pass
End If
```

Exercise 4 – ADX Values over 40

Goal: Isolate stocks with an Average Directional Index (ADX) value greater than 40

Solution: Use the ADX indicator from the library and compare it with a user input variable.

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition:Adx Greater Than Value
'|*****************************************************************
'# ADX = indicator.Library.Directional Movement DI ADX
'# greaterThan = userinput.integer = 40

If ADX.value > greaterThan Then pass
```

Exercise 10 – Price down Ten Percent in a month

Goal: To isolate stocks which have moved down ten percent or more over the last 21 trading days

Solution 1: Use the Percent Change function with a user input parameter to specify the change period and compare it with a user input variable for the degree of the percent change.

Solution 2: We included a second solution here that uses a calendar month instead of 21 trading days. This solution gets monthly price data for the current symbol and performs a percent change comparison vs the change amount.

Solution 1: Price down 10 percent in last 21 days

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition:Price Percent Change
'|*****************************************************************
'# changePercent = userinput.integer = 10
'# bars = userinput.integer = 21

If price.PercentChange(bars) <= (0 - changepercent) Then pass
```

Solution 2: Alternate version using true calendar months

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition:Price Percent Change
'|*****************************************************************
'# changePercent = userinput.integer = 10
Static mp As RealCodeIndicator
If isFirstBar Then
    Dim tf As New MonthlyTimeFrameProvider
    tf.numMonths = 1
    tf.dostreaming = False
    mp = PriceData(Me.currentsymbol, tf)
End If

If mp.percentchange <= (0 - changepercent) Then pass
```

Exercise 12 – Filter out low volume stocks

Goal: To filter out stocks in a WatchList that trade on average less than 200,000 shares each day

Solution: Test for the volume value greater than 2000

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition:High Volume
'|*****************************************************************
If volume.value > 2000 Then pass
```

Exercise 15 - MACD Histogram turning up

Goal: To isolate stocks where the MACD Histogram indicator is crossing from negative to positive

Solution: Test the MACD Histogram indicator from the library

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition:MACD histo turning positive
'|*****************************************************************
'# MACDHisto = indicator.Library.MACD Histogram

If MACDHisto.value > 0 AndAlso MACDHisto.value(1) <= 0 Then pass
```

Exercise 17 – Moving Averages Crossing

Goal: To identify stocks with two moving averages (of price) crossing.

Solution: Compare Price.AVG functions with each other

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition:Crossing MA
'|*****************************************************************

If currentindex < 50 Then
    Me.setindexinvalid
End If

If price.AVG(20, 1) < price.AVG(50, 1) AndAlso price.AVG(20) >= price.avg(50) Then pass
```

Exercise 18 – Oversold Stochastics

Goal: To identify stocks with a Stockastics Value below twenty

Solution: Use the Stoc child function to test for a value below 20

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition:Oversold Stochastics
'|*****************************************************************

If price.stoc(12, 2) < 20 Then pass
```

Exercise 21 – Price beween $10 and %50

Goal: to isolate stocks that trade between $10 and $50 per share

Solution: Check the price.last value

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition:Price between 10 and 50
'|*****************************************************************
If price.last >= 10 AndAlso price.last <= 50 Then pass
```

Exercise 25 – Price Above a moving average

Goal: to isolate stocks with price bavoe of below their moving average

Solution: Check for price value above the AVG child function

```
'|*****************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition: Price above 20 period ma
'|*****************************************************************
If price.last > price.avg(20) Then pass
```

Exercise 26 – Stocks with above average volume

Goal: to find stocks which currently are experiencing an unusually high volume of trading.

Solution: Compare volume with the child AVG function

```
'|*************************************************************|
'|*** StockFinder RealCode Condition - Version 4.9 www.worden.com
'|*** Copy and paste this header and code into StockFinder *********
'|*** Condition: volume above 30 bar avg
'|*************************************************************

If volume.value > volume.AVG(30) Then pass
```