RealCode™ Programmers Reference

V1.6 – Revised on 10/22/2008

Updated for version 4.0 Build 37

By Ken Gilb
Chief Software Engineer, Worden Brothers Inc.

Contents

What’s new for RealCode in StockFinder version 4.0

The latest version of StockFinder introduces minimal changes to the RealCode programming language. There is one new property available and one new user input.

By popular demand, we have also introduced saving the individual indicator or rule so you can use it from your own personal library or share it with other users. See chapter for 3 for information on Saving your RealCode item.

We have also overhauled the RealCode editor by adding a toolbar at the top and moving the tabs to the bottom. We also upgraded the editor engine to improve performance and fix some color coding bugs. The editor now also supports the Common and All tabs on the Code Tips to show the most commonly called items on an Object. The new editor toolbar allowed us to add the new save button and the Import Indicator/Rules button.

Rules and Indicators can now be referenced by using the newly mentioned Indicator/Rules button. See chapter 6 for more details.

New RealCode items in StockFinder 4.0

MyValue(offset)	Returns values you’ve already set in code. MyValue(1) returns the previous value set for Plot = . Allows you to lookback at your own data without needing to store it in a local or static variable.
Userinput.Date	Allows you to define a user input that is a date value.

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 other trading 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.

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.

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 If you’re an experienced develop you can skip over the introduction to VB and move on to the RealCode fundamentals.

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”.

You can view his blog online at http://blog.gilb.net/kuf . There are many more code examples and in depth articles dedicated to writing RealCode.

Additional Resources

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

Microsoft Online References:

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

Worden Brothers, Inc

- 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 customer.
- Try/Catch: http://blog.gilb.net/kuf - The Author's personal blog with RealCode examples and articles.

What types of RealCode can you create?

You can create the following RealCode items:

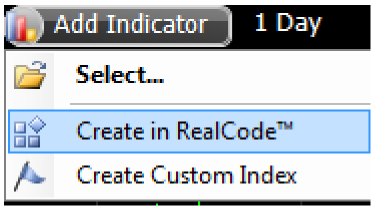
- Indicators (plots on chart)
- Rules (true/false rules to scan, filter, color or backtest)
- Paint Brush (indicator coloring based on code)
- Scans (via RealCode Rules applied to a WatchList)
- Filters(via RealCode Rules applied to a WatchList)
- Sorts (via RealCode Indicators or Rules applied to a WatchList column)
- BackScanner entry or exit rule. (RealCode Rule applied to entry/exit rule)

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

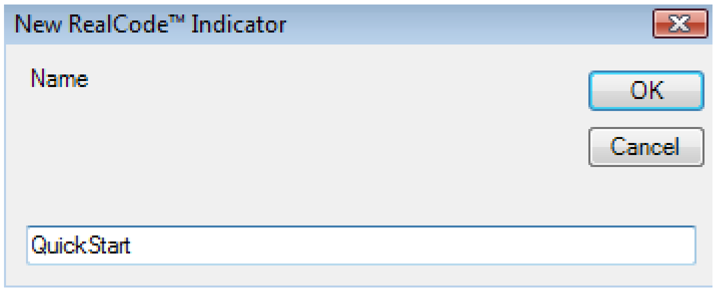
Quick Start

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 of a chart.
2. Choose Create in RealCode™



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 from our 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 and 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

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. Specifically RealCode classes are compiled into a special class called a Block. Blocks are compiled code that can be connected together in a block diagram to perform a calculation. Every plot, rule, scan or sort in the StockFinder application is performed through a block diagram.

When you write RealCode the application compiles it for you and creates a block diagram to perform your calculation. The diagram is always visible to you (simply right click on your RealCode item and select *block diagram*) 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. Did we say the word *compiled* enough? It's a sticking point because most trading languages are scripting languages that are slower to execute than compiled code) and that the .net framework and all the hundreds of pre-build 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[1] .

If you are not familiar with VB.net or with programming in general, the next few sections will go over basic programming principals.

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 rules) 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

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

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`[2]

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).

Conditionals (IF statements)

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` was 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
```

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

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
```

Operators

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 evalute 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 to make 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 rule 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 late and trying to remember what the hell 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

Whether creating a RealCode Indicator, Rule 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 rule 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 6.

The Price Chart:

All RealCode is based on the Chart that owns (displays) the RealCode. When StockFinder calls your RealCode it prepares all the Pricing and Volume data for the current symbol and the selected time frame. Starting at the oldest “bar” (Open/High/low/Close values) for the chart’s timeframe, StockFinder calls your RealCode to get a value for current bar. It then stores the resulting value and advances to the next bar, calling your code once more for the second value. This repeats until the latest bar of data and it then constructs the appropriate line, paintbrush or scan based on the values you set in code. 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 Time Frame of the chart the “current” bar could be a day, month, year, hour or minute or any other custom timeframe provided by the Chart. This means if your chart 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.

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
Rule	Pass (or Fail)	Boolean
Paint Brush	PlotColor =	Drawing.Color

Lets’ say you wanted a basic rule 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 rule 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 rule 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.

Advanced Tip:

If there is not enough data to calculate your rule (because you are calling for data x bars ago) RealCode will not output a value until there is enough data for your rule 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.

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 Rule. You can access QuickEdit by simply left clicking on the Indicator or rule on the Chart.

Additionally, your RealCode can access the other Rules and Indicators 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 rule on your chart. 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 on the Chart. 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.

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.

Advanced Tip:

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 timeframe 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 timeframe. 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 timeframe of the chart). If you mix your own custom timeframe with another indicator that is on a longer or shorter timeframe, the chart will automatically adjust to add more dates to accommodate the longer timeframe.

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). Also when running a scan or showing scan lights in the WatchList, you do not need to test historically when your rule passed, you only need to test for the most current bar (to test if it's passing now). When your indicator or rule 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 rule has enough data to pass.

RealCode Properties and Methods

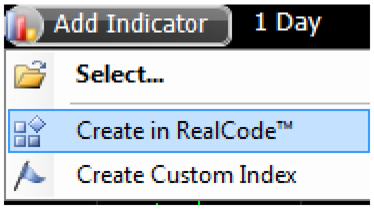
Besides all the built in .net classes, methods and functions, RealCode has added some additional properties and methods. These are always available in the RealCode editor.

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)
Me.CurrentDate	Date	Returns the date and time for the currently calculating bar
Me.CurrentSymbol	String	Returns the symbol for the currently calculating bar
Me.CurrentIndex	Integer	Returns the 0 based index for the currently calculating bar.
DateValue	Date	Returns the date for the currently calculating date. Same as Me.CurrentDate
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 (CurrentIndex = 0)
Me.isLastBar	Boolean	Returns true if calculating for the last bar of the calculation. (CurrentIndex = Price.Count -1)
Me.Log.Info(txt)	String Parameter	Call to write to the Info Debug Log
Me.Log.Warn(txt)	String Parameter	Call to write to the Warning Debug Log
Me.Log.Err(txt)	String Parameter	Call to write to the Error Debug Log
Me.Log.Err(txt,ex)	String Parameter, Exception parameter	Call to write to the Error Debug Log with a string and an exception. Will include exception stack trace

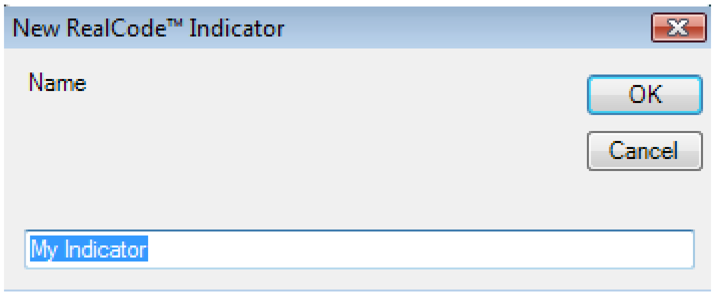
Table 1 – RealCode Properties and Methods

RealCode Indicators

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



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 timeframe. 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 timeframe 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:

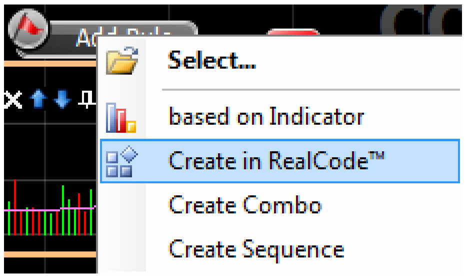
```
Plot = 0

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 Rules

To create a new RealCode rule: click on the *Add Rule* button at the bottom of the Price Pane and select *Create in RealCode*. To edit an existing RealCode rule, right-click on the rule "bubble" and choose *Edit RealCode*. See Figure 1 for an example of a Rule Bubble.



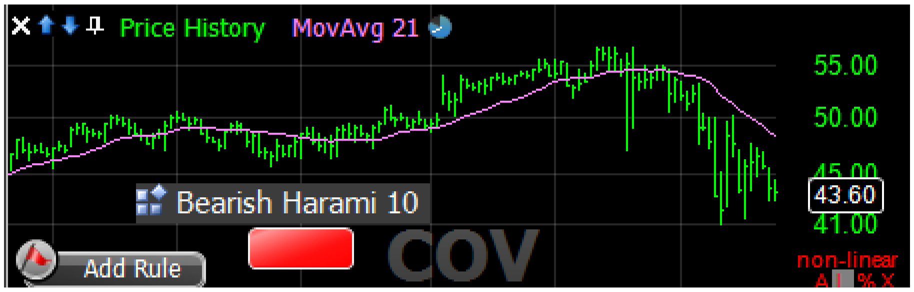


Figure 1 - RealCode rule bubble in red at bottom of pane

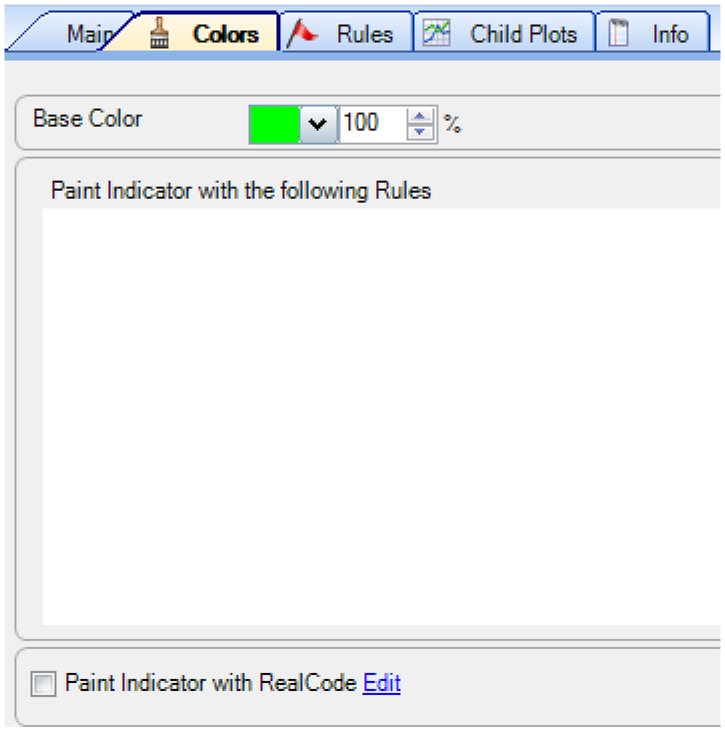
Like an indicator, your RealCode rule runs in the context of the current chart's symbol and TimeFrame. By default RealCode rules are in the fail (false) state so you only need to tell the RealCode rule when it passes. Rules can be used to paint indicators, filter or scan a list, or perform a condition in Backscanner.

When your rule 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.

You can paint your indicators by your RealCode rules, but sometimes you might want some special logic to perform your painting. To create a RealCode Paint Brush, Left click on the indicator you wish to paint and select the Colors tab from the editor. Check the *Paint Indicator with RealCode* checkbox. Click the *Edit* link to open the RealCode Editor.



Base Color 🟩 100 %

Paint Indicator with the following Rules

☐ Paint Indicator with RealCode Edit

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 Rule in that they both test for a Boolean value. Unlike a Rule, 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

All RealCode items are owned by the Chart they are created on. You can save RealCode Indicators and Rules by clicking the Save button on the editor or right clicking on the item and choosing save. To save a Paint Brush, you simply need to Save the Chart or Layout. All charts can be saved by clicking on the Save icon (at the top of the chart to the right of the TimeFrame picker) and choosing Save Chart to save an existing Chart or Save Chart As to save the chart under a new name. A saved chart can be reloaded from the charts button into any other Layout.

Alternatively, instead of saving the Chart directly, you can simply save the Layout that contains your Chart with the RealCode. You can save your Layout by clicking on the Save button at the top of the application or from the Save As button under the File menu.

If your RealCode references any external items (other rules or indicators) then you will not be able the save the item itself and you will need to save the chart that includes your RealCode and the items you reference. For more information on referencing external items, see chapter 6.

Chapter 4 – 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.

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 wit hteh 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 thecurrent bar
TradeRange(offset)	Single	Returns the High minus the low for the offset number of bars ago
Me.TimeFrame	TimeSpan	Returns the currently selected timeframe as a TimeSpan object.

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
   Max = 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
   Max = System.Math.Max(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)

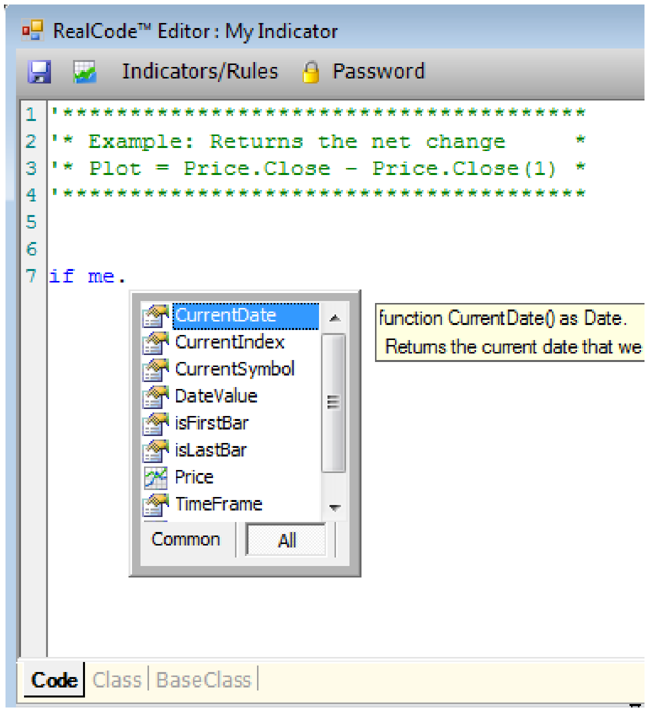
```
Plot = Price.High(1) - Price.Low(1)

Plot= Price.TradeRange(1)
```

TimeFrame

`Me.TimeFrame` returns a `TimeSpan` object that represents the currently selected TimeFrame on the chart that is hosting the RealCode. The following code is an example of how to use the TimeSpan object to determine the number of minutes or days for the current timeframe:

```
If me.TimeFrame.Days > 0 then
   ' Dailly or higher timeframe
   Select Case me.TimeFrame.Days
         Case  1
               ' daily chart
         Case 2
               ' 2 day chart
   End Select
Else
   ' Intraday timeframe
   Select Case me.TimeFrame.Minutes
         Case  1
               ' 1 Minute chart
         Case 2
               ' 2 Minute chart
   End Select
End if
```

Chapter 5 - The RealCode Editor

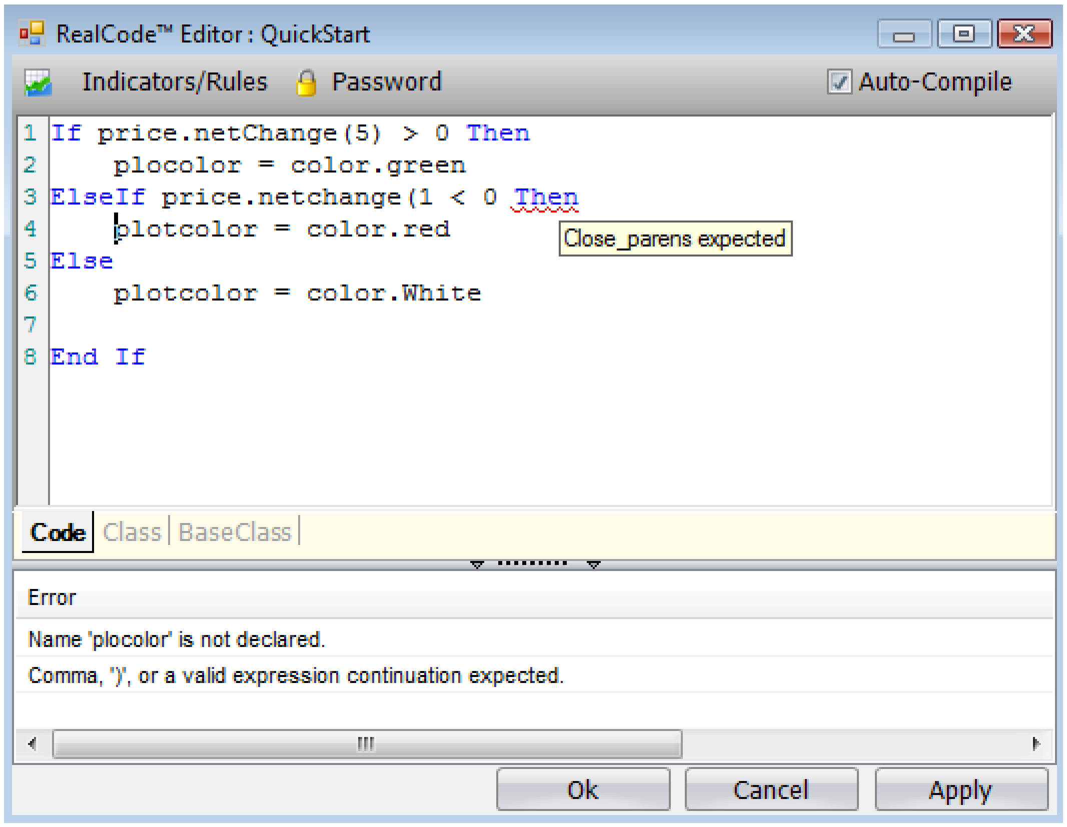


```
'****************************************
'* Example: Returns the net change     *
'* Plot = Price.Close - Price.Close(1) *
'****************************************


if me.
```

function CurrentDate() as Date.
Returns the current date that we

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.

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.



```
1 If price.netChange(5) > 0 Then
2     plocolor = color.green
3 ElseIf price.netchange(1 < 0 Then
4     plotcolor = color.red
5 Else
6     plotcolor = color.White
7
8 End If
```

Error
Name 'plocolor' is not declared.
Comma, ')', or a valid expression continuation expected.

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. Lastly, you can dock the code editor window into your Layout.

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 6 - Accessing External Indicators, Rules and Parameters.

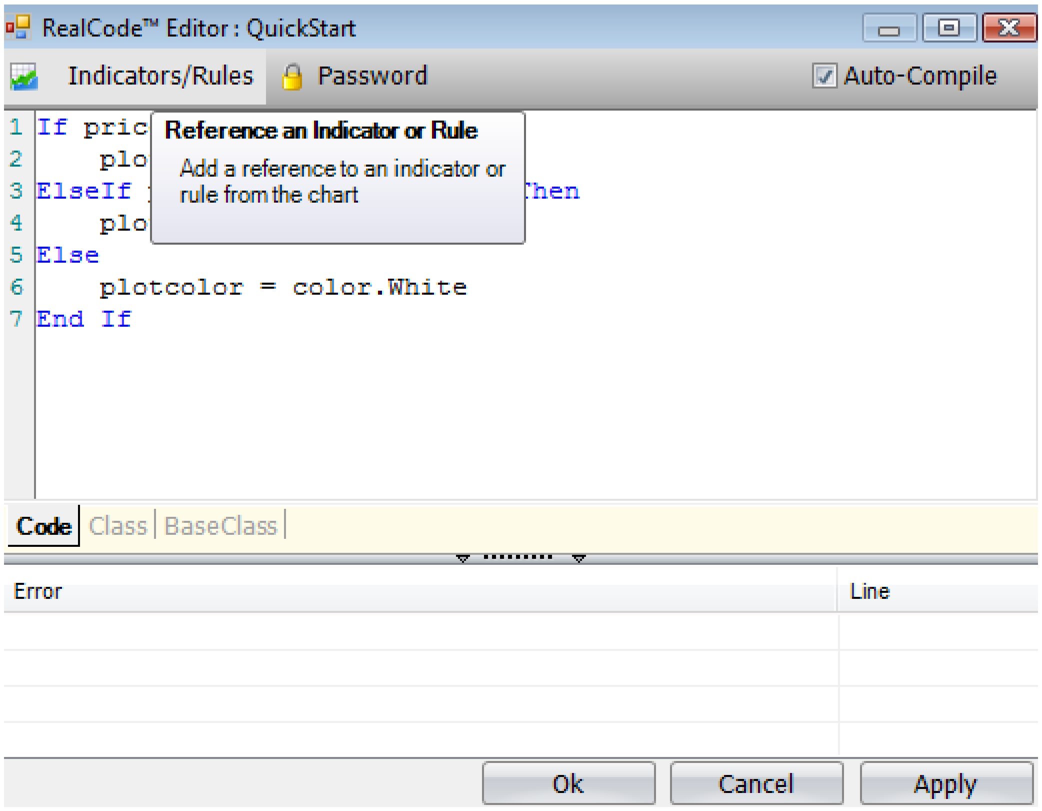
A RealCode item can read the values of the existing *Indicators* and/or *Rules* on your Chart (including other RealCode Items). You can also define *User Input Variables* that can be assigned at run time via *QuickEdit* . This is a really great feature of RealCode. This means you can daisy chain calculations to create more complex and aggregate data and use variable parameters for performing your calculations. Instead of having to write a calculation (say like moving average), you can simply apply the moving average indicator and import it into your RealCode.

External data is referenced through a *Directive*. A RealCode directives start with the comment symbol (') and then the pound symbol (#). **Error! Reference source not found.** and **Error! Reference source not found.** show RealCode directives to reference Indicators and Rules from the chart. 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. This is a StockFinder RealCode™ exclusive 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 on your Chart can be accessed by your RealCode.

The easiest way to access an Indicator is to add it to your Chart from the Indicator library. Once added to the chart, you can click the Indicators/Rules button to include it in your RealCode (see figure below)

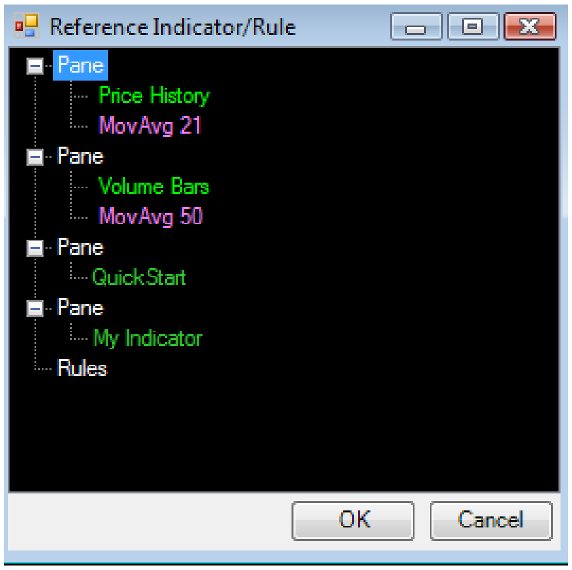


```
If pric
    plo
ElseIf              Then
    plo
Else
    plotcolor = color.White
End If
```

Reference an Indicator or Rule

Add a reference to an indicator or rule from the chart

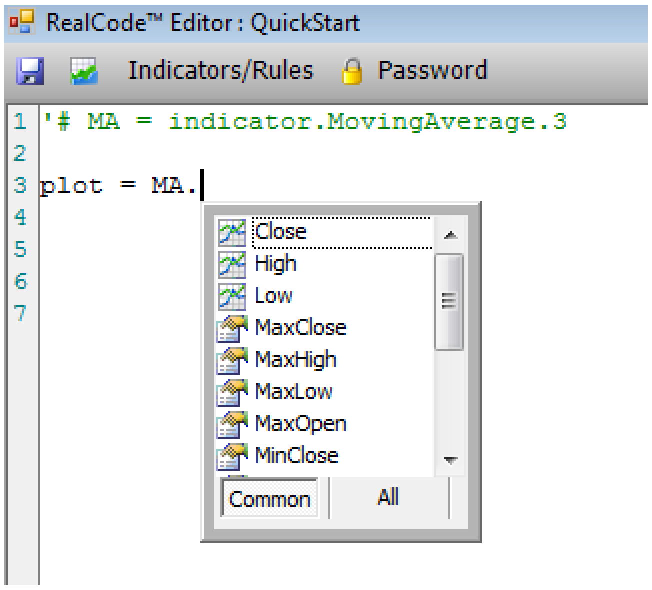
When you click on the Indicators/Rules button, the list of items from your chart will be displayed. Click on the item you wish to reference and click *Ok*.



In the above image, if we were to click on one of the moving averages and hit ok, it would add the following line of code:

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

This line of code starts with our input directive `'#` and then declares a variable name for the indicator. In the above example MA is the variable name we will use in code to reference the moving average. MA was assigned by default when we added the indicator to our code but you can change the variable name to anything you like. 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):



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

plot = MA.
```

Figure 4 – Referencing an indicator in RealCode

The Moving Average Indicator can now be accessed using `MA.Value` to get the value at the current bar, or `MA.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 add as many *Indicators* into your RealCode as you like. The only requirement for referencing an indicator is that it must exist on the same chart as your RealCode.

Importing Rules

Besides being able to access the existing Indicators on your Chart, you can also access the existing conditions on your Chart. This allows you to test if the current bar is passing the rule. Just like referencing an existing indicator, you can use the Indicators/Rules button to import an existing rule into your RealCode.   With a rule referenced in your RealCode you can call the `Value` property to see if the rule is passing for the current bar, or call `Value(x)` to test if the rule is passing or failing x number of bars ago. The value property returns a Boolean data type.

UserInput Variables

Sometimes when performing a calculation, you would like to change a few variables without needing to open up and recompile the code. Other times you just might want to add some user customization by allowing the user of your code to change a parameter. *UserInput* variables are the way to get external data into your code that can be changed at calculation time (run-time).

UserInput variables also use the pre-compiler directive (`'#`) just like the Indicators and Rule inputs. There are 5 types of data you can declare as a UserInput variable: Single, Integer, String, Date and Color. They all directly map to the built in .net Data types. The last one, Color, is only available when creating a Paint Brush.

To declare a UserInput variable start with the directive syntax (`'#`) followed by the variable name you will reference in code. You assign the UserInput data type with `= UserInput.`*`dataType`* replacing the *datatype* with the type of variable you wish to declare. Lastly, you can set a default value at the end of your declaration with `=` *`value`* (where *`value`* is a valid value for the assigned data type).

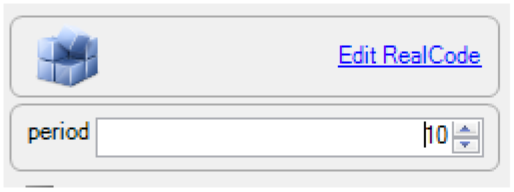
Below is an example declaration of the 5 UserInput types and how to declare them with default values:

```
   '# period = UserInput.Integer = 5
   '# testDate = UserInput.Date
   '# CrossOver = UserInput.Single = 10.5
   '# ColorPaint = UserInput.Color = Color.Orange
   '# SymbolToIgnore = UserInput.String = “MSFT”
```

The following code outputs the net change of price using a UserInput variable to define the number of bars ago to calculate the net change:

```
'# netChangePeriod = UserInput.integer = 1
plot = price.close - price.close(netChangePeriod)
```

The period for the net change can now be set by opening the editor for the indicator (left click on the indicator or right click and choose *Edit*) and changing the value (Figure 5)



period 10

Figure 5 - QuickEdit with UserInput

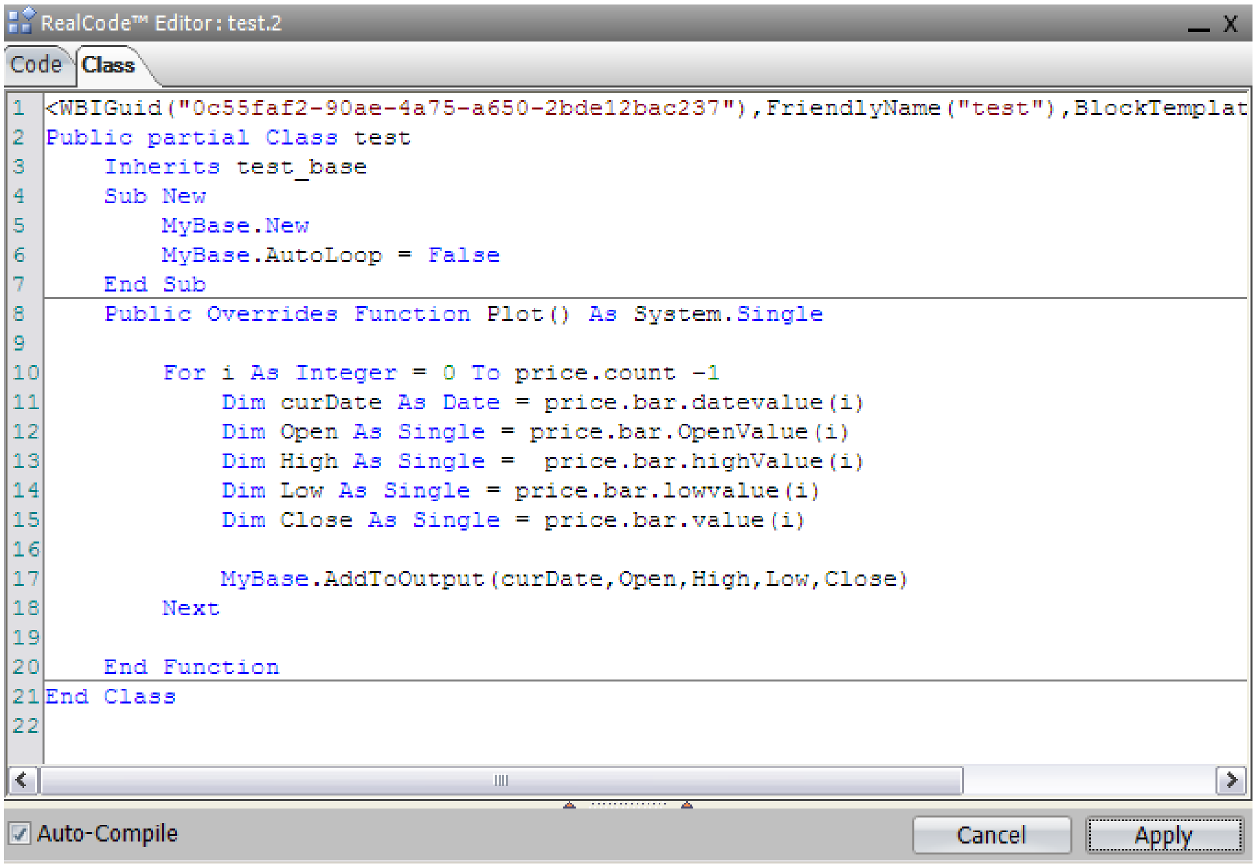
Chapter 7 - RealCode Classes

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 top of the RealCode Editor (see Figure 6).



```
<WBIGuid("0c55faf2-90ae-4a75-a650-2bde12bac237"),FriendlyName("test"),BlockTemplat
Public partial Class test
    Inherits test_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
            Dim curDate As Date = price.bar.datevalue(i)
            Dim Open As Single = price.bar.OpenValue(i)
            Dim High As Single =  price.bar.highValue(i)
            Dim Low As Single = price.bar.lowvalue(i)
            Dim Close As Single = price.bar.value(i)

            MyBase.AddToOutput(curDate,Open,High,Low,Close)
        Next

    End Function
End Class
```

Figure 6- A RealCode Class with custom looping

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 (see Figure 6). 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.

Chapter 8 -RealCode Tips and Tricks

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 Rule). If your code requires a bit more complexity than a simple mathematical calculation then below are some tips to help you write more complex RealCode.

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. This 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: Build 15 introduces Classes to RealCode and you no longer need to use static variables to keep values between calls to your code. The above code can also be represented in a class as the following:

```
Public partial Class AdvanceCount
inherits AdvanceCount_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
```

Debugging

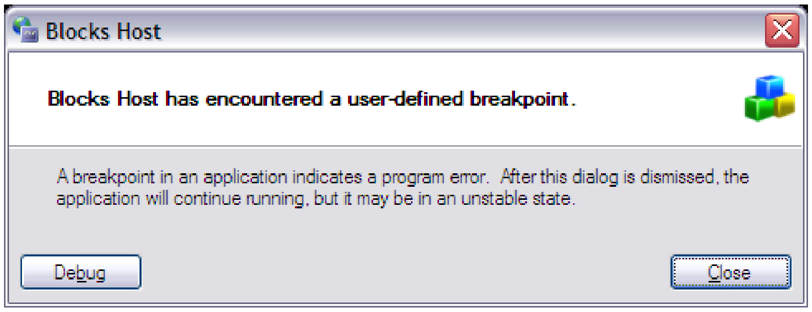
When your calculation doesn't look right, it's time to start debugging. There are a few basic options for debugging your RealCode. The first option is to write to the debug log. The debug log can be written to by using the `Me.Log` property. You can view the debug log from the *Help* tab. The Log property 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")`

Besides using the debug log, 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 System.Diagnostics.Debugger.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 time it's called. Example:

```
If Me.isFirstBar Then
   system.Diagnostics.debugger.break
End If
```

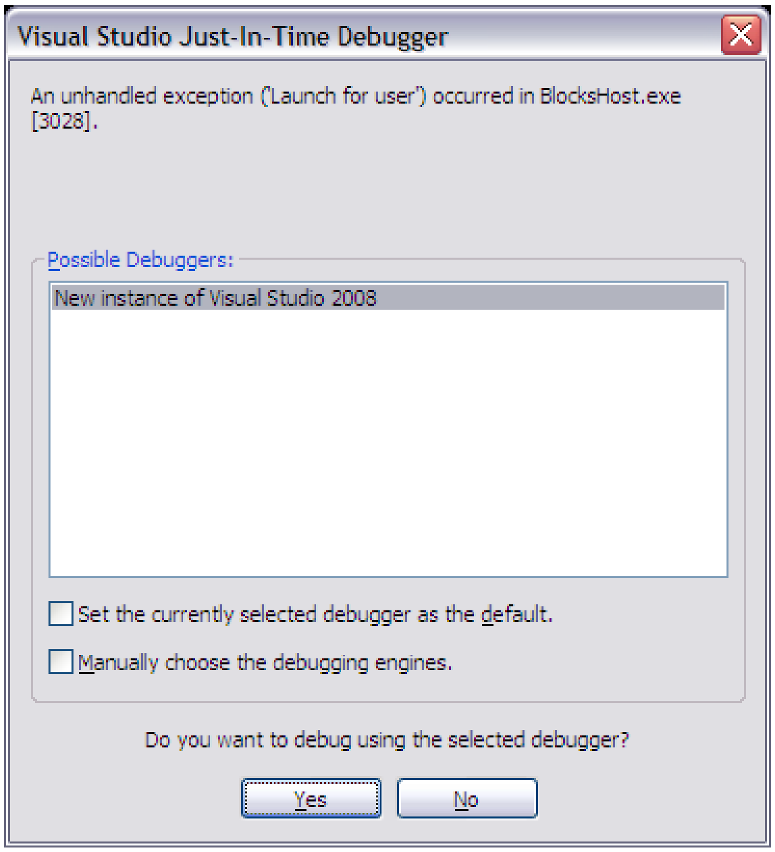
When the debugger hits that line of code it will prompt you to debug (**Error! Reference source ot found.**). Click the debug button. You will then be prompted for the Just-in-time debugger to use. Choose *New Instance of Visual Studi o.(*Figure 8*).* You will now be on your line of code in the Visual Studio Debugger and can use all the features for debugging your code.



Blocks Host has encountered a user-defined breakpoint.

A breakpoint in an application indicates a program error. After this dialog is dismissed, the application will continue running, but it may be in an unstable state.

Figure 7 User-Defined Breakpoint


Visual Studio Just-In-Time Debugger
Possible Debuggers:
New instance of Visual Studio 2008
Yes
No

An unhandled exception ('Launch for user') occurred in BlocksHost.exe [3028].

Set the currently selected debugger as the default.

Manually choose the debugging engines.

Do you want to debug using the selected debugger?

Figure 8 - Just-In-Time Debugger

Chapter 9 - Code Examples

Here are some code examples.

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.

Green Up / Red Down Bars (or user customizable colors)

Level: Beginner
Concepts Used:

- RealCode Paintbrush
- UserInput Variables

In this example we paint the classic green bars on up days, red bars on down days (white for no change). Using UserInput variables we can change the green/red/white colors in QuickEdit without needing to re-compile the code.

```
'# Up = userinput.color = color.Lime
'# Down = userinput.color = color.Red
'# Even = userinput.color = color.White

If price.Close > price.close(1) Then
   plotcolor = up
ElseIf price.close < price.close(1) Then
   plotcolor = down
Else ' no change
   plotcolor = even
End If
```

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 4 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

Concepts used:

- RealCode Rule
- 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 timeframe to 1 minute. Click on *RealCode Editor – Rule*. 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 Timframing
- 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
   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

If volCount  > Million * (1000000) 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 timeframe 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 Rules

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 rule to your Chart. As part of your conditional testing to see if your rule 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 rule. Click *apply* and close the code editor window. This will now play the wave file anytime you browse to a symbol where the rule is true for the last bar.

To make this alert fire for any symbol in your WatchList, drag the rule 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 rule. Again, make sure your rule 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 rule 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 rule and drag it into your alert rule to create an alert when the existing rule 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 test
      Inherits test_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 rule 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.

Appendix

Corrections from previous manual versions

Version 1.2 (and earlier) of the manual had incorrect code in the chapter on “Creating Cyclical Charts: Average monthly percent change”. The month percent change was being incorrectly calculated (it was inverted). If you have created any indicators based on that chapter you should correct this line of code to match below:

```
   Dim monthPercentChange As Single = ((price.close(1) - FirstClose) /
FirstClose) * 100
```