

## 20091220 StockChartX Settings 02

Malcolm Moore

© 20-Dec-2009

### Contents

Loading Historical Data into StockChartX.....	1
Understanding Charting Style Types - 1 .....	1
Structuring the Data Series .....	2
Adding a Series.....	3
Managing Date and Time Issues .....	4
Working with Julian Time.....	4
Working with Real Time.....	5
Loading Price Data into a Series.....	5
Using ML Reader with MetaStock Data.....	6

### Loading Historical Data into StockChartX

StockChartX is a rather complex object that includes a scalable flat file database within it, so that StockChartX can directly present historical data (as a number of “series”) on the screen in a visual form that is easily to interpret. In this same scalable database, StockChartX also holds the historical data for indicators that will be presented, either over the historical data series, or in separate panels with the time axis vertically aligned.

This process assumes a reasonably good knowledge about StockChartX before you start using StockChartX, so there is a bit of homework that needs to be covered first, explaining the various types of charts that can be graphed and then how the data can be loaded. The problem is that you have to know what form that you want your chart to take – before you load the data into a series in the StockChartX object, so my approach is to keep it simple and then expand on that – once the object has been somewhat ‘tamed’!

In these notes I am assuming that the historical trade data is based on End Of Day (EOD) data, and that this data is held in a MetaStock database.

#### ***Understanding Charting Style Types - 1***

Within StockChartX there are a number of Charting Price Style Types (just as Microsoft ® Word has Styles that include Font, Paragraph, Numbering etc in each Style). StockChartX includes a range of charting preset presentation characteristics that relate to charting / graphing presentations.

There are several Price Styles (that start with “ps”) and each of these Price Styles includes a family of charting presentations. At this stage we are concerned with the basics and we are focussing on the standard Price Style that is called psStandard and this is the default unless otherwise specified.

This code snippet shows the standard Price Style being specified.

```
stockChartX1.PriceStyle = psStandard
```

There are several Chart Price Styles that are built into StockChartX1 so that it can display prices, volumes and related values including indicators, and these come with a subset of Chart Style Type Parameters, which have their own nomenclature, starting with “st” for “style”.

Instead of dealing with a whole lot of numbers the StockChartX object has these parameters for charting price styles decoded into fairly plain English.

The four basic charting Style Types as the standard subset of psStandard are:

- **stCandleChart**, which provides the structure for candlestick using Open, High, Low and Close prices;
- **stOHLcchart**, which provides the structure for candlestick using Open, High, Low and Close prices;
- **stLineChart**, which provides a line chart based on a single series of data; and
- **stVolumeChart**, which provides a bar chart (usually) based on the volume of trades as a series of data.

There are several other types of charting types that are built into StockChartX but they will be covered later once the basics are well understood.

Sequentially, we need to tell StockChartX1, which Price Style that we want, then set the associated parameters where applicable, and then load the prices along with date/time.

But firstly the Price Styles! *(Note that when you type a price style in, when you come to the first dot you will get a drop-down list where PriceStyle is a table with a hand pointer (not a flying green brick)). This hand pointer tells you that you have to point the statement to something to tell it what it equals. Right after PriceStyle, enter an "=" sign (without the talkies) and a new drop down list will come with another hand pointer. Use the Up/Down keys to highlight the one that you want and then <Enter> on that.*

### ***Structuring the Data Series***

What we now do know is that we wish to load the EOD data (from an historical database) for a particular security into StockChartX, and the EOD data will include the Open, High, Low, Close and Volume data on a daily basis.

Before we start loading historical data into StockChartX, its internal database needs to be initiated so that the name of the security and the visual panel are associated. This is how to get started:

```
' Clean out everything in this object
StockChartX1.RemoveAllSeries
' We are loading the standard CandleSticks, Line and Volume charts
StockChartX1.PriceStyle = psStandard
' Stop annoying series length offset errors
StockChartX1.IgnoreSeriesLengthErrors = True
' Let StockChart know what the parent table will be called
StockChartX1.Symbol = sSymbol
```

In this code sSymbol is a string variable that holds the acronym Symbol of the Security Name (for example CBA for Commonwealth Bank of Australia).

So now we have the table name for the flat file (the Symbol) in the object, but we have to load the field headings and tell the object where (that is, in which panel) these headings and associated data series will be displayed.

## ***Adding a Series***

Before we load historical trade data into StockChartX, we need to create and/or specify a panel (to tell the object where this data will be displayed, and then add a “series” name associated with the specified panel, and give the “series” a name so that StockChartX1 can relate to a price style – so that the price style can display the prices in much the same way that a font specification specifies how text will be seen in a word processor.

The usual way to do this is to use the security Symbol (nominally a three-letter to six-letter code), followed by a dot and the sub-parameter to complete the specification. We then need to tie this to the price style type, and then the (long) panel number.

If the string Symbol was for example CBA (Commonwealth Bank of Australia), and the panel number (p) was say 0 (the first panel), then the hard code could look like:

```
' Add a panel to the blank (black) screen
p = StockChartX1.AddChartPanel()
' Add four fields unto the StockChartX object
StockChartX1.AddSeries "CBA.open", stCandleChart, 0
StockChartX1.AddSeries "CBA.high", stCandleChart, 0
StockChartX1.AddSeries "CBA.low", stCandleChart, 0
StockChartX1.AddSeries "CBA.close", stCandleChart, 0
```

So the first panel has been added and now four series headings have been added to make the necessary data for the Candle chart, that will go into panel “p” (which is a long integer). You can now see how this coding would look to StockChartX, but the flexibility has been removed because it is specifically looking to load the CBA securities, and this is why the software structure below is recommended.

```
' Add a panel to the blank (black) screen
p = StockChartX1.AddChartPanel()

' Add the Prices to the StockChartX as a Candle Chart
StockChartX1.AddSeries sSymbol & ".open", stCandleChart, p
StockChartX1.AddSeries sSymbol & ".high", stCandleChart, p
StockChartX1.AddSeries sSymbol & ".low", stCandleChart, p
StockChartX1.AddSeries sSymbol & ".close", stCandleChart, p

' Add the Volume chart in a new panel
p = StockChartX1.AddChartPanel()

' Add the Volume to StockChartX as a Bar (Volume) Chart
StockChartX1.AddSeries sSymbol & ".Volume", stVolumeChart, p
```

You can see here that the Volume series heading has been entered in another Panel, separate from the Price series heading, and the Volume will display in its own Panel (below the Price panel), but the style type for the graph is a bar chart for use with Volumes.

## ***Managing Date and Time Issues***

Loading historical data would be rather straightforward if everybody used the same parameters. The problem here is that time / date is registered with slightly different calendar parameters.

With EOD data this data comes with the date on it in terms of a Gregorian calendar, that is, the date assumes that the data is/was recorded on that date and that it is after the close of business (because it is end of day data).

### **Working with Julian Time**

StockChartX1 uses the Julian calendar, because most financial organisations use the Julian calendar for their time and date references. The Julian Time date changes date at midday – not the previous midnight as per Gregorian Time; so there has to be a translation to bring the Julian Date in line with the Gregorian Date. By adding 12 hours to the Gregorian Date (as we see it in Years, Months, Days, Hours etc.) this artificially shifts the Gregorian Date forward by half a day and this offset aligns the two date systems.

In StockChartX, this object utilises a Double Precision number to express the date. The integer part is the day and the fractional part is the proportion of the day, so you would expect the date to change at midnight.

StockChartX contains a very nice little “ToJulian” function that converts String and Time offsets into a Julian Date so that virtually any data source that includes dates and or time can be converted into Julian time, and this time can then be added with the data to position the data in the right calendar slots. Julian calendar format is a Double Precision number, and StockChartXs ToJulian function translates a string specifying a date and time into Julian time. An example in changing a string (YYYYMMDD) into Julian time can be done as follows:

```
Dim Ddate as Double
```

```
JDate =StockchartX1.ToJulian(Left$(YYYYMMDD, 4), Mid$(YYYYMMDD, 5, 2),  
Right$(YYYYMMDD, 2), 0, 0, 0)
```

Note that near the end of this code there are three zeros for Hours, Minutes and Seconds which is used for real time capture (and the graphing is different for real time as compared to end of day, so it is necessary to stipulate what type of timeframe your data is working in.

If the timeframe is end of day (EOD) then the following code is necessary – before you load data through StockChartX1 in Julian form! With EOD data this data is evaluated at / after the close of the market and that is effectively about 4.30 pm, so to get things right, it is necessary to add another 4 hours and 30 minutes to make the date on the graph correct. So the total hours added is 16 in this case.

```
Dim Ddate as Double
```

```
JDate =StockchartX1.ToJulian( _  
    Left$(YYYYMMDD, 4), _  
    Mid$(YYYYMMDD, 5, 2), _  
    Right$(YYYYMMDD, 2), 16, 30, 0)
```

The above code shows the addition of 16 and 30 in the “Hours, Minutes” positions so this makes the visualised data move forward by one day (as the “Hours” is 12 or more), and the “Minutes” is 30, making the extra half hour.

In keeping with EOD data, we do not want Real Time X labelling, so we need to set the Boolean qualifier for Real Time X Labels as false, and this is the code to use:

```
StockChartX1.RealTimeXLabels = False
```

If the data is to be real time then the right hand part of this equation is True.

If you are working with EOD MetaStock data and are using the MetaLib DLL to pull the data out of EOD historical files, then here is an example of the MetaStock EOD Date being string manipulated and then offset (delayed) by 16 hours and 30 minutes (4.30 pm) so that the display will not be a day (date) in front of itself.

```
' This is for EOD values
' Time moved forward by 12 hours to get the date right,
  and another 4 hours (16) for 4 pm and 30 minutes 4:30 pm
  JDate = StockChartX1.ToJulianDate(Left(Reader.iSeDate, 4), _
    Mid(Reader.iSeDate, 5, 2), Right(Reader.iSeDate, 2), 16, 30, 0)
```

If the Financial world does indeed use the Julian time calendar, then live trade data should be loaded directly into StockChartX, and this should show up in the right time-slots.

### Working with Real Time

For Real Time data loading it may be necessary to add 12 hours to the time to bring StockChartX in line with midnight – and use 24-hour time. The issue is that most financial establishments use Julian calendar, and because of that the hours, minutes and seconds could be read in directly from live MetaStock data, as per the example below:

```
If StockChartX1.RealTimeXLabels Then
  ' Process the real time function to seconds etc.
  JDate = StockChartX1.ToJulianDate(Left(Reader.iSeDate, 4), _
    Mid(Reader.iSeDate, 3, 2), Right(Reader.iSeDate, 2), _
    Left(Reader.iSeTime, 2), Mid(Reader.iSeTime, 2, 2), _
    Right(Reader.iSeTime, 2))
```

This coding should now explain what is needed for real time data recording.

### Loading Price Data into a Series

When data is loaded into StockChartX1, the data is loaded as a Record – that is all the relevant data for one period of (Julian) time is loaded, and all this data forming a record is given an index (or reference) number. This index number is extensively used for presentation processes, and it is referred to as a “Record Number” starting at the long integer 1 through to the last record number. Remember that StockChartX1 uses the Julian date/time as its primary reference and the record numbers follow after the data is loaded.

If we were putting in data for a Candle (or OHLC) chart, then we would have to append values for Open, High, Low and Close, and here is an example for Open:

```
StockChartX1.AppendValue "CBA.open", JDate, dOpen
```

The string in literals spells out the table and column, the JDate spells out the time in Julian format in Double Precision, and the Double Precision open price spells out the value. The same would have to be repeated for every price.

Note that the Panel is not addressed as the string in literals has named the table (before the dot), and the panel has already been associated to the table series by the AddSeries command described above.

If we now want to add the volume for this same record it is simply a matter of adding the line to say so, like this example:

```
StockChartX1.AppendValue "CBA.Volume", JDate, dVolume
```

It should be fairly obvious that this data should be loaded with a For-Next loop or Do-Until loop or a Do-While loop depending on your preference. The programming below shows how the values could be loaded as part of a programming loop:

```
'Load the new Data Series in to StockChart
StockChartX1.AppendValue sSymbol & ".open", JDate, Reader.dSeOpen
StockChartX1.AppendValue sSymbol & ".high", JDate, Reader.dSeHigh
StockChartX1.AppendValue sSymbol & ".low", JDate, Reader.dSeLow
StockChartX1.AppendValue sSymbol & ".close", JDate, Reader.dSeClose
```

Note that every point on the total chart needs to be plotted into StockChartX, and that means that in this case for the open, high, low, close prices; these (plus the volume) need to be loaded in a loop. Fortunately, StockChartX is a very fast object and this is almost instant.

### Using ML Reader with MetaStock Data

To get started, it is assumed that the trade data is End Of Day (EOD) data and that it is in MetaStock form, and that the ML Reader object has also been loaded to read the data from the MetaStock-based trade data files. This assumption is fairly strong because most technical data is saved in MetaStock format, and ML Reader is a very simple object to use.

This next chunk of code calls the Reader object, which reads historical EOD data from the historical MetaStock database into the StockChartX object database:

```
Do Until (Reader.iSeRecordsLeft = 0)
  ' Insert Data values into StockChartX
  ' Read in the Open High Low Close Volume values,
  ' starting from Day 1 until now
  Reader.ReadDay

  If StockChartX1.RealTimeXLabels Then
    ' Process the real time function to seconds etc.
    JDate = StockChartX1.ToJulianDate(Left(Reader.iSeDate, 4), _
    Mid(Reader.iSeDate, 3, 2), Right(Reader.iSeDate, 2), _
    Left(Reader.iSeTime, 2), Mid(Reader.iSeTime, 2, 2), _
    Right(Reader.iSeTime, 2))
  Else
    ' This is for EOD values
    ' Time moved forward by 12 hours to get the date right,
    ' and another 4 hours (16) for 4 pm and 15 minutes 4:15 pm
    JDate = StockChartX1.ToJulianDate(Left(Reader.iSeDate, 4), _
    Mid(Reader.iSeDate, 5, 2), Right(Reader.iSeDate, 2), 16, 15, 0)
  End If

  If bFirstLoop Then
```

```

    bFirstLoop = False
    JDateOld = JDate
End If

' Load the new Data Series in to StockChart
StockChartX1.AppendValue sSymbol & ".open", JDate, Reader.dSeOpen
StockChartX1.AppendValue sSymbol & ".high", JDate, Reader.dSeHigh
StockChartX1.AppendValue sSymbol & ".low", JDate, Reader.dSeLow
StockChartX1.AppendValue sSymbol & ".close", JDate, Reader.dSeClose

' Load in the Volume data
If Reader.dSeVolume > 0 Then

    StockChartX1.AppendValue sSymbol & ".Volume", _
        JDate, (Reader.dSeVolume) ' / 1000000

    ' StockChartX1.AppendValue sSymbol & ".Volume", _
    ' JDate, 10 * Log(Reader.dSeVolume) ' / 1000000
End If

JDateOld = JDate
dOpenOld = Reader.dSeOpen
dHighOld = Reader.dSeHigh
dLowOld = Reader.dSeLow
dCloseOld = Reader.dSeClose

Loop
    StockChartX1.Update

Chart.lRecords = Reader.iSeRecords

' Hold this number of record data for span, scroll and zoom commands
' Close the Reader interface
Reader.CloseSecurity
Reader.CloseDirectory
Reader.DestroyArrays

```

It is important to close the Reader objects' Security table, Directory and destroy the arrays in the Reader before moving on because if this is not done, then the Reader leaves the MetaStock filing system open and the reading will 'lock-up' and not proceed past where the previously open files have not been closed.