Chapter 6
Creating Charts and Graphs

It has been said that a picture is worth a thousand words. This is especially true when analyzing trends in financial applications. Viewing the data in a graphical format is usually more meaningful than merely reviewing a bunch of numbers in a spreadsheet. In this chapter, we will explore several mechanisms by which we can generate graphs to be displayed on forms or printed in reports. (Note: Creating graphs for display in web pages can be accomplished using the Office Web Components. This is covered in Chapter 16: VFP on the Web).

Graphing terminology

When we first began working with graphs, we were quite confused by all the terms used to refer to the components of the chart object. The worst thing was that we were unable to find any definition for these terms in any of the documentation. Take, for example, the following excerpt from the MSGraph help file entry on the series object:

> *Using the Series Object*
>
> *Use* ***SeriesCollection****(index), where index is the series' index number or name, to return a single* ***Series*** *object. The following example sets the color of the interior for series one in the chart.*
>
> *myChart.SeriesCollection(1).Interior.Color = RGB(255, 0, 0)*

Clearly, this is less than helpful if you don't know what a series is. So let's begin with defining a few basic terms. A chart series is a single set of data on the graph. For example, if we create a chart from this data:

Graphs.ppt - Datasheet

		A	B	C	D	E
		1st Qtr	2nd Qtr	3rd Qtr	4th Qtr	
1	East	20.4	27.4	90	20.4	
2	West	30.6	38.6	34.6	31.6	
3	North	45.9	46.9	45	43.9	
4						
5						

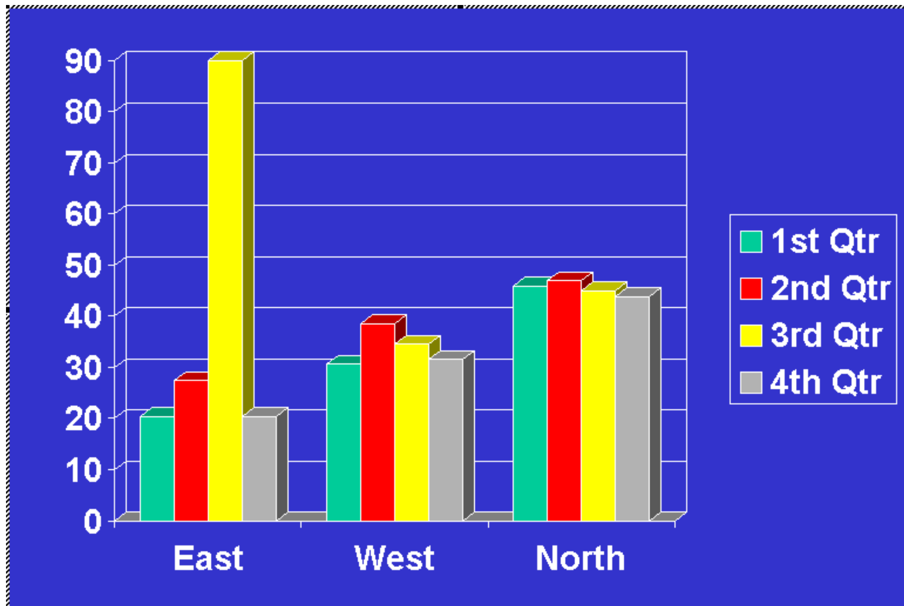
##F06001.TIF

Figure 6.1:Data used to generate a graph

each column in the table, excluding the first, would create one series object in the chart's series collection. At least, this is the way it works most of the time. It depends on whether or

not the graphing engine plots the series in rows or columns. The default for both the MSChart control and Excel is to plot the series in columns. However, the default for MSGraph is to plot the series in rows! Fortunately, the way in which the series are plotted is configurable and can be controlled programmatically.

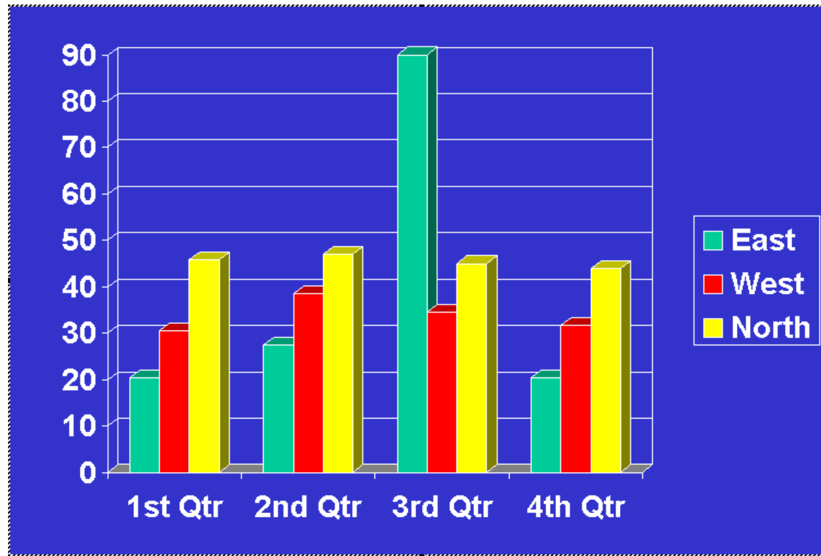
The way in which a series is represented depends upon the chart type. Figure 6.2 shows the four series that are created by the sample data above when the series are plotted in columns. The data in each series object is represented in this chart type by columns of different colors.



##F06002.TIF

Figure 6.2: 3-D clustered column graph containing 4 series objects (series in columns)

On the other hand, this is what the same chart looks like in Figure 6.3 when the Series are plotted from the data in the rows:



##F06006.TIF

Figure 6.3: 3-D clustered column graph containing 3 series objects (series in rows)

Most chart objects contain an axis collection. Two-dimensional charts have an x-axis (horizontal) and a y-axis (vertical). Three-dimensional charts add a z-axis (for depth). These axes are also referred to as:

- *Category axis* When the series data is plotted in columns, this identifies each row in the data that is used to generate the chart. This is usually, but not necessarily, synonymous with the x-axis. In Figure 6.2, the category axis displays the names of the regions. In Figure 6.3, where the series data is plotted in rows, the category axis displays the quarters.

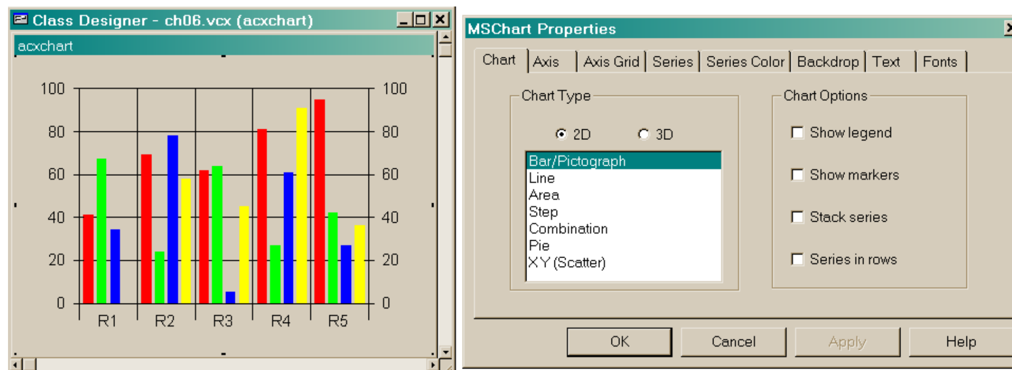
- *Value Axis* Identifies the range of values that will be displayed in the chart. This is usually, but not necessarily, synonymous with the y-axis. When defining the scale it is important to ensure that the maximum and minimum values of any series are encompassed by the value axis. In Figure 6.2 the values range between 0 and 90.

- *Series Axis* In three-dimensional charts each series is allocated its own set of spatial co-ordinates. This is usually, but not necessarily, synonymous with the z-axis. When the series data is plotted in columns, the labels along the series axis correspond to the column headings in the original data. When the series data is plotted in rows, this corresponds to the contents of the first column in the data used to generate the graph. The series axis labels and the legend entries display the same text.

Axes have grid lines and tick lines. The grid lines for the value axis in Figure 6.2 are the horizontal lines at each interval of 10. The lines that separate the labels on the category axis are the tick lines. These labels are also known as tick labels.

How do I create a graph using MSChart?*(Example: MsChartDemo.scx and ch06.vcx::acxChart)*

The MSChart control is a good starting point for working with graphs because it is a visual control. You can drop it on a form and see how changing its properties affect the appearance of the graph (Figure 6.4). Unfortunately, Microsoft stopped supporting this control on July 1, 1999 because, being single-threaded, it is incompatible with versions of Microsoft Internet Explorer later than Version 4.0. However, it still ships with Visual FoxPro and, if all you want to do is display a simple graph in a Visual FoxPro form, it is still a good solution.



##F06003.TIF

Figure 6.4:MSChart Control properties

The MSChart control is associated with a *DataGrid* that is used to create the necessary series objects. So in order to get MSChart to display a graph, we first have to populate the *DataGrid*, ensuring that we get things in the correct locations. Since the Chart control is a data bound control, we could create an ADO Recordset that contains the data to display, and use it as the Chart control's *DataSource*. When we use a recordset (and only when we use a recordset) the first field is assumed to be the label for the category axis when it holds character data. Otherwise the first column is treated no differently than any other and is used to define a series object. So, if the labels for your category axis represent numeric values, you must format them as character strings when using an ADO Recordset with the chart control.

Unfortunately, we cannot bind the Chart control directly to a Visual FoxPro cursor. So, unless we want to create an ADO Recordset from the cursor, we must iterate through the records in our cursor and populate the control's *DataGrid* directly like this:

```
LOCAL lnCol, lnRow
WITH THISFORM.oChart
  *** Set the number of fields in the cursor
  .ColumnCount = FCOUNT( 'csrResults' ) - 1
```

```
  *** Set the number of rows to the number of records in the cursor
  .RowCount = RECCOUNT( 'csrResults' )

  *** Populate the DataGrid Object.
  SELECT csrResults
  SCAN
    .Row = RECNO( 'csrResults' )
    FOR lnCol = 1 TO .ColumnCount
      .Column = lnCol
      *** Since the first column is used for the category labels
      *** We must increment our counter
      .Data = EVALUATE( FIELD( lnCol + 1 ) )
    ENDFOR
  ENDSCAN
ENDWITH
```

Note that when we manually populate the Chart's *DataGrid* like this, the labels for the category axis do not automatically come from the first column of the data. In fact, used this way, the *DataGrid* can only contain the actual values that will be used to generate series objects. Labels are added by explicitly setting the properties for them on both the category and value axes.

Having populated the grid, we can set the properties that control the output. There are an awful lot of these, but the most important ones for explaining what a graph shows are:

- *RowLabel*: The labels collection for the category axis
- *ColumnLabel*: The labels collection for the value axis
- *AxisTitle.Text*: The title for the axis title object
- *AxisTitle.VtFont.Size*: The font size for the axis title object

The sample form creates three different graphs on the fly and displays them using the MSChart control. To make the graph generation process extensible and maintainable, we store the graph definitions in a free table called QUERIES.DBF with this structure:

Table 6.1:Structure of metadata used to store graph definitions

Field Name	Type	Length	Description
cQueryName	C	20	Keyword used to look up the record in Queries.dbf
cQueryDesc	C	50	Query description for use in end user displays
cPopupForm	C	80	Name of popup form used to obtain values for the query's ad hoc where clause if one is specified
nChartType	I		Type of chart to generate (e.g. 3-D Bar, 2-D Line) defined by one of the chart type constants in *mschrt20.h*
nGraphType	I		Serves the same purpose as nChartType when used to generate graphs using MsGraph (We need both because the constants have different meanings in MSGraph and MSChart)
mQuery	M		The query used to obtain the data that will be used to generate the chart. This may include expressions like "WHERE Customer.cust_id = '<<pcParm1>>'" to specify an ad hoc WHERE clause because the TEXTMERGE() function will be used before the query is run.
cMethod	C	80	The name of a form method to run after the query is run to massage

			the result cursor
cTitleX	C	50	Title for the X-axis
cTitleY	C	50	Title for the Y-axis
cTitleZ	C	50	Title for the Z-axis

The form has seven custom methods that use the data in QUERIES.DBF to gather any required parameters from the user and generate the graph.

Table 6.2:MSChartDemo.scx custom methods

Method name	Description
MakeGraph	Called by the *onClick()* method of the 'Create Graph' command button, this is the control method that generates the graph.
DoQuery	Called by the form's *MakeGraph()* method, uses the passed parameter object to run the query contained in the mQuery field of the current record in *Queries.dbf*. It always places the query results in a cursor named *csrResults* so that we can write generic code in the form to handle the results of different queries.
GetQueryParms	Called by the form's *MakeGraph()* method. This method instantiates the form specified in the cPopUpForm field of the current record in *Queries.dbf*. The popup form returns a parameter object which is passed back to the *MakeGraph()* method.
MakeMonthColumns	Called by *MakeGraph()* when it is specified in the cMethod field of *Queries.dbf*. It takes the contents of *csrResults* , which is a "vertical" structure with a single record for each month, and converts it into a "horizontal" structure with one field for each month in a single record.
MakeYearColumns	Called by *MakeGraph()* if it is specified in the cMethod field of Queries.dbf. It takes the contents of *csrResults*, which is a "vertical" structure with a single record for each year, and converts it into a "horizontal" structure with one field for each year in a single record. The number of fields in the new structure depends upon the range of distinct years contained in the original cursor.
PopulateDataGrid	Called by *MakeGraph()* to populate the graph's *DataGrid* object from the information in *csrResults*.
SetAxisTitles	Uses the information in the title fields in *Queries.dbf* to set the axis titles. Also sets the fonts for the titles and labels.

As you can see from Table 6.2, the form's custom *MakeGraph()* method controls the whole process. It passes any required parameters to the form's custom *DoQuery()* method. *DoQuery()*, as its name implies, runs the query from QUERIES.DBF to generate the cursor *csrResults* thatholds the data used to generate the graph. Next, when a method name is specified in *Queries.cMethod*, *MakeGraph()* first checks to make sure that the method exists and then calls it. The *csrResults* cursor, generated by the *DoQuery()* method is used by *PopulateDataGrid()* to pass data to the chart like this:

```
LOCAL lnCol, lnRow
WITH THISFORM.oChart

  *** Set the chart type
  .ChartType = Thisform.cboChartType.Value

  *** Set the number of fields in the cursor
  .ColumnCount = FCOUNT( 'csrResults' ) - 1

  *** Set the number of rows to the number of records in the cursor
```

```
  .RowCount = RECCOUNT( 'csrResults' )

  *** Populate the DataGrid Object.
  SELECT csrResults
  SCAN
    .Row = RECNO( 'csrResults' )

    *** Set up the label for the category axis
    .RowLabel = EVALUATE( FIELD[ 1 ] )

    *** Populate the data grid with the numeric data
    FOR lnCol = 1 TO .ColumnCount
      .Column = lnCol
      .Data = EVALUATE( FIELD( lnCol + 1 ) )
    ENDFOR
  ENDSCAN
  FOR lnCol = 1 TO .ColumnCount
    .Row = 1
    .Column = lnCol
    .ColumnLabel = ALLTRIM( FIELD( lnCol + 1 ) )
  ENDFOR
ENDWITH
```

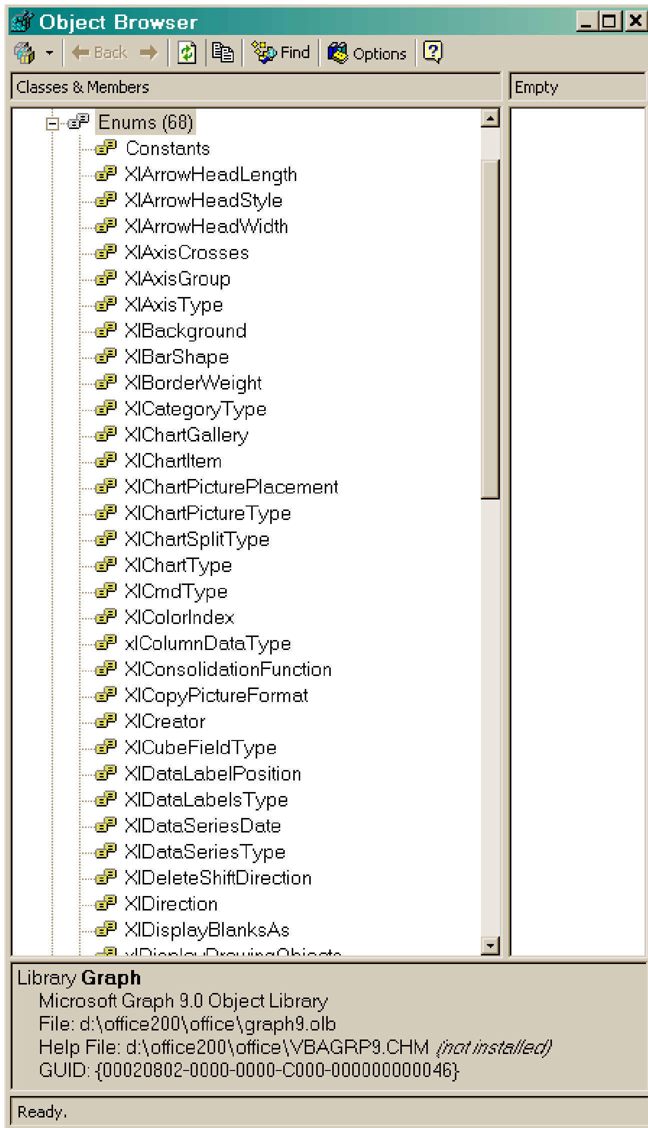
The act of populating the *DataGrid* forces the chart to display on the form, however, at this point all it has it the raw data and axis labels. The final steps in the *MakeGraph()* process are to call *SetAxisTitles()* and then tidy up the display by setting the chart's *Projection*, *Stacking* and *BarGap* properties to values which are more suitable than the defaults which MSChart supplies when the chart is re-drawn.

Note that the *PopulateDataGrid()* method manipulates the *Data* property of the chart object directly. However, if you open MSCHART20.LIB in the object browser you will not be able to find this property. Apparently it is not exposed by the type library. However, it is documented in the help file (MSCHRT98.CHM) and certainly appears to be present and available. It is used to get, or set, a value at the current data point in the data grid of a chart. A data point is made current by setting the chart's *Row* and *Column* properties to its co-ordinates. By the way, these properties do not appear in the object browser either, even though they too are listed in the help file.

As you can see, getting a graph into a form is actually pretty easy with MSChart. However, including an MSChart graph in a printed report is not. MSChart does have an *EditCopy()* method that copies the chart object to the clipboard in metafile format, but there is no easy way to transfer the metafile from clipboard to disk without using additional software. Nor can you insert the Chart object into a *General* field, so it cannot be included in a printed report that way either. So if printing is a requirement, you need to use something other than MSChart.

How do I create a graph using MSGraph? *(Example: MsGraphDemo.scx)*

MsGraph is actually a cut-down version of the Microsoft Excel graphing engine. You can see this if you open GRAPH9.OLB in the object browser and expand the *enums* node (Figure 6.5).

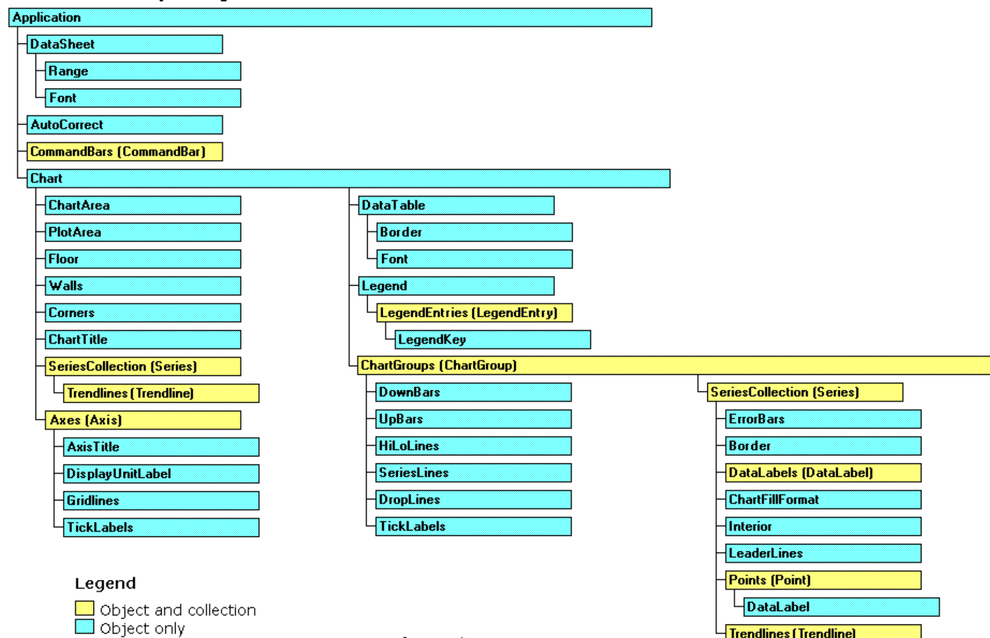


##F06004.TIF

Figure 6.5:MSGraph constants in the class browser

Notice that all the constant names begin with 'Xl'? There is a very good reason for this, they are the same names and values that are used by Excel's graphing engine. So, if you start out using MSGraph to create your graphs and later decide to move to Excel automation, you should find that most of the code to manipulate the graph will run with no modification.

Having a much simpler object model than Excel (Figure 6.6), MSGraph is lighter and quicker to instantiate and therefore the results appear more quickly too.



##F06005.TIF

Figure 6.6:MSGraph object model

MSGraph gives you much more control over the graph's appearance than MSChart does. It is also possible to include graphs in printed reports if you use MSGraph to generate them because they *can* be added to, and printed directly from, a *General* field in a table or cursor.

##Note Icon #1

One unpleasant surprise that we encountered when working with MSChart and MSGraph was the lack of consistency between the two. Not only did the properties, methods and events have different names, even the constants had different meanings! For example, a chart type of '1' in MSChart produces a 2-dimensional bar graph. In MSGraph, this is an area graph.

The easiest way that we have found to manipulate MSGraph is to store a "template" graph in a *General* field of a table. Then, whenever we need to create a particular graph, we drop an OleBound Control on a form and set its *ControlSource* to the *General* field in a cursor created from that table. This has several benefits:

- Avoids contention when several users try to create a graph at the same time since each user has his own cursor

- Does not require multiple graphs to be stored. After all, graphs are generally used to depict current trends and, as such, are usually generated 'on the fly' so it makes no sense to store them permanently in a table or on disk
- Using an OleBound Control gives us direct access to the graph through its properties. This means we can manipulate the graph programmatically while the form is invisible and then display or print the final result.

The sample form uses this technique. Notice that we are using the same data-driven methodology that we used with the MSChart control. Thus, the custom *MakeGraph()* method in the sample form controls the graph generation process and calls the following supporting methods:

- *GetQueryParms* — Pulls up the popup form to gather any values required for the query's ad hoc where clause if a pop up form is specified in the cPopupForm field of QUERIES.DBF
- *DoQuery* — Creates a cursor named *csrResults* by running the query specified in QUERIES.DBF
- *UpdateGraphData* — Constructs the proper format string to use with **APPEND GENERAL DATA** and issues the command
- *FormatGraph* — Sets various graph properties such as axis titles, tick label fonts, and so on.

Two new custom methods, *UpdateGraphData()* and *FormatGraph()* use the cursor to render the appropriate graph. The only differences from the MSChart sample are in the details of the code that is used to generate and format the graph object.

The form's custom *UpdateGraphData()* method uses the native Visual FoxPro **APPEND GENERAL** command with the **DATA** clause to update the graph in the *General* field of a cursor named *csrGraph*. This cursor is created from the table *vfpgraph.dbf* in the form's *Load()* method. (Remember, the *vfpgraph.dbf* table has only one record which stores the template graph and which is never updated). In order to use this form of **APPEND GENERAL**, the data must be in "standard clipboard" format which means that the fields are separated by **TAB** characters, and the records are separated by carriage returns. The first part of the method deals with converting the data from *csrResults* into the correct format.

```
LOCAL lcGraphData, lnFld, lnFieldCount

*** Make the oleBoundControl invisible
*** and unbind it so we can update the general field
Thisform.LockScreen = .T.
Thisform.oGraph.ControlSource = ''

*** Now build the string we need to update the graph
*** in the general field
lcGraphData = ""
SELECT csrResults
lnFieldCount = FCOUNT()
```

```
*** Build tab-delimited string of field names:
FOR lnFld = 1 TO lnFieldCount
  lcGraphData = lcGraphData + FIELD( lnFld ) ;
    + IIF( lnFld < lnFieldCount, CHR( 9 ), CHR( 13 ) + CHR( 10 ) )
ENDFOR

*** Concatenate the data, converting numeric fields to character:
SCAN
  FOR lnFld = 1 TO lnFieldCount
    lcGraphData = lcGraphData + TRANSFORM( EVALUATE( FIELD( lnFld ) ) ) + ;
      + IIF( lnFld < lnFieldCount, CHR( 9 ), CHR( 13 )  + CHR( 10 ) )
  ENDFOR
ENDSCAN

GO TOP IN csrResults

*** OK, ready to update the graph
SELECT csrGraph
APPEND GENERAL oleGraph CLASS "MsGraph.Chart" DATA lcGraphData
```

Having updated the *General* field we can bind the control on the form directly to the cursor and set the following properties:

- *ChartType*: Determines the type of graph. Values are defined in *graph9.h*
- *Application.PlotBy*: Determines whether series objects are generated from rows, or columns in the data.

```
WITH Thisform.oGraph
  *** Reset the controlSource of the OleBound control
  .ControlSource = "csrGRaph.oleGraph"

  *** Set the chart type
  .object.ChartType = Thisform.cboGraphType.Value

  *** Set the data to graph the columns as the series
  *** Unless, of course, this is a pie chart
  IF NOT INLIST( .ChartType, xl3DPie, xlPie, xlPieOfPie, xlPieExploded, ;
      xl3DPieExploded, xlBarOfPie )
    .Object.Application.PlotBy = xlColumns
  ELSE
    .Object.Application.PlotBy = xlRows
  ENDIF
ENDWITH

Thisform.LockScreen = .F.
```

It is evident from this code listing that we ran into a couple of problems when creating the sample form. First, we discovered that the `APPEND GENERAL` command refused to update the graph in the general field while it was bound to the OleBound Control. We finally had to unbind the control before issuing the command and re-bind it afterward. Second, we had to explicitly tell MSGraph to use the columns as the data series for all charts except pie charts, overriding the default behavior which is data series in rows and which can produce some very odd-looking graphs.

That is all that must be done to generate and display the graph. However, we found that the default values produced ugly graphs and so we needed to set some additional properties to improve the appearance. The properties and methods available for MSGraph are, to say the least, comprehensive. The full list is included in VBAGRP9.CHM, the help file for MSGraph. However, the actual documentation is rather sparse, so, once you are certain that the item you are interested in actually exists in the MSGraph object model, we suggest that you look it up in the Excel documentation - which is slightly better. Remember, MSGraph is just a cut-down version of Excel's graphing engine.

While it is beyond the scope of this chapter to show you how to manipulate all of these properties, the custom *FormatGraph()* method shows how manipulating a few of the graph's properties changes its appearance.

The first thing that we want to do is to set the axis titles and fonts. However, not all chart types have an axes collection (most notably Pie charts). The chart object exposes a *HasAxis()* method which, despite being referred to in the documentation as a *Property*, accepts a constant that identifies the axis type and returns a logical value. You would be forgiven for thinking that we could use this to tell us whether a given axis exists. However, it turns out that if the graph does not have an *axes* collection, trying to access this "*property*" simply causes an OLE error. So we have no alternative but to check the graph type explicitly:

```
IF NOT INLIST( .ChartType, xl3DPie, xlPie, xlPieOfPie, ;
               xlPieExploded, xl3DPieExploded, xlBarOfPie )
```

and only if the chart has axes do we then proceed to configure them, by setting the following properties for each axis object in the Axes collection:

- TickLabels.Font.Size
- HasTitle
- AxistTitle.Text
- AxisTitle.Font.Size

```
WITH .Axes( xlCategory )
  .TickLabels.Font.Size = 8
  lcTitleText = ALLTRIM( Queries.cTitleX )
  IF NOT EMPTY( lcTitleText )
    .HasTitle = .T.
    .AxisTitle.Text = lcTitleText
    .AxisTitle.Font.Size = 10
  ELSE
    .HasTitle = .F.
  ENDIF
ENDWITH
```

For the chart types that don't have an axes collection, we only need to set the following properties on the Chart object:

- HasTitle
- ChartTitle.Text

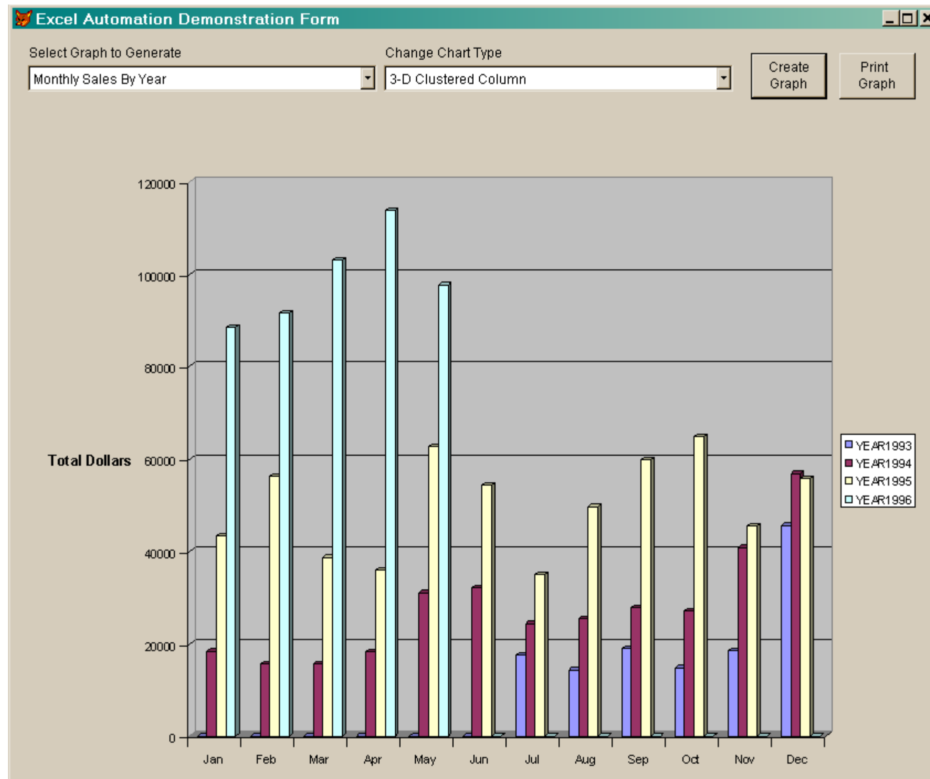
- ChartTitle.Font.Size

but we also need to call an additional method, *ApplyDataLabels()*, to assign labels to the pie chart segments.

One word of caution. Even though a given property or method is defined as being part of the object model, not all properties and methods are always available. As we found with the *HasAxis()* method, it is imperative to ensure that a specific instance of the graph actually has the required property or method before trying to access it. If, for any reason, it is not available, MSGraph hands you back a nasty OLE error.

How do I create a graph using Excel automation?*(Example: ExcelAutomation.scx)*

Producing graphs represents only a tiny fraction of what you can do when you harness the power of Excel in your Visual FoxPro applications. While VFP is an excellent tool for manipulating data, it is definitely not the best tool when it comes to dealing with complex mathematical formulae. An entire book could be devoted to the topic of Excel automation (in fact, several have) and a complete discussion of its capabilities is beyond the scope of this chapter. For specific examples using Visual FoxPro, see "*Microsoft Office Automation with Visual FoxPro*" by Tamar E. Granor and Della Martin (Hentzenwerke publishing, 2000 ISBN: 0-9655093-0-3).



##F06007.TIF

Figure 6.7:Excel automation sample form

The example form (Figure 6.7) uses the same data-driven methodology that we have used with MSGraph and MSChart. The difference is that instead of formatting our data and feeding it directly to the graphing tool, we now have to feed the data to Excel and then instruct it to create a graph using that data. To do this we added a custom *AutomateExcel()* method that first creates an instance of Excel, then opens a workbook and populates a range in the active worksheet with the data from our results cursor:

```
*** create an instance of excel.
loXl = CREATEOBJECT( 'Excel.Application' )

*** Now add a workbook so we can populate the active worksheet
*** with the data from the query results
loWB = loXl.Workbooks.Add()

*** Now fill in the data
WITH loWb.ActiveSheet

  *** Give it a name so we can reference it
  *** after we add a new sheet for the chart
  .Name = "ChartData"
```

```
  *** Make sure we have the field names in the first row of the work sheet
  *** we do not want the field name for the first column which is used to
  *** identify the categories
  FOR lnCol = 2 TO FCOUNT( 'csrResults' )

    *** Convert the field number into an Excel cell designation
    *** We can do this easily because 'A' has an ascii value of 65
    lcCell = CHR( lnCol + 64 ) + "1"

    *** Go ahead and set its value
    .Range( lcCell ).Value = ALLTRIM( FIELD( lnCol, 'csrResults' ) )
  ENDFOR
```

Populating the cells in the worksheet is a little trickier than populating the *DataGrid* of the MSChart control, or sending data to MSGraph, because the cells in an Excel spreadsheet are identified by an alphanumeric key. The columns are identified by the letters A through Z while the rows are identified by their row numbers. So, to access a particular cell, you must use a combination of the two. For example, to identify the cell in the first row of the first column of the spreadsheet, you identify it as Range('A1'). As you can see in the following code, it is easy enough to convert a column number to a letter because the letter 'A' has an ASCII value of 65. So all we need to do is add 64 to the field number in our cursor and apply the **CHR()** function to the result.

```
  *** Now just scan the cursor and populate the rest of the cells
  SELECT csrResults
  SCAN
    FOR lnCol = 1 TO FCOUNT( 'csrResults' )

      *** Get the cell in the worksheet that we need
      *** Since the first row has the column headings, we must
      *** start in the second row of the worksheet
      lcCell = CHR( lnCol + 64 ) + TRANSFORM( RECNO( 'csrResults' ) + 1 )

      *** Go ahead and set its value
      .Range( lcCell ).Value = EVALUATE( FIELD( lnCol, 'csrResults' ) )
    ENDFOR
  ENDSCAN
  GO TOP IN csrResults
ENDWITH
```

This code works, but there is one major problem with it. It is slow! Poking values into individual cells in a spreadsheet inside of a tight loop is not the most efficient way to get the job done. Fortunately, we can use the *DataToClip()* method of the Visual FoxPro application object to copy the data in our cursor to the clipboard. Then we can use the active worksheet's *Paste()* method to insert the data from the clipboard into the spreadsheet. Using the modified code listed below yields an improvement in performance of up to 60% depending on the volume of data being sent to Excel. Another benefit of using our modified version of the code to send data to Excel is that there is less of it. Less code means fewer bugs.

```
WITH loWb.ActiveSheet
  *** Give it a name so we can reference it
  *** after we add a new sheet for the chart
```

```
  .Name = "ChartData"

  *** Get the number of columns
  lnFldCount = FCOUNT( 'csrResults' )

  *** Add one because copying the data to the clipboard
  *** adds a row for the field names
  lnRecCnt = RECCOUNT( 'csrResults' ) + 1
  SELECT csrResults
  GO TOP

  *** Copy to clipboard with fields delimited by tabs
  _VFP.DataToClip( 'csrResults', RECCOUNT( 'csrResults' ), 3 )

  *** Get the range of the data in the worksheet
  lcCell = 'A1:' + CHR( 64 + lnFldCount ) + TRANSFORM( lnRecCnt )

  *** And paste it in
  .Paste( .Range( lcCell ) )

  *** But now we have to make sure that cell A1 is blank
  *** Otherwise the chart is not created correctly
  .Range( "A1" ).Value = ""
  GO TOP IN csrResults
ENDWITH
```

After we transfer the data from the cursor to the spreadsheet, we are ready to create the graph. This is done by adding an empty chart object to the workbook's charts collection and telling it to generate itself. The chart object's *SetSourceData()* method accepts a reference to the range that contains the data together with a numeric constant that specifies how to generate the series objects (i.e. from rows or columns). To ensure that the display is in the correct format, the *AutomateExcel()* method forces the chart object's *ChartType* property to the correct value:

```
loChart = loWB.Charts.Add()

*** Set the data to graph the columns as the series
*** Unless, of course, this is a pie chart
IF NOT INLIST( Thisform.cboGraphType.Value, xl3DPie, xlPie, ;
         xlPieOfPie, xlPieExploded, xl3DPieExploded, xlBarOfPie )
  lnPlotBy = xlColumns
ELSE
  lnPlotBy = xlRows
ENDIF

WITH loChart
  *** Generate the chart from the data in the worksheet
  .SetSourceData( loWB.Sheets( "ChartData" ).Range( lcCell ), lnPlotBy )

  *** Set the chart type
  .ChartType = Thisform.cboGraphType.Value
```

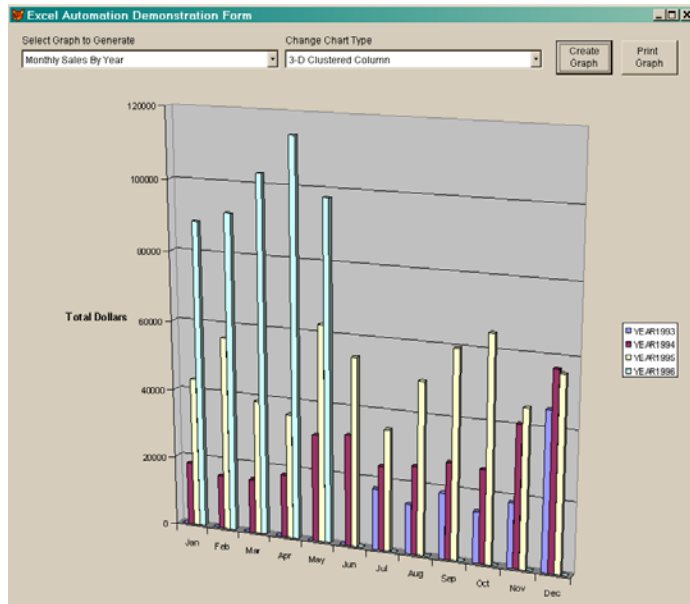
At this point we have a chart in an Excel spreadsheet, but what we really want is to display it in a Visual FoxPro form. The easiest way to do this is to use the chart's *SaveAs()* method to save it to a temporary file. We can then use the **APPEND GENERAL** command to suck

the chart into the *General* field of the cursor that was created in the *Load()* method of the demonstration form. Once the graph is safely in the *General* field, we can quit Excel, erase the temporary file, and bind the OleBound control on the form to the cursor's *General* field. The last part of the *AutomateExcel()* method does exactly that:

```
  *** Save to a temporary file
  lcFileName = SYS( 2015 ) + '.xls'
  loChart.SaveAs( FULLPATH( CURDIR() ) + lcFileName )
ENDWITH

*** and quit the application
loXl.Quit()

*** insert the graph into the general field in the cursor
SELECT csrGraph
APPEND GENERAL oleGraph FROM ( lcFileName ) CLASS "Excel.Chart"

*** and clean up
ERASE ( lcFileName )

WITH Thisform
  *** Reset the controlSource of the OleBound control
  .oGraph.ControlSource = "csrGraph.oleGraph"
  .LockScreen = .F.
ENDWITH
```

Now that the graph is bound to the OleBound control on the form, we can manipulate its appearance in much the same way that we did for MSGraph. As a matter of fact, the code the in form's custom *FormatGraph()* method is almost identical to the code in the previous example. Other than changing references to *Thisform.oGraph.Object* to *Thisform.oGraph.Object. ActiveChart*, all we had to change for our Excel automation sample was to add this line:

```
*** Now set the axes at right angles for 3-d bar, column, and line charts
IF INLIST( .ChartType, xl3DColumnClustered, xl3DColumnStacked, ;
            xl3DColumnStacked100, xl3DBarClustered, xl3DBarStacked, ;
            xl3DBarStacked100, xl3DLine )
  Thisform.oGraph.Object.ActiveChart.RightAngleAxes = .T.
ENDIF
```



##F06008.TIF

Figure 6.8:Default perspective of 3-D graph generated by Excel

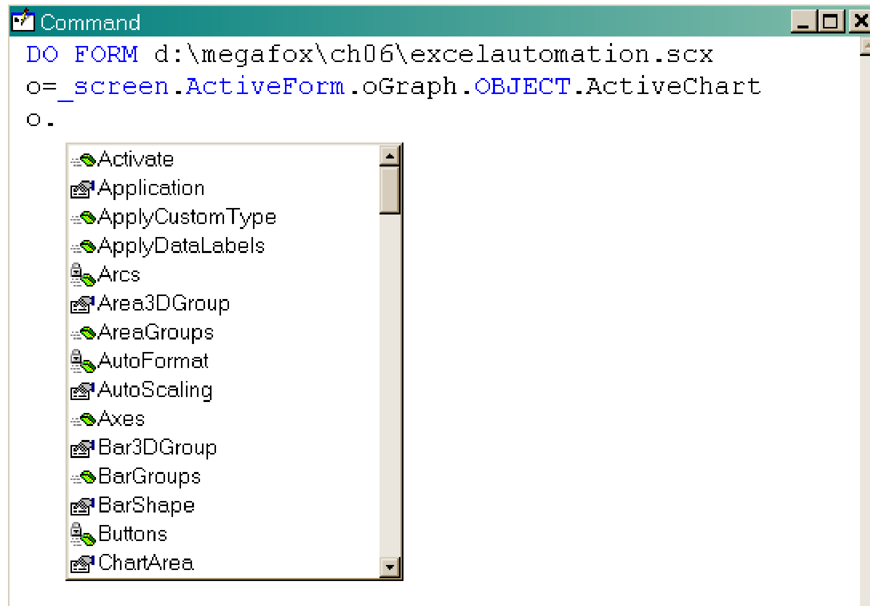
This is because MSGraph, by default, creates a pretty 3-dimensional graph with the axes at right angles to each other. Excel does not. Before we added this line of code, the graph looked like Figure 6.8. As you can see, it had an unappealing ragged appearance.

The graph object has a huge numbers of properties and methods that you can use to manipulate its appearance which are described in the Excel 2000 help file, *VbaXl9.chm*. In addition to the documentation, our sample form makes it easy for you to experiment with the effect of changing properties. All you have to do is run the form, click the '*Create Graph*' button, and type this in the command window:

```
o = _Screen.ActiveForm.oGraph.Object.ActiveChart
```

You can then call the methods of the charts object or change its properties and immediately see either an OLE error or a change in the appearance of the graph in the form.

Using Visual FoxPro 7 makes the discovery process even easier because of IntelliSense. Once you have a reference to the chart object, you can see a list of all the methods and properties that apply (Figure 6.9).



##F06009.TIF

Figure 6.9: Exploring the Excel object model from the VFP 7.0 command window

Conclusion

A picture is indeed worth a thousand words. Including graphs and charts in your applications, whether displayed in a form or printed in a report, adds a lot of pizzazz and a professional look and feel. In the past, adding this functionality was more than a little painful because of the lack of documentation and good working examples. We hope that this chapter has given you a better starting point than we had when we wrote it and that you can use the examples to create some really spectacular graphs of your own.