

# What to do when (column) size matters

William Vaughn

One of the features I found particularly useful in the first Visual Studio .NET Beta was the ability to automatically size a DataGrid control based on its contents using a single Boolean property setting. For some reason, this feature was dropped from the .NET Framework in V1.0 and 1.1. I expect that we'll see it reappear someday, but until then, this article discusses how to return this functionality to your own application.

In Visual Basic 6.0 we could set the size of a grid column by setting the column width using the TextWidth method to pass a value to the grid control's ColWidth property. In those days we could address the grid control's columns fairly easily. The following code illustrates a simple Visual Basic 6.0 routine to set the width of the MSHFlexGrid control columns based on the first data row's value. Note that the ordinary DataGrid control in Visual Basic 6.0 is managed differently so this technique does not apply to that control.

```
With Grid1
    .Col = 0
    For Each fld In rs.Fields
        .Text = fld.Value
        If .Row = 2 Then .ColWidth(.Col) = TextWidth(fld.Value) * 1.5
        If .Col < rs.Fields.Count - 1 Then .Col = .Col + 1
    Next fld
End With
```

*Figure 1: Visual Basic 6.0 ColWidth setting for MSHFlexGrid control.*

In this case, the TextWidth method returns the number of twips<sup>1</sup> in the specific text string—but in the context of the Form, not the DataGrid. This can lead to issues where the fonts used in the grid don't match what you're using in the Form.

When I wrote a sample application to convert from Visual Basic 6.0 to Visual Basic .NET, I included code like this to resize the grid columns—it didn't convert, but I didn't press the issue initially as there was a suitable substitute in the beta version of the .NET Framework. Apparently, the Microsoft doc team thought so too as they included the property in many of their grid examples. But when Visual Studio .NET Beta 2 came out the property disappeared and lots of my code was left in the lurch. I cobbled something together at the last minute to fill in the gap, but I was never really happy with the result—until now.

## Autosizing with .NET

The DataGrid control team fell back 10 yards and punted when they decided to include the DataGrid PreferredColumnWidth property that tells Windows to set the width of all of the grid columns to a specific number of pixels. This makes sense since the new unit of measurement for controls in a .NET Windows application is pixels—both the control Width and Height are specified in pixels. It's up to your code to determine how many pixels you need to display text in your control columns.

## Measuring Strings

So, our first task will be to come up with a way to determine the pixel width of a specific fixed-length string. Well, the problem is that the pixel width of a string can vary quite a bit. This is because the width (and height) of a particular grid cell (or the header text) is a function of the display font at execution time. You'll want a routine that adapts itself at runtime to varying font settings so you don't have to worry about your code working on one system but not another—just because someone selected "Large Fonts" in the video settings dialog.

## Padding the cells

The code in Figure 2 returns the display width in pixels of a string in the context of a specific Windows control. In this case I'm measuring the width of a string of "N" character "M"s to use as padding around the text in the headers and cells. This makes sure the data does not touch the grid cell boundary lines. This routine starts with a Graphics object derived from the target control (our DataGrid in this case) which exposes the MeasureString method which accepts (in this overload), the string to measure and the font to be used to measure it with. That

---

<sup>1</sup> Twip – the standard unit of measurement in a Windows form. The doc says (if you care) it's a screen-independent, absolute unit of measurement (such as an inch or a centimeter). A twip is a unit of length equal to 1/20 of a printer's point, and a printer's point is 1/72 of an inch. There are approximately 1440 twips to a logical inch or 567 twips to a logical centimeter (the length of a screen item measuring one inch or one centimeter when printed).

is, by passing the DataGrid Font property, the MeasureString function can determine exactly how the text is going to be rendered on the screen (with its current Windows settings), and return the “size” of the text (both the height and (more importantly) the width) in a SizeF object.

Remember, that most of the fonts used now-a-days are *not* fixed-pitch fonts like *Courier*, so you can't just calculate the width of single character and multiply that value times the length of the string. We're going to have to take into account the fact that two strings of the same length won't necessarily resolve to the same width.

To measure the width of my padding string, I created a SizeF and Graphics typed variables to hold the dimensions of the string of “M” characters. I used “M” as it's typically the widest of all characters used. Note that the SizeF Width property returns a Single datatype value so expect the Width to be returned as a floating-point number such as 42.02933113. The Graphics object is extracted from the DataGrid control using the CreateGraphics method.

```
Dim sngPadding as Single
Dim gr as Graphics
Dim sz As SizeF 'Stores an ordered pair of floating-point numbers, typically the width and
height of a rectangle.
gr = dgData.CreateGraphics
' Pad "n" M-width characters
sz = gr.MeasureString(New String("M", intPadding), dgData.Font)
sngPadding = sz.Width
```

Figure 2: Determining the size of a control region

The next part of the problem is the algorithm used to size the grid itself. I used the following criteria to code the autosizing routine.

- The header text must always be visible after resizing (not cut off because the column was too narrow).
- The data must always be visible after resizing. This means that the width of the column would be the longer of the header or text widths.
- The autosizing routine code had to be fast and tight so it would not noticeably interfere with the application UI.
- The routine would need to scan data values attached via the grid DataSource property. If no DataSource is present, the routine throws an exception.
- The data bound to the grid control might be hundreds (or thousands) of rows long so it would be impractical (and slow) to examine more than the currently visible rows.
- The routine is to accept a padding value to add a variable amount of space around the cell or header text to prevent the text from getting cut off by the grid lines. This can occur when the MeasureString function returns a width value which is slightly too small—which happens sometimes.

## Setting up the DataGrid TableStyles

Before we wade into the routine, there is another little issue that needs to be discussed. In Visual Basic 6.0 the MSHFlexGrid had far fewer properties and was a bit easier to program—up to a point. The DataGrid in the .NET Framework is a lot more complex but far more powerful. Instead of simply setting a ColWidth property on a column-by-column basis (as we did in Visual Basic 6.0) we're going to have to create a new DataGrid TableStyle object and add it to the DataGrid TableStyles collection. Once this object is in place we can set the TableStyle MappingName to permit named reference to this “style sheet” which tells Windows how to render the DataGrid. Using this approach also permits a much finer control over how the DataGrid is displayed and behaves.

## Application initialization

Figure 3 shows how the sample application is initialized. Here I create a simple parameter query that returns several rows from the Titles table. You can easily modify this code for your own database and query.

```
Dim cn As SqlConnection
Dim da As SqlDataAdapter
Dim ds As New DataSet
Dim ts As New DataGridTableStyle ' To expose DataGrid styles so we can alter them
Dim intDefaultPadding As Integer = 1
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
Try
    cn = New SqlConnection("Data Source=demoserver;integrated security=sspi;ini...")
    da = New SqlDataAdapter("SELECT Title, Year_Published, ISBN, PubID, " _
        & " Price FROM Titles WHERE Title LIKE @TitleWanted ", cn)
```

```
da.SelectCommand.Parameters.Add("@TitleWanted", txtTitleWanted.Text)
```

Next, I add the DataGrid TableStyle object so we can manage the column width.

```
ts.MappingName = "Titles" ' Address the TableGrid Style
DataGrid1.TableStyles.Add(ts) ' Attach the new style
btnQuery.PerformClick() ' show the default query
Catch ex As Exception
MsgBox(ex.ToString)
End Try
End Sub
```

Figure 3 – Initializing the AutoSize test routine.

## The AutoSize routine

The code shown in Figure 4 is the completed AutoSize subroutine. It's called after the query is executed. We start with defining exceptions thrown by the routine in case the DataGrid is not bound to a DataSet or DataTable. Sure, you can modify this code to work with a bound array, but there's no room to discuss that here.

```
Sub AutoSize(ByVal dgData As DataGrid, ByVal strMappingName As String, _
ByVal intPadding As Integer)
Dim ex1 As New Exception("DataGrid must be databound")
Dim ex2 As New Exception("DataSource must be DataSet or DataTable")
Dim dt As DataTable
Dim dr As DataRow
Dim dcCol As DataColumn
Dim gr As Graphics
Dim sngGridNameWidth, sngDataWidth As Single
Dim i As Integer, sngPadding As Single
If dgData.DataSource Is Nothing Then
Throw ex1
Exit Sub
End If
If TypeOf dgData.DataSource Is DataSet Then
ds = dgData.DataSource
dt = ds.Tables(dgData.DataMember)
ElseIf TypeOf dgData.DataSource Is DataTable Then
dt = dgData.DataSource
Else
Throw ex2
Exit Sub
End If
```

Once I determine that the DataGrid is correctly bound and there are rows to display, I create a Graphics object to expose the MeasureString method for the DataGrid control. Note the Graphics object *must* be disposed of when the routine exits.

```
If dt.Rows.Count > 0 Then ' Don't change size if there are no rows.
Try
dgData.BeginInit()
dr = dt.Rows(0)
' SizeF Stores an ordered pair of floating-point numbers,
' typically the width and height of a rectangle.
Dim sz As SizeF
gr = dgData.CreateGraphics
```

As discussed earlier, determine the length of the padding string.

```
' Pad "n" M-width characters
sz = gr.MeasureString(New String("M", intPadding), dgData.Font)
sngPadding = sz.Width
```

Next, I walk through the bound DataGrid header row to measure the width of each HeaderText. I use this to set the Width of the GridColumnStyles object referenced through the TableStyles collection by the MappingName string.

```
For Each dcCol In dt.Columns ' Loop through the column header
' Measure the "width" of the text in each grid column header
sz = gr.MeasureString(dgData.TableStyles(strMappingName) _
.GridColumnStyles(dcCol.Ordinal).HeaderText, dgData.Font)
dgData.TableStyles(0).GridColumnStyles(dcCol.Ordinal).Width = _
sz.Width + sngPadding
Next ' dcCol loop through each column in the bound DataTable
```

Next, I repeat the process, but this time I examine the visible data rows visiting each column and using the width of the data value or the existing width—whichever is greater. Note that I use Dispose on the Graphics object when the routine exits. Leave this off and you're bound to leak memory.

```

For i = 0 To dgData.VisibleRowCount - 2 ' Calculate width on all visible rows
  For Each dcCol In dt.Columns ' Loop through each column
    dr = dt.Rows(i) ' Navigate to visible rows
    ' Measure the "width" of the text in the first row of each column
    sz = gr.MeasureString(dr(dcCol.Ordinal).ToString, dgData.Font) _
    dgData.TableStyles(0).GridColumnStyles(dcCol.Ordinal).Width = _
    Math.Max(sz.Width + sngPadding, _
    dgData.TableStyles(0).GridColumnStyles(dcCol.Ordinal).Width)
  Next ' Next DataGrid Column
Next i ' Next DataRow
Catch ex As Exception
  MsgBox(ex.ToString)
Finally
  dgData.EndInit() ' complete graphics edit
  gr.Dispose() ' Release graphic object
End Try
End If ' Test for DataTable Rows
End Sub

```

Figure 4 – The AutoSize routine.

## Summary

Sure, this seems like a lot of trouble just to make a DataGrid look better. But let's take a quick look at the difference. First, Figure 5 shows the “before” picture of a DataGrid bound to our recordset with the default width behavior. Notice that some of the headers and much of the data is chopped off.

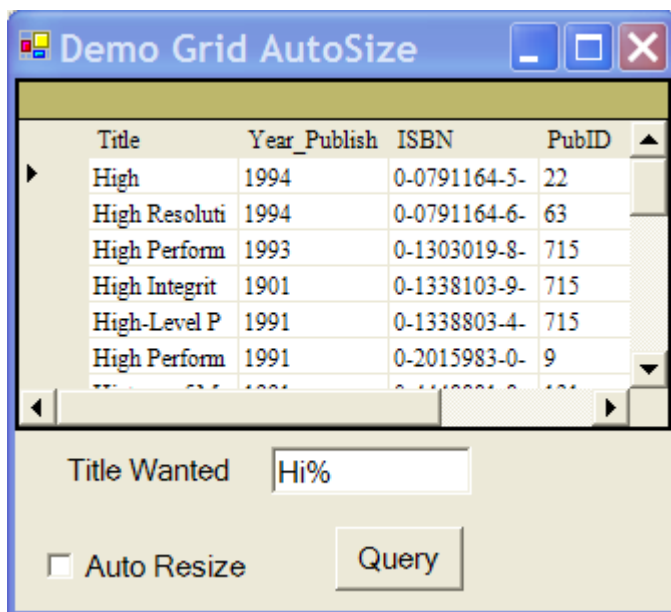


Figure 5 – “Before” autosizing.

Next, (Figure 6) shows the result after autosizing and dragging the form to conform to the new size of the data Grid.

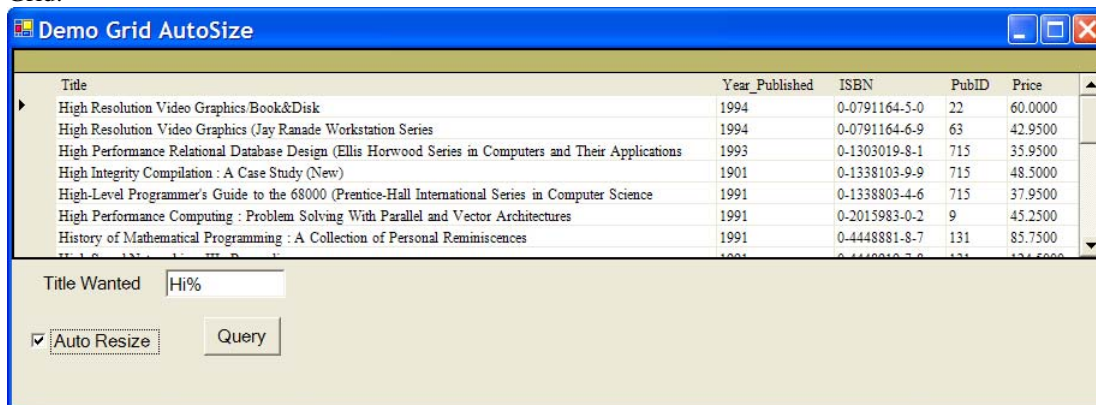


Figure 6 – After autosizing.

In this case we can see all of the data (and a bit more) and the DataGrid is far more readable. Yes, I applied an “auto format” to the DataGrid—this is simply icing on the cake. So, when size matters, (and when doesn’t it?) be sure to satisfy your user with DataGrid controls sized to fit their data.