

CIRCUIT CELLAR®

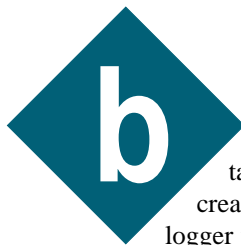
THE MAGAZINE FOR COMPUTER APPLICATIONS

LESSONS FROM THE TRENCHES

George Martin

Graphing the Data

In the July issue of *Circuit Cellar Online*, George was working on a data-logging project using Visual Basic. It took him a while to get the bugs worked out, but now he's ready to show us how to finish up this data logger.



ack in July, I talked about creating a simple data logger with a serial port to interface to a data-acquisition device using Visual Basic (VB). Remember that we could save the data to a file and I said you could load that file into a spreadsheet program and create a graph? At the time, I planned to plot the data on the screen as it was acquired, but when the time for my next column came about, I couldn't figure it out. Well, I finally figured it out.

I actually came up with several possible solutions. I could use MSChart, which is part of the VB package. It's hundreds of kilobytes of code in size. Although it works fine for plotting small amounts of static data, it didn't feel like a good match for the large amounts of dynamic data that I could be dealing with. It has too much in some areas and too little in others. Besides, with what I've already published, you could use a spreadsheet and publish gorgeous graphs. Graph32 is another tool recently released with VB,

and is somewhat smaller than MSChart, but it still feels wrong.

I could purchase a charting package. If it was just me and I was under the gun to get a project done on time, I would probably take this path. But, fortunately, I've got some flexibility about when I need to be finished.

What I need is a lean, mean charting routine, one where you can see, understand, and even modify the code to suit your specific application. Monitoring a battery-charging circuit is different than monitoring a joystick force-position curve.

Two articles helped me zero in on what I'm about to present. Ward Hitt's "Write a Lean Chart Control" (*Visual Basic Programmers Journal*, June 1998) described the exact software I was looking for. However, he offers a GraphLite ActiveX control with no source code. So, I now knew it was possible, which was a big advantage.

In Chris Barlow's article, "Make Forms Look Good on Paper" (*Visual Basic Programmers Journal*, July 99), he covers the basics of getting lines, curves, and text onto a form and out to the printer. Eureka! All the pieces just fell into place.

HOW I DID IT

It's an amazing process—this process of solving puzzles. But, I guess that's exactly why you read *Circuit Cellar* and *Circuit Cellar Online*.

I modified the code by adding VB Form to hold the chart and controls. I placed a VB PictureBox on the form as a place to draw the chart and then I added control buttons to print and refresh the chart and to close the form. All in all, it wasn't a lot of code.

The key to this programming approach is using VB's Printer Object. With VB, you can place your data on the PictureBox and then redirect all that data to the printer. The coding is

straightforward, but not so intuitive.

First, define an object (oPrinter):
Public oPrinter As Object. Next, set that object to the PictureBox on the charting form: *Set oPrinter = Picture1.* Now any of the methods that VB offers for the PictureBox object can be used to implement the chart.

And, whenever I wish to print the data in the PictureBox, I set that object (oPrinter) to a printer, repaint it, issue an *EndDoc*, and then reset the object to the PictureBox. The *EndDoc* method causes the page to be printed. Take a look at this code to see how it works:

```
Set oPrinter = Printer ' Set the out-
put to the printer
Picture1.Paint ' Repaint the output
PlotTheOldData ' Add the data to the
graph
oPrinter.EndDoc ' Print it
Set oPrinter = Picture1 ' Set the out
put to the PictureBox
```

I never was a big fan of all this object-oriented stuff. I didn't understand it and it looked like too much overhead for the embedded work I'm involved in.

Well, I've become a believer. The above code couldn't be simpler. And, it worked the first time right out of the box. It took me about four hours to get grid lines—no data yet—on the chart and plotting. Perhaps objects aren't ready for embedded code, but they do make test tools much easier.

In the PictureBox, I wanted to draw a border around the chart, label the axes, place tick marks in the data field, and label the variables. These functions are done by the *Repaint* function that is called each time the chart window needs to be repainted. I didn't repaint the data in this routine.

BACKGROUND PAINTING

I sized the picture box so that it fit comfortably on my screen. I've got a 17" monitor and I'm using 1152 × 864 (pixels) resolution. PictureBox is 12000 × 7789 (twips). If you've got a smaller screen, you could reduce the size of the PictureBox or keep the size large and implement scroll bars.

The pixel (short for picture element) is the smallest graphical unit of

measurement on the screen. Pixel size and spacing is screen dependent (i.e., its dimensions vary with the system display and resolution). A twip, on the other hand, is a screen-independent unit of measure equal to one-twentieth of a printer's point. There are approximately 1440 twips to an inch.

Next, I created a box that outlined the active area for the chart. The box's corner coordinates were:

```
TL_X 400 top left X
TL_Y 200 top left Y
BR_X 11200 bottom right X
BR_Y 7200 bottom right Y
```

And, I drew that box using the following VB statement:

```
oPrinter.Line (TL_X, TL_Y)-(BR_X,
BR_Y), B
```

Notice that I created the constants *TL_X*, *TL_Y*, etc. to define the border's outline. That way I could change the size by altering these constants and not rewriting the code.

I next put grid marks in the active area. I created *STEP_Y* and *STEP_X* constants for grid spacing with *STEP_X* being just an arbitrary setting equal to 500 twips, while *STEP_Y* was defined to represent 0 to 100% grid spacing on the y axis.

$$STEP_Y = (SIZE_Y / 255) \times 25.5$$

Remember that we have channels 1–4 that are 8-bit A/D inputs, and channels 5 and 6 that are 12-bit A/D inputs. The 8-bit inputs range from 0 to 255 counts, while the 12-bit inputs range from 0 to 4095 counts. So, a percentage seems like a good choice for y-axis scaling. You could choose engineering units or a nonlinear scaling that suits your specific application. Keep in mind that the more math you perform, the slower the graphing becomes.

The actual grid marks are drawn with this code:

```
Const GRID = 20
```

```
For x = TL_X To BR_X Step STEP_X
For y = BR_Y To TL_Y Step -
```

STEP_Y

```
oPrinter.Line (x - GRID, y)-(x +
GRID, y)
oPrinter.Line (x, y - GRID)-(x, y +
GRID)
Next y
Next x
```

Grid markers are made by drawing a horizontal line then a vertical one. The width of the lines are defined by the constant *GRID*. If you change the value of *GRID*, the size of the hash marks change.

The y-axis scaling is printed using this routine:

```
px = TL_X - 350 ' Start the text to
the left of the box
x = 0 ' Value
For y = BR_Y To TL_Y Step -STEP_Y
oPrinter.CurrentX = px
oPrinter.CurrentY = y - 100
oPrinter.Print x
x = x + 10
Next y
```

The variable *px* is the x-coordinate for the label and it remains constant. The variable *x* is the numeric value for the grid marker. The variable *y* represents the y coordinate for the label, and it changes for each grid mark. As you can see, this code fragment steps through the grid axis, labeling each point.

x-axis labeling is similar, except that no scaling is implied. Larger numbers positioned on the right work best. The data is plotted from left to right until the end is reached. At that point, the process starts over again without erasing any of the previously plotted data. This plotting style is only useful for some monitoring situations, but you may wish to modify this charting technique to find one more useful for your application. Hopefully, I've given you enough of the basics to accomplish that.

```
py = BR_Y + 100 ' Start the text to the
bottom of the box
y = 0 ' Value
For x = TL_X To BR_X Step STEP_X
oPrinter.CurrentX = x - 100
oPrinter.CurrentY = py
oPrinter.Print y
```

$$y = y + 10$$

Next x

I started to tune this loop to represent readings. If you change the size of any of the constants being plotted, how fast the readings come in, or the step size of the x direction, then all the scaling needs to be changed. I could do this with formulas, but that slows down the code. So, in keeping with the lean, mean design approach, this is it.

My last task is to print a list of the channels and the colors used for each channel. I filled in the boxes with the color just for clarity, using simple colors (the first six I came across) for the channels.

DATA PLOTTING

Separate routines are used to add the data to the graph. I separated this function because I think it's easier to understand and it makes it easier to tune the code to your particular situation.

The chart form has three controls. The first control prints what's on the form. The second closes the form and stops plotting the data. The third control refreshes the data. The charting is started by two controls on the main form. The *Show Chart* command displays the charting form and the *Begin* data gathering sends the readings to the chart.

New data is added to the chart every time a reading takes place. The data is scaled vertically for the 0 to 100% scale factor. Each reading is plotted horizontally in an incremental manner from left to right with a constant spacing between readings. I just picked an arbitrary box size to plot each point and an arbitrary step size to increment across the chart.

Once the right edge of the plotting area is passed, the data is then plotted again on the left, which just puts data points on top of data points. I could have repainted the screen to get rid of the old data, but remember the goal of this design was lean and mean. At the end of the chart, you could automatically print the page, clear the form, and start plotting. I would then add time and date to the PictureBox and perhaps a title just to keep track of everything.

Whenever another window covers the chart and the chart appears on the top once again, the chart needs to be repainted, which the routine *Picture1_Paint()* does. I've added a control button to separately repaint the data using the routine *PlotTheOldData()*. I separated plotting the data so that it wouldn't unnecessarily slow down data gathering. This approach also isolates the details of data plotting, so you can more easily change them.

Well, I hope you got something from this. I sure did. About once a year, I need a simple chart, and now I know how to do it.

George started his career in the aerospace industry in 1969. After five years at a real job, he set out on his own and cofounded a design and manufacturing firm. Typical systems that George designs include servo-motion control, graphical input and output, data acquisition, and remote control. George is a charter member of the Ciarcia Design Works Team and most recently, he's been working on the people-tracking system for Bill Gates's new house. You can reach him at george.martin@worldnet.att.net.

SOFTWARE

The source code for this article is available via this download.