

Data Marker Override Expressions in Graphs (how-to guide)

Normally Data Markers and Lines are plotted based on the results of the expression or expressions in the [Data Expression 1 to 6](#) fields under the [Dependent Data tab](#). You can override the Data Markers based on different expressions using the **Override Line/DM Expressions** tab:

For example, let's say you are generating a graph of N Value vs Depth and you want the data marker color to be tied to the layer in which the data was taken. Using the AGS database structure, there is a field in the GEOL table called GEOL_GEOL.

Let's say we had values in this field called "A", "B", "C", and "D". In the **Data Marker Color Expression 1** property we would write something like the following:

```
<<Case(<<SqlRange(_
    <<Depth>>,_
    [GEOL].[GEOL_GEOL],_
    [GEOL].[GEOL_BASE],_
    [GEOL].[Depth],_
    1_
)>>,_
= "A",255,_
= "B",16711935,_
= "C",49152,_
= "D",8421376,_
)>>
```

The SqlRange function will retrieve the GEOL_GEOL value associated with the current depth. See the Help topic for [SqlRange](#) for details of its usage.

The Case function then assigns the appropriate color based on the GEOL_GEOL value. You can get the numbers from the Symbol Design►Colors application. You must use the color numbers, not the color names. The names are not used by gINT. You can change them to whatever you wish without affecting the actual colors because gINT stores the color numbers, not names.

If the GEOL_GEOL value is not in the list, the program uses the data marker color that is dictated by the other properties that affect Data Marker color under the **Data Representation** tab. These are [Data Marker Override Color](#) and [Set Data Marker color to line color](#). The order that the program uses for determining the color is:

1. Data Marker Color Expression
2. Set Data Marker color to line color
3. Data Marker Override Color
4. Native color of the data marker in Symbol Design►Data Markers

The above **Data Marker Color Expression** property will work for one project but now on another project you have different layer codes. You would have to change the expression to reflect the new codes. This is not a maintainable solution, especially if you are working on two or more projects at the same time. One solution is to create a separate report for different projects. Another horrible solution.

There are a number of very clever and complex methods of making this work in a maintainable manner. We will show one method which we believe is the best overall technique.

Set up a new table in your project database where we will store the layer codes for the project. Let's call it "LAYER CODES". It will have a key set of "ItemKey" and we will caption that field as "Code". Add two other fields called "Description" and "Color". The Color field will have the graphic lookup of Colors. The Description field is necessary to make a legend (see below) and for AGS export to properly populate the ABBR group with these codes.

In the GEOL_GEOL field, set the lookup property of that field to this project table. When you start a new project you will populate the new table with all your codes with their associated descriptions and colors. You will then pick from this list when in the GEOL_GEOL field.

The Data Marker Color Expression property will then change to:

```
<<Let(Layer Code = _
    <<SqlRange(_
        <<Depth>>,_
        [GEOL].[GEOL_GEOL],_
        [GEOL].[GEOL_BASE],_
        [GEOL].[Depth],_
```

```

1_
)>>_
)>>_
<<Lookup(<<LAYER CODE.Color>>,<<Get(Layer Code)>>)>>

```

Now you would like to build a legend indicating which layers are represented by each color. This could be made up of one text entity that would give the layer code and, optionally, the description, and a solid-filled rectangular Polyline with the fill at the appropriate color.

The legend text entity could have the following Text property:

```

<<Let(Layer Code = _
  <<RecordItem(<<LAYER CODE.ItemKey>>,<<#>>)>>_
)>>_
<<ListBuildSepTrim(_
  ":  ",_
  <<Get(Layer Code)>>,_
  <<Lookup(_
    <<LAYER CODE.Description>>,_
    <<Get(Layer Code)>>_
  )>>_
)>>

```

This would be a repeating text entity. Under the "Repeat" tab of the properties dialog supply the appropriate values as follows:

Repeat Spacing: This would be a negative value if you wish each line to be below the previous line.

Number of Repetitions: Set a number greater than the number you believe you will ever have in a project.

Repeat Direction: Probably "Y", that is, vertical.

Repeat Variable Start: 1

Repeat Variable Increment: 1

The <<#>> in the text expression is called the "repeat variable". The above configuration will replace <<#>> with 1 and then 2, 3, etc. So the expression first retrieves the first layer code, then the second, etc. It then prints the code and description separated by a colon and two spaces (" : "). If the entity runs through 10 repetitions and you only have 4 layers, the last six will not print anything.

Now for the filled rectangles. Create the same number of rectangles as the number of repetitions you specified in the text entity and lay them out so that they will be adjacent to the text. You would specify the following properties in each:

Output Condition:

```

<<HasData(<<RecordItem(<<LAYER CODE.ItemKey>>,1)>>)>>

```

Fill Type[!Symbol]: Solid

Override Fill Color Expression:

```

<<Let(Layer Code = _
  <<RecordItem(<<LAYER CODE.ItemKey>>,1)>>_
)>>_
<<Lookup(<<LAYER CODE.Color>>,<<Get(Layer Code)>>)>>

```

In the Output Condition and Override Fill Color Expression properties, the "1" is for the first rectangle. Use 2 for the second in these two properties, 3 for the third, etc.

The output condition ensures that a rectangle will not print if there are more rectangles than there are layer codes.

The above setup is maintainable, that is, changing codes in one place, the new LAYER CODE table, automatically changes the report.