All the calculations for `Chart.FScale` and `Chart.FOffset` are performed in `TChart.CalculateTransformationCoeffs`.

Input parameters to the algorithm are:

- `BottomAxis, LeftAxis`
- `FClipRect, AMargin`
- `FCurrentExtent`

Results are stored in:

- `FScale, FOffset`
- `FCurrentExtent` (new values are written to when the algorithm finishes)

The **rX** and **rY** local variables are in fact records, holding temporary data for calculations. They hold the following data:

- `FAxis: TChartAxis`
- `FImageLo, FImageHi: Integer`
- `FLo, FHi: Integer`
- `FMin, FMax: PDouble`

```
procedure TChart.CalculateTransformationCoeffs(const AMargin: TRect);
var
  rX, rY: TAxisCoeffHelper;
begin
  rX.Init(
    BottomAxis, FClipRect.Left, FClipRect.Right, AMargin.Left, -AMargin.Right,
    @FCurrentExtent.a.X, @FCurrentExtent.b.X);
  rY.Init(
    LeftAxis, FClipRect.Bottom, FClipRect.Top, -AMargin.Bottom, AMargin.Top,
    @FCurrentExtent.a.Y, @FCurrentExtent.b.Y);

  FScale.X := rX.CalcScale(1);
  FScale.Y := rY.CalcScale(-1);

  if Proportional then begin
    if Abs(FScale.X) > Abs(FScale.Y) then
      FScale.X := Abs(FScale.Y) * Sign(FScale.X)
    else
      FScale.Y := Abs(FScale.X) * Sign(FScale.Y);
  end;

  FOffset.X := rX.CalcOffset(FScale.X);
  FOffset.Y := rY.CalcOffset(FScale.Y);

  rX.UpdateMinMax(@XImageToGraph);
  rY.UpdateMinMax(@YImageToGraph);
end;
```

As we can see, the code used for horizontal axis (i.e. `BottomAxis`) is corresponding to the code for vertical axis (i.e. `LeftAxis`), **with two exceptions**:

- in the `rY.Init()` call, `Bottom` and `Top` parameters are passed in the reverse order – a natural order would be `Top` and then `Bottom`
- in the `rY.CalcScale()` call, `-1` is passed instead of `1`

0) Now let's take a look at the algorithm for vertical axis:

```
rY.Init(...) call:
  FAxis := AAxis;
  FImageLo, FImageHi are initialized
  FMin, FMax are initialized
  FLo, FHi are initialized

rY.CalcScale(...) call (after required bug patching):
  if (FMax^ <= FMin^) or (Sign(FHi - FLo) <> ASign) then
    Result := ASign
  else
    Result := (FHi - FLo) / (FMax^ - FMin^);
  if (FAxis <> nil) and FAxis.IsFlipped then
    Result := -Result;

rY.CalcOffset(...) call:
  Result := (FLo + FHi) / 2 - AScale * (FMin^ + FMax^) / 2;

rY.UpdateMinMax(...) call:
  FMin^ := AConv(FImageLo);
  FMax^ := AConv(FImageHi);
  if (FAxis <> nil) and FAxis.IsFlipped then
    Exchange(FMin^, FMax^);
```

1) Improvements, step 1:

- replace `FAxis: TChartAxis` variable with `FAxisIsFlipped: Boolean` variable,
- instead of dividing by 2, multiply by 0.5 (multiplying is faster than dividing - and since both 2 and 0.5 values can be represented in floating-point math exactly, with no roundings (because they are both powers of 2, and Double is also internally represented by using powers of 2), we can switch between them without introducing any additional math errors (i.e. differences in the calculation's result, appearing somewhere on least significant decimal digits):

```
rY.Init(...) call:
  FAxisIsFlipped := (FAxis <> nil) and FAxis.IsFlipped;
  FImageLo, FImageHi are initialized
  FMin, FMax are initialized
  FLo, FHi are initialized

rY.CalcScale(...) call:
  if (FMax^ <= FMin^) or (Sign(FHi - FLo) <> ASign) then
    Result := ASign
  else
    Result := (FHi - FLo) / (FMax^ - FMin^);
  if FAxisIsFlipped then
    Result := -Result;

rY.CalcOffset(...) call:
  Result := ((FLo + FHi) - AScale * (FMin^ + FMax^)) * 0.5;

rY.UpdateMinMax(...) call:
  FMin^ := AConv(FImageLo);
  FMax^ := AConv(FImageHi);
  if FAxisIsFlipped then
    Exchange(FMin^, FMax^);
```

2) Improvements, step 2:

Let's pass parameters to the `rY.Init(...)` call in their natural `Top .. Bottom` order; this will effectively exchange `FImageLo` and `FImageHi` values, and also `FLo` and `FHi` values – as a consequence, the „`FHi - FLo`" clause will have the opposite sign:

```
rY.Init(...) call:
  FAxisIsFlipped := (FAxis <> nil) and FAxis.IsFlipped;
  FImageLo, FImageHi are initialized inversely
  FMin, FMax are initialized
  FLo, FHi are initialized inversely


rY.CalcScale(...) call:
  if (FMax^ <= FMin^) or (Sign(FHi - FLo) <> 1) then // we replaced ASign with 1 here:
                            // for X axis, ASign = 1, so nothing changes in this case;
                            // for Y axis, ASign = -1, but we needed to replace
                            // ASign with –ASign, which effectively also gives 1

    Result := 1 // we replaced ASign with 1 here;
                // for X axis, ASign = 1, so nothing changes in this case;
                // IMPORTANT: we'll receive an opposite
                // sign here for Y axis
                // => we must change Result's sign for Y axis

  else
    Result := (FHi - FLo) / (FMax^ - FMin^); // IMPORTANT: we'll receive an opposite
                                             // sign here for Y axis
                                             // => we must change Result's sign for Y axis
  if FAxisIsFlipped then
    Result := -Result;
  if AxisIsVertical then
    Result := -Result;


rY.CalcOffset(...) call:
  Result := ((FLo + FHi) - AScale * (FMin^ + FMax^)) * 0.5; // FLo + FHi has still
                                                              // same value, so nothing
                                                              // changes here

rY.UpdateMinMax(...) call:
  FMin^ := AConv(FImageLo);
  FMax^ := AConv(FImageHi); // IMPORTANT: FImageLo works as FImageHi for Y axis,
                            // and FImageHi works as FImageLo for Y axis
                            // => we must exchange FMin^ with FMax^ for Y axis

  if FAxisIsFlipped then
    Exchange(FMin^, FMax^);
  if AxisIsVertical then
    Exchange(FMin^, FMax^);
```

3) Improvements, step 3 – final tuning:

Changing `Result`'s sign twice is same as not changing at all; exchanging two values twice is same as not exchanging at all. So we can use:

```
  if FAxisIsFlipped xor AxisIsVertical then
    Result := -Result;
```

and:

```
  if FAxisIsFlipped xor AxisIsVertical then
    Exchange(FMin^, FMax^);
```

Since, in all cases, `FAxisIsFlipped` is used along with `AxisIsVertical`, we can just pass an additional `AAxisIsVertical` parameter to the `rY.Init(...)` call and initialize the `FAxisIsFlipped` variable as:

```
  FAxisIsFlipped := ((FAxis <> nil) and FAxis.IsFlipped) xor AAxisIsVertical;
```

The „`(Sign(FHi - FLo) <> 1)`" clause can be represented more simply as „`(FHi <= FLo)`".

Since `CalcScale(...)` call doesn't use its `ASign` parameter anymore, we can remove it.

The final code will be:

```
rY.Init(..., AAxisIsVertical) call:
  FAxisIsFlipped := ((AAxis <> nil) and AAxis.IsFlipped) xor AAxisIsVertical;
  FImageLo, FImageHi are initialized
  FMin, FMax are initialized
  FLo, FHi are initialized

rY.CalcScale() call:
  if (FMax^ <= FMin^) or (FHi <= FLo) then
    Result := 1
  else
    Result := (FHi - FLo) / (FMax^ - FMin^);
  if FAxisIsFlipped then
    Result := -Result;

rY.CalcOffset(...) call:
  Result := ((FLo + FHi) - AScale * (FMin^ + FMax^)) * 0.5;

rY.UpdateMinMax(...) call:
  FMin^ := AConv(FImageLo);
  FMax^ := AConv(FImageHi);
  if FAxisIsFlipped then
    Exchange(FMin^, FMax^);
```

where:

```
rX.Init(
  BottomAxis, FClipRect.Left, FClipRect.Right, AMargin.Left, -AMargin.Right,
  @FCurrentExtent.a.X, @FCurrentExtent.b.X, False);
rY.Init(
  LeftAxis, FClipRect.Top, FClipRect.Bottom, AMargin.Top, -AMargin.Bottom,
  @FCurrentExtent.a.Y, @FCurrentExtent.b.Y, True);
```

This seems to be more understandable than the current implementation.