基于 C#的波形显示控件的实现

编者记：

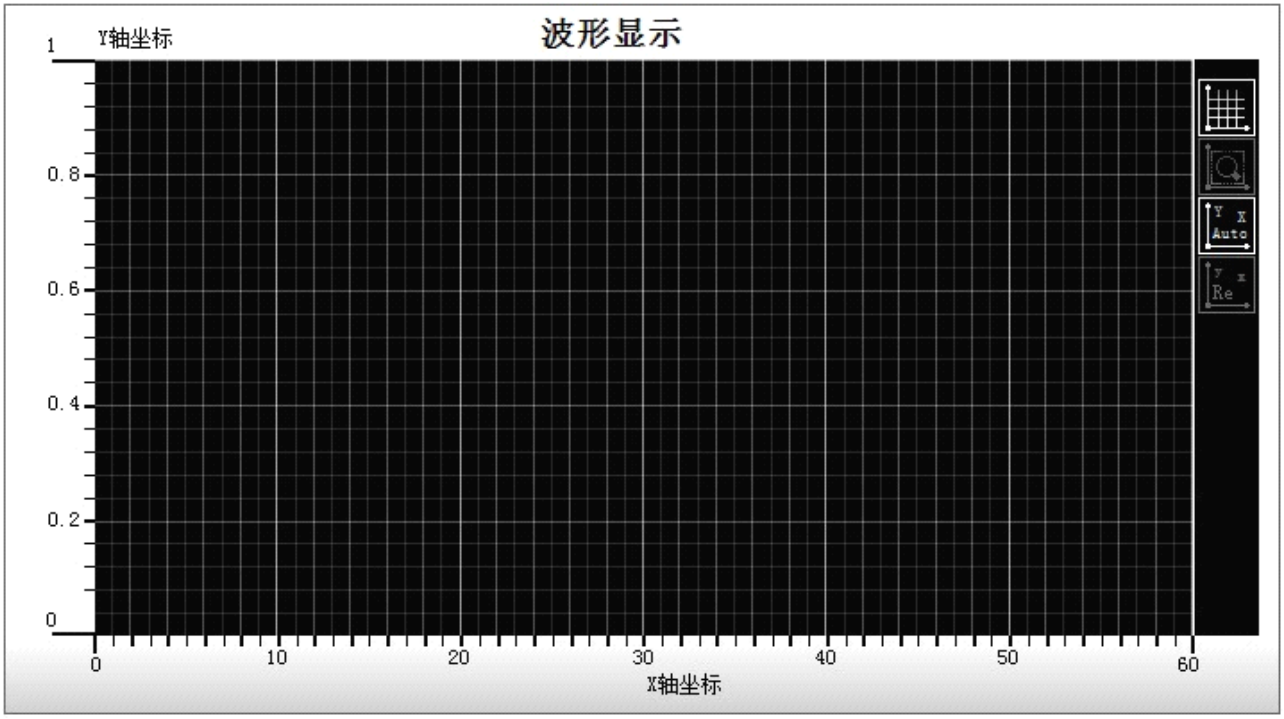
09年暑假正好在学院实验室呆了一段时间，做了个完整的上位机软件（具体实现：根据下位机的指令，实现通过串口来操纵下位机进行实验，并将采集的数据进行处理和保存，并以图形的方式显示），整个项目边学 C# WinForm 边设计，这个波形显示控件就是项目中的一部分，也花了自己绝大多数时间。此外，顺便将该波形显示控件当作自己毕业设计的内容，下文实际上是节选自自己的本科毕业论文，希望对大家能有所帮助。代码以及文章有疏漏、错误、不妥之处在所难免，欢迎交流

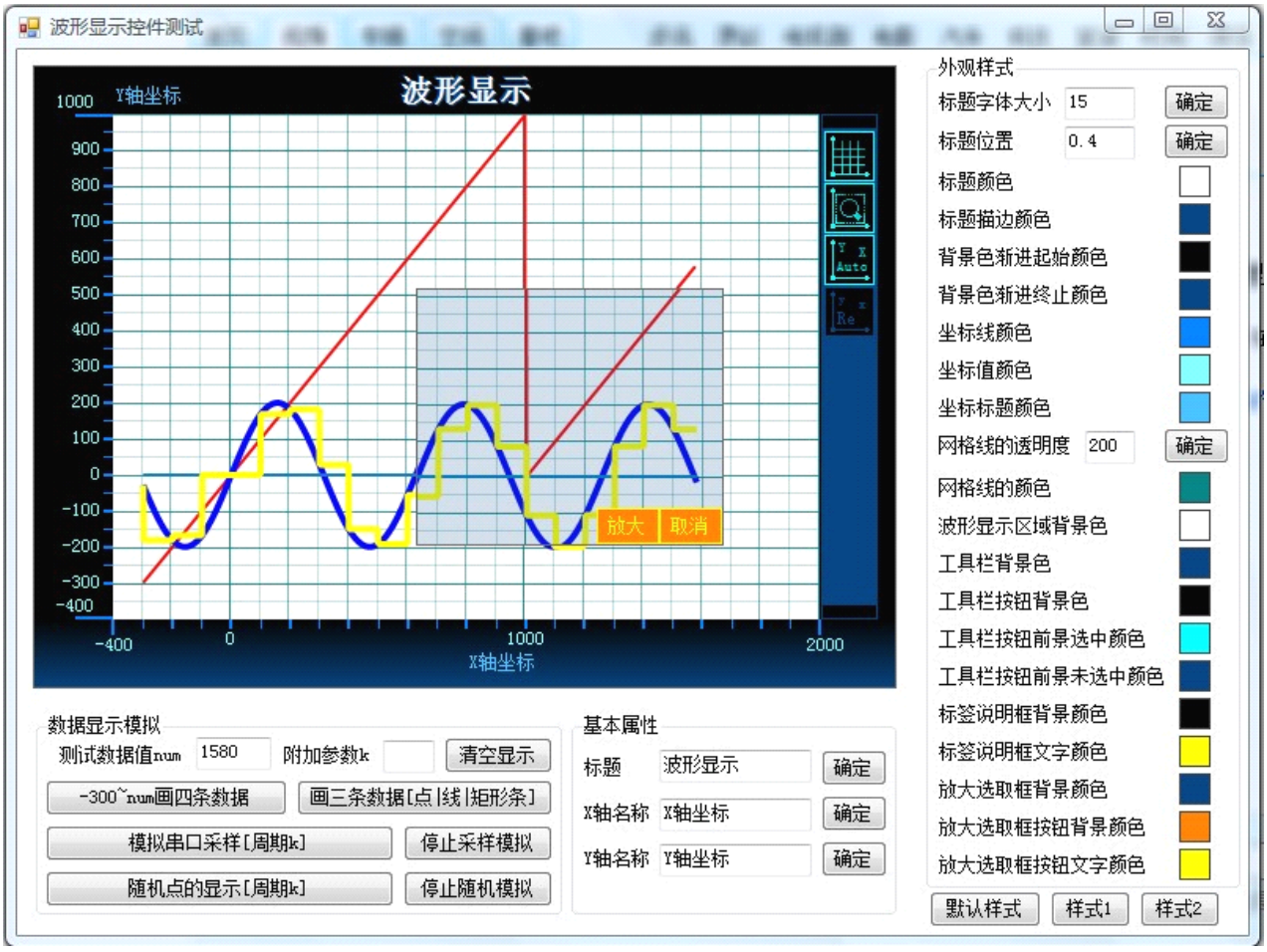
“基于 C#的波形显示控件”设计于09年暑假 **源代码：** 博客园下载 | CSDN 下载 **解压密码：** CSharp_WinForm_Waveform

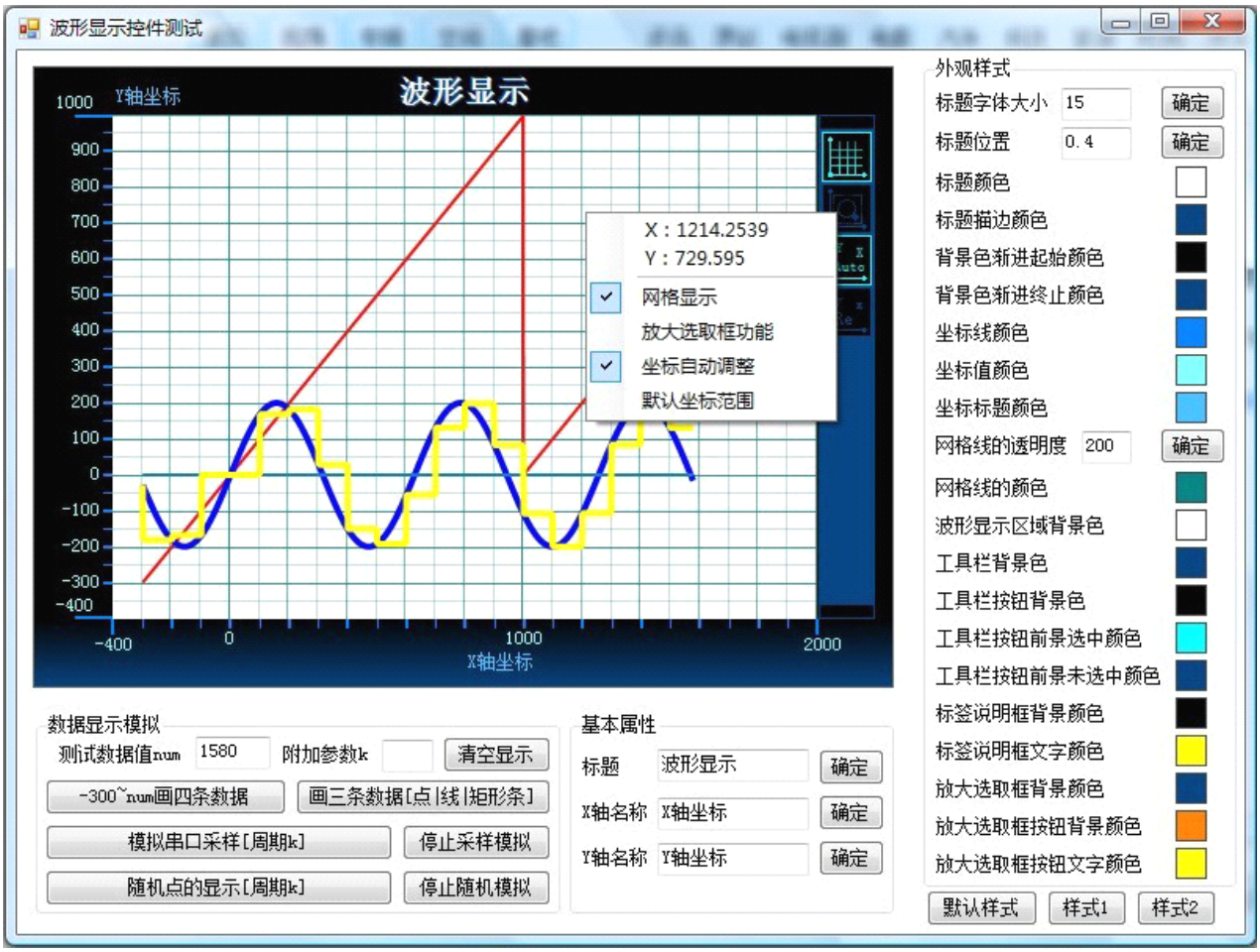
《基于 C#的波形显示控件的实现》写于2009年9月

“基于 C#的波形显示控件的演示程序”为配套控件使用而设计的（演示程序源代码同控件源代码一起奉上了）

下面先上几张图，大家可以先了解最后做出来是个什么东西：







再奉上两个波形显示控件的**演示视频**（两个视频内容差不多的呵呵）：

摘　　要

计算机技术的飞速发展使得其在自动化系统中的应用日益增强。大量监控、图像数据显示软件活跃在自动化工业及自动化教学领域。同时，软件系统的日益复杂化使得模块化开发变得尤为重要。本课题所设计的基于 C#的波形显示控件就可在微软.NET 平台下进行代码功能重用，达到模块化开发和快速开发的目的，使得程序员能够集中精力设计软件的具体业务流程，而不必担心波形呈现的问题。

本文先介绍了.NET 平台下用户控件开发的基本方法，以及用 C#描述的 GDI+图形开发技术，然后提出一种基于 C#的波形显示控件的设计思路，并对波形坐标值转换、坐标标尺、工具栏、局部放大等具体的设计细节进行详细解析。

本课题设计的波形显示控件实现了同时显示多条数据曲线、局部放大查看、波形显示自动调整最佳坐标范围、动态显示波形等功能。创新之处在于设计了一种方法，使得波形显示控件的坐标轴的起点值和终点值能够以浮点数显示，并自动根据当前波形显示控件的大小，描绘出符合用户视觉的坐标标尺。

最后，还设计了一个波形显示控件的演示程序，用于介绍该控件的使用方法以及功能测试，并提出了该波形显示控件可改进的地方。

关键词：波形显示控件；C#；GDI+；动态波形

目　录

1 绪论

1.1 课题背景

1.2 波形显示控件实现的功能

2 主要开发技术介绍

2.1 .NET 用户控件介绍

2.2 GDI+技术介绍

3 波形显示控件整体设计

3.1 数据存储结构的设计

3.2 控件界面模块的设计

3.3 控件工作流程的设计

4 波形显示控件各细节的实现

4.1 坐标值和标尺的实现

4.1.1 坐标相关的成员变量

4.1.2 坐标标定权值的概念

4.1.3 坐标标尺的绘制

4.1.4 子标尺线的选择性显示

4.2 数据点的描绘

4.2.1 数据值转换为坐标值

4.2.2 溢出坐标范围的数据点的处理

4.2.3 遍历所有数据线并绘制出

4.3 波形显示区域网格的实现

4.3.1 网格相关的成员变量

4.3.2 网格的绘制

4.4 工具栏按钮的实现

4.4.1 工具栏按钮相互关系

4.4.2 工具栏提示标签的实现

4.5 波形放大功能的实现

4.5.1 局部放大选择框的实现

4.5.2 放大选择框的鼠标操作

4.5.3 放大选择框的按钮操作

4.5.4 更新数据显示范围为放大的范围

4.6 坐标自动调整及恢复默认坐标的实现

4.6.1 坐标自动调整功能

4.6.2 恢复默认坐标范围功能

4.7 波形显示控件接口的实现

4.7.1 控件基本属性

4.7.2 控件外观样式

4.7.3 控件绘图接口
4.8 波形显示控件其他细节的处理
4.8.1 坐标值产生遮盖时的处理
4.8.2 波形显示控件大小改变时的处理
4.8.3 按钮点击时进行禁用操作
4.8.4 右键菜单的显示
4.8.5 XML 注释以及智能提示
5 波形显示控件功能的演示和使用
5.1 波形显示控件演示程序的设计
5.2 波形显示控件功能的演示
5.2.1 外观样式的更改
5.2.2 波形显示演示
5.2.3 波形显示控件在实际项目中的使用
6 课题总结
参考文献

1 绪论

1.1 课题背景

波形显示控件广泛见于监控测量，图像数据显示等自动化相关软件中，更是组态软件必不可少的一部分。例如美国国家仪器有限公司（National Instruments）的 NI Measurement Studio 集成式套件以及 LabView 图形化程序开发环境等，都包含技术成熟的波形显示控件，功能丰富且强大。

目前购买成熟的自动控制相关的第三方控件库往往需要支付较高的费用，且大多数项目只用到些许专业性控件，例如一个喷管实验平台软件，图形用户界面只需要波形显示控件、U 型差计控件和压力表控件，因此不适合科研机构以及中小型软件开发公司。所以掌握简单的控件开发技术，并开发属于自己团队或公司的控件库，从而降低软件开发成本，显得十分必要。

微软的.NET 解决方案依据其对已有代码的互操作性、简化部署、分布式、高效开发等诸多优点，完全胜任自动化系统软件的开发。C#是微软为.NET 平台开发的一门语言，通过.NET 平台可以轻松使用 GDI+技术开发可重用的用户图形界面控件。GDI+是目前在 Windows 窗体应用程序中以编程方式呈现图形的唯一方法，它使程序设计者可以创建图形、绘制文本以及将图形图像作为对象操作，旨在提供较好的性能并且易于使用。

波形显示控件因为涉及到坐标系、数据显示方式、精确度、实时性等因素，存在一定的开发难度。本课题意在使用微软.NET 平台下的 C#语言，提供一种坐标计算方法，解决自动调整最佳坐标范围，以符合用户视觉的方式显示坐标标尺，并通过 GDI+技术显示波形数据，具有一定的实际开发参考价值。开发完成后的波形显示控件，可以快速嵌入到.NET 平台下的软件工程中，开发者只需简单的操作，就可以使用该控件友好地显示波形数据，从而达到快速开发的目的。

1.2 波形显示控件实现的功能

本课题所设计的波形显示控件，主要实现以下功能：

（1） 多条波形数据的显示。

该波形显示控件能够同时显示多条波形数据，用户能够控制每条波形数据的显示颜色、线宽、线帽、以及线转折的样式。并且提供了三种波形数据显示的方式：连续数据线、离散点、条形图。

（2） 友好坐标标尺的显示。

该波形显示控件能够根据当前显示数据的坐标范围，友好地显示坐标标尺。例如 X 轴坐标起始值为34.2，结束坐标值为100.7，则控件不是简单得将坐标10等分并显示并不友好的坐标值，而是通过计算当前波形显示控件的大小，显示40、50……这样的友好的坐标值，并判断是否需要继续在40到50的坐标值之间显示更小分度的坐标值。

（3） 波形显示区域网格的显示。

该波形显示控件可以显示同坐标标尺的坐标线相对应的网格，使得用户能够更直观地观察波形数据。

（4） 波形的局部放大。

该波形显示控件提供了波形局部放大的功能。并根据实际使用和测试，控制了波形放大的精度，以免产生数据溢出的问题。

（5） 坐标自动调整。

该波形显示控件能够根据当前要显示的波形数据的值，自动选择最佳的坐标范围，来直观地在控件的波形显示区域显示完整的波形曲线。

（6） 外观颜色方案的修改。

该波形显示控件能够修改外观样式，诸如背景色、网格颜色、坐标线颜色、坐标值颜色等都可以进行调整，以使控件外观能够符合软件整体风格。

（7）　其他细节。

该波形显示控件还设计了一个工具栏，可以方面地使用网格显示、局部放大、坐标自动调整、恢复默认坐标功能。另外还设计右键菜单，能够显示当前鼠标位置的具体坐标值，以及工具栏按钮的快捷按钮。

2　主要开发技术介绍

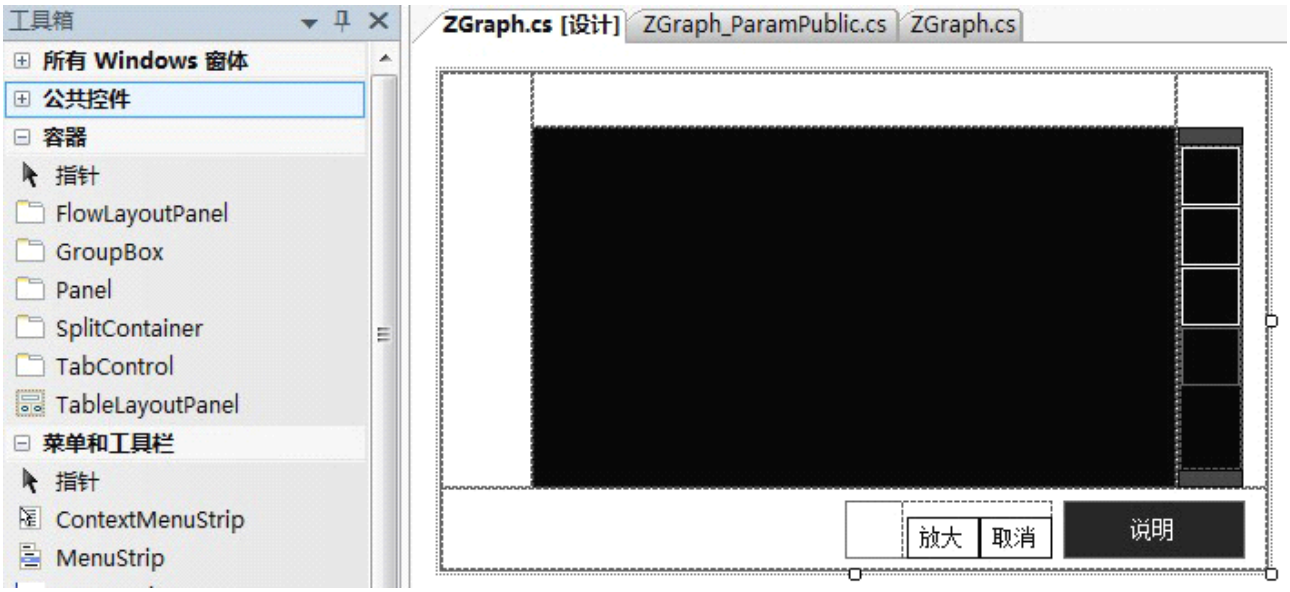
2.1　.NET 用户控件介绍

在微软.NET 平台下，可以设计出在应用程序内部或应用程序之间提供一致性行为和用户界面的复合控件，即用户控件。用户控件可以是某个应用程序的本地控件，也可以编译成 DLL 供多个应用程序使用。

用户控件极大限度的帮助程序员进行代码重用和快速开发。通常用户控件通过继承 UserControl 基类，并组合微软提供的基础控件，如 Button 控件、PictureBox 控件、Label 控件等，然后进行功能的扩充和组合，从而创建出特定功能的用户控件。

在 Visual Studio 2005下使用 C#开发.NET 用户控件，可以通过新建 Windows 控件库来实现。创建成功后软件会自动从 UserControl 基类继承一个类，并允许通过设计视图添加所需要的已有控件。

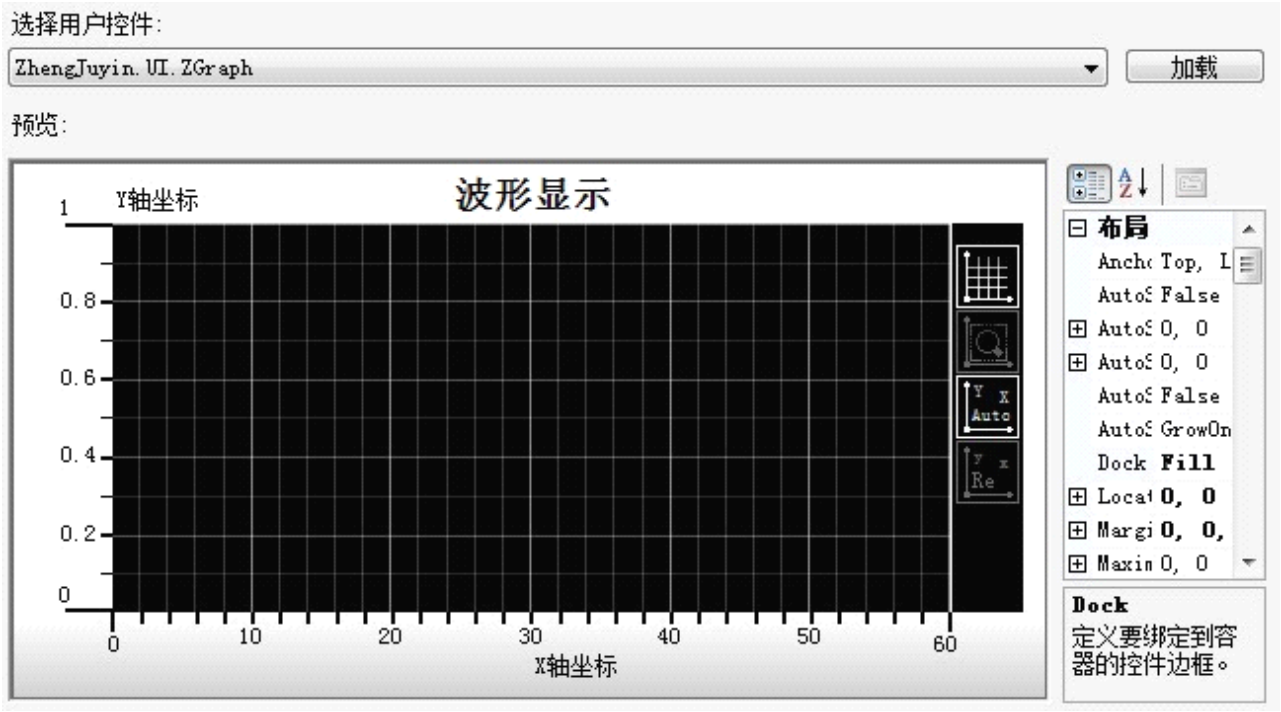
图 2- 1　设计用户控件界面



设计好大致的界面后，就可以着手设计具体的代码，用户控件通常通过对消息处理函数的编写，来实现特定的功能，并将控件中的各个模块联系起来。用户控件还需要设计出公开的接口属性和方法给控件的使用者，这样控件的使用者就能够调用公开的属性和方法，来实现对控件的操作和控制。另外，在编写控件的时候对各个属性和方法编写 XML 注释，并在生成控件的时候输出 XML 文档文件，这样控件使用者在使用该控件的时候，能够享受 Visual Studio 智能提示带来的便利。

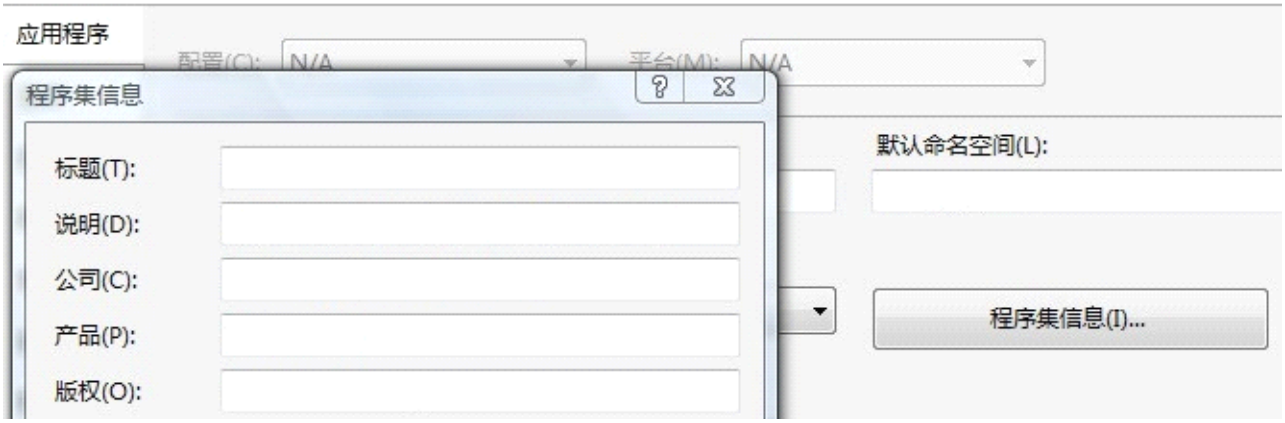
编译运行后可以检查基本的控件效果：

图 2- 2　用户控件的调试运行



最后可以通过项目属性设置程序集信息。

图 2- 3　设置程序集信息



这样，一个用户控件就设计完成了。

用户控件自身不能独立作为程序运行，需要嵌入到特定的软件中。可以通过选择工具箱，添加新创建的.NET Framework 组件，添加到工具箱中。其用法同微软提供的基础控件一样，可以通过拖拽的方式进行使用。

2.2 GDI+技术介绍

GDI+是.NET Framework 类库，用于进行图形编程。它是一种基于 GDI 的技术，GDI 即 Windows Graphical Device Interface，是 Windows API（应用程序编程接口）中处理图形的部分。

GDI+是目前在 Windows 窗口应用程序中以编程方式呈现图形的唯一方法，它使程序设计者可以创建图形、绘制文本以及将图形图像作为对象操作，旨在提供较好的性能并且易于使用。

GDI+技术在 Windows 应用程序中得到了广泛的应用，使得 Windows 图形图像编程变得格外得容易。目前 Windows 平台下的大量应用软件的图形界面通过 GDI+技术实现。利用 GDI+技术可以方便地在屏幕、打印机或者其他输出设备上输出图形、文本等内容，程序员无需关心硬件设备和设备驱动，就可以编写出设备无关的应用程序。一些新兴的信息化软件通过 GDI+技术能实现动态交互的高级效果，比如可视化电子地图。

在使用 GDI+绘图之前，需要获得绘图对象 Graphics，通常在编写控件的 Paint 事件时，Graphics 对象可通过 PaintEventArgs 参数获得。所有的绘制过程都是面向对象的，例如实例化1像素的实线黑色画笔并绘制线条，可用如下的方式创建：

```
Graphics Grap = e.Graphics;
Pen p = new Pen(Color.Black);
Grap.DrawLine(p, 0, 0, 100, 100);
p.Dispose();
```

若填充一个区域的颜色，则代码样式如下所示：

```
SolidBrush b = new SolidBrush(Color.Red);

Grap.FillRectangle(b,0,0,10,10);

b.Dispose();
```

其直观的操作方式使得程序员能快速绘制椭圆，椭圆弧，矩形，贝塞尔曲线，路径等。

通过 GDI+技术创建自定义控件能实现各种功能。只需要在建立自定义控件后，编写响应 Paint 事件的代码，通过 GDI+技术绘制必须的图形，就可以控制自定义控件的外观。本课题设计的波形显示控件，大量得运用了 GDI+技术进行绘图操作，使得控件能够呈现出出色的界面效果。

3 波形显示控件整体设计

3.1 数据存储结构的设计

本课题设计的波形显示控件主要用于显示数据点集合。数据点集合具有大小不确定性，因此可以使用 .NET 提供的泛型支持，其主要的优点是性能，因此要显示的数据集合可以使用泛型集合中的 List<T>类存储，List<T>类可以动态增大和减少其容量。同时考虑到波形显示控件需能够同时显示多条曲线，且数据点集合根据 X 轴的值和 Y 轴的值分别创建 List<float>。

经过上面分析，得出波形显示控件内部需要创建两个 List<List<float>>对象，一个用于存储要显示的数据点集合的 X

轴值的引用，另一个用于存储要显示的数据点集合的 Y 轴值的引用。另外，还需要创建 List<color>、List<int>、List<LineJoin>、List<LineCap>、List<DrawStyle>分别来存储每条要显示的数据点集合的颜色、线条宽度、连接点样式、起始线帽、样式。其中 DrawStyle 为自定义枚举，用于标识画图样式：线条、点或者条形图。具体代码设计如下所示：

```
public enum DrawStyle{ Line,dot,bar }

private List<List<float>> _listX = new List<List<float>>(); //X 轴数据集

private List<List<float>> _listY = new List<List<float>>(); //Y 轴数据集

private List<Color> _listColor = new List<Color>(); //线条颜色

private List<int> _listWidth = new List<int>(); //线条宽度

private List<LineJoin> _listLineJoin = new List<LineJoin>(); //连接点

private List<LineCap> _listLineCap = new List<LineCap>(); //起始线帽

private List<DrawStyle> _listDrawStyle = new List<DrawStyle>(); //样式
```

由上面可知，波形显示控件实际上本身是不存储实际的数据值的，而相当于存储了要显示的数据的“引用”，即实现的是一个无源控件，这样有利于数据处理层和数据呈现层的分离，使软件更趋向于模块化。另一方面，为实现多波形显示带来了效率：只需引用传递波形显示控件要显示的数据，而无需整体地复制所有的显示数据给波形显示控件。

3.2 控件界面模块的设计

控件的界面可以通过使用微软提供的基础控件进行布局。

首先使用5个 pictureBox 控件分别表示底部的 X 轴坐标区域、左侧的 Y 轴坐标区域、顶部标题栏区域、右侧工具栏区域、以及中间的波形显示区域。除中间波形显示区域的 pictureBox 控件外，其余4个 pictureBox 控件大小有所限制，这样有助于坐标显示的计算。具体参数如下表所示：

表格 3- 1　控件界面5大模块

区域	控件类型	控件名称	控件大小
底部 X 轴坐标区域	PictureBox	pictureBoxBottom	高度固定45像素
左侧 Y 轴坐标区域	PictureBox	pictureBoxLeft	宽度固定50像素
顶部标题栏区域	PictureBox	pictureBoxTop	高度固定30像素
右侧工具栏区域	PictureBox	pictureBoxRight	宽度固定50像素
中间波形显示区域	PictureBox	pictureBoxGraph	不设置固定值

右侧工具栏区域中，需要更多控件来标识工具栏，参数如下表所示：

表格 3- 2　工具栏模块

作用	控件类型	控件名称	控件大小
底部背景	Panel	panelControlItem	36,178
内部滑动背景	Panel	panelItemsIN	34,178
网格显示按钮	Button	buttonLinesShowXY	32,32
放大选取框功能按钮	Button	buttonBigModeXY	32,32
坐标自动调整按钮	Button	buttonAutoModeXY	32,32
默认坐标范围按钮	Button	buttonReXY	32,32
工具栏上移按钮	Button	buttonControlItemUP	36,10
工具栏下移按钮	Button	buttonItemsDown	36,10
工具栏提示标签	Panel	labelItemShuoMing	100,32

其中的内部滑动背景 Panel、工具栏上移按钮和工具栏下移按钮是为今后功能扩充的时候预留的。当工具栏的工具按钮过多的时候，可以通过上移按钮和下移按钮进行工具栏中项目的滚动。本课题实际设计过程中，上移按钮和下移按钮是处于禁用状态。另外，工具栏提示标签初始为隐藏状态，只有在鼠标经过工具栏按钮的时候，工具栏提示标签则移动到指定位置处并显示相应的提示文字。

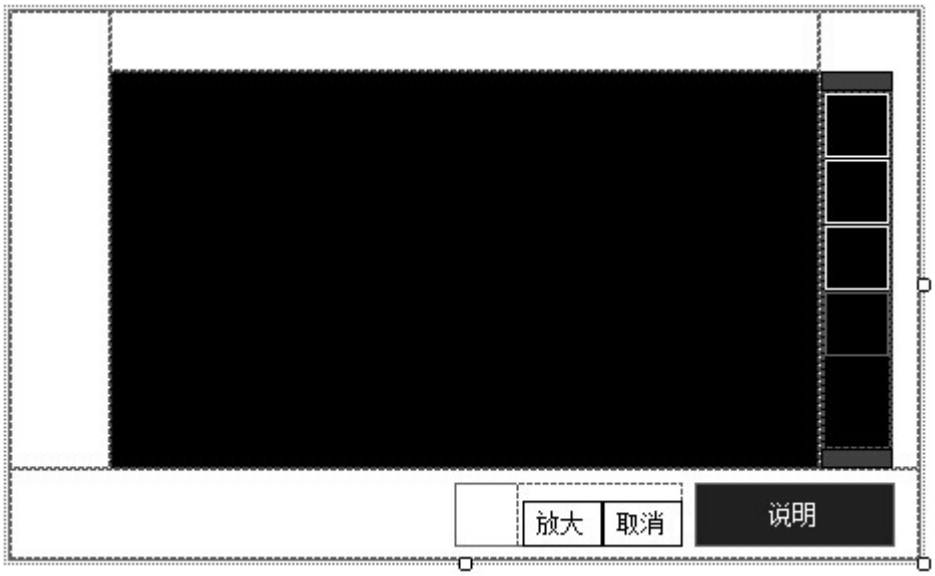
在实际使用过程中，还存在一个波形局部放大的功能，可以通过一个半透明的 PictureBox 控件实现，另外还需要两个按钮：放大和取消。参数如下表所示：

表格 3- 3　波形局部放大模块

作用	控件类型	控件名称	控件大小
半透明的放大框	PictureBox	pictureBoxBigXY	根据实际操作
两个按钮的容器	Panel	panelBigXY	83,32
放大按钮	Button	buttonBigXYBig	40,32
取消按钮	Button	buttonBigXYQuit	40,32

具体设计结果如下图所示：

图 3- 1　控件界面模块的设计



波形显示区域还提供了右键菜单的功能，具体参数如下表：

表格 3- 4右键菜单模块

作用	控件类型	控件名称
标识菜单	ContextMenuStrip	MenuRightClick
当前位置 X 坐标值	ToolStripTextBox	ToolStripTextBoxX
当前位置 Y 坐标值	ToolStripTextBox	ToolStripTextBoxY
网格显示	ToolStripMenuItem	网格显示 ToolStripMenuItem
放大选取框功能	ToolStripMenuItem	放大选取框功能 ToolStripMenuItem
坐标自动调整	ToolStripMenuItem	坐标自动调整 ToolStripMenuItem
默认坐标范围	ToolStripMenuItem	默认坐标范围 ToolStripMenuItem

3.3　控件工作流程的设计

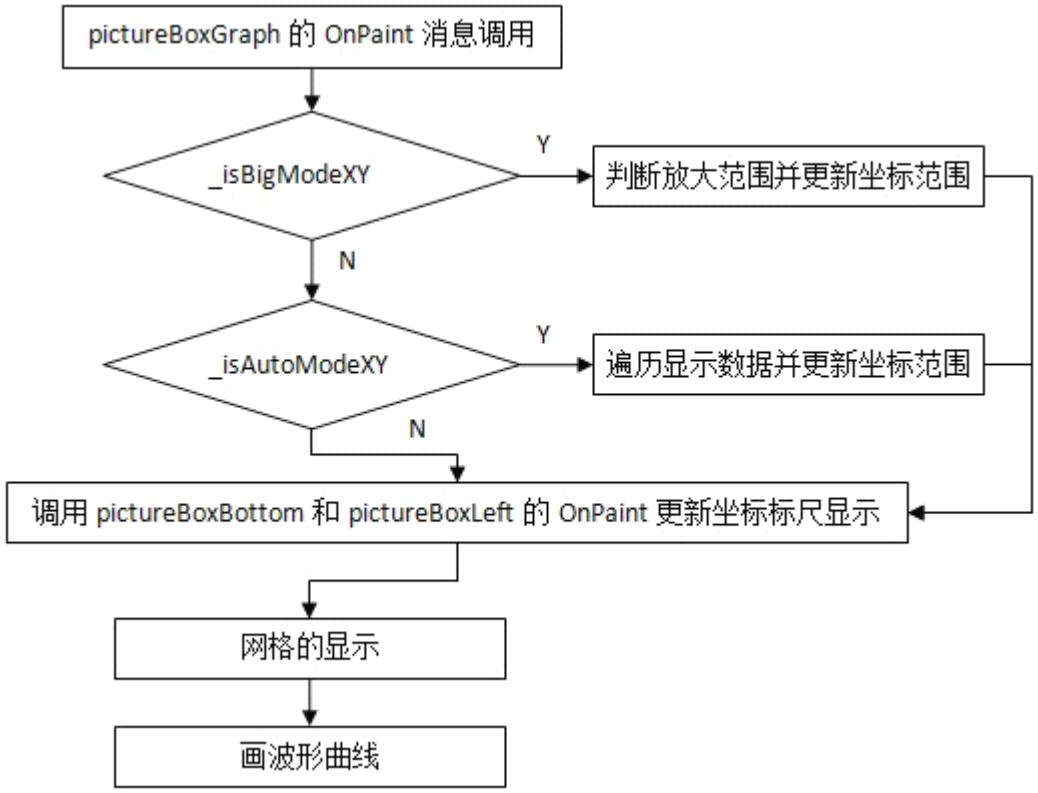
波形显示控件设置了两个影响显示方式的标记：标识当前是否处于放大查看模式，以及标识当前坐标是否自动调整以适合窗口大小。作为波形显示控件的私有成员变量，如下所示：

```
private bool _isBigModeXY = false; //是否处于放大查看模式

private bool _isAutoModeXY = true; //是否自动调整以适合窗口大小
```

若当前处于放大模式，则波形在动态显示的时候，并不自动改变 X 轴和 Y 轴的坐标范围（即使有数据点越出波形显示控件的显示范围），只有在放大选取框进行局部放大的时候改变 X 轴和 Y 轴的坐标范围。若当前非放大模式，且为自动调整坐标模式，则遍历要显示的数据值，确定要显示的最大坐标范围，然后修改 X 轴和 Y 轴的坐标范围。整个波形显示控件的显示方式判断和实现过程，设计为在中间波形显示区域 pictureBoxGraph 的 OnPaint 消息中。

具体波形显示的工作流程如下所示：

图 3- 2　控件工作流程



4　波形显示控件各细节的实现

4.1　坐标值和标尺的实现

4.1.1　坐标相关的成员变量

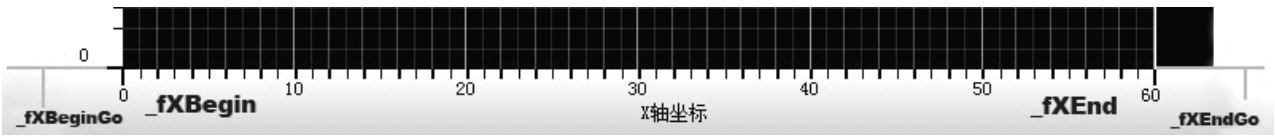
因为控件存在局部放大的功能，所以局部放大后的坐标轴起始值和结束值通常会是浮点数。仅仅根据坐标轴两点的浮点数值难以描绘出符合用户视觉的坐标线和坐标值。因此引入了坐标标定值的概念，得到下列私有成员变量：

```
private float _fXBegin; //当前显示波形的 X 轴起始坐标值

private float _fXEnd; //当前显示波形的 X 轴结束坐标值

private float _fYBegin; //当前显示波形的 Y 轴起始坐标值

private float _fYEnd; //当前显示波形的 Y 轴结束坐标值

private float _fXBeginGO; //当前显示波形的 X 轴坐标标定起始值
```

```
private float _fXEndGO; //当前显示波形的 X 轴坐标标定结束值

private float _fYBeginGO; //当前显示波形的 Y 轴坐标标定起始值

private float _fYEndGO; //当前显示波形的 Y 轴坐标标定结束值
```

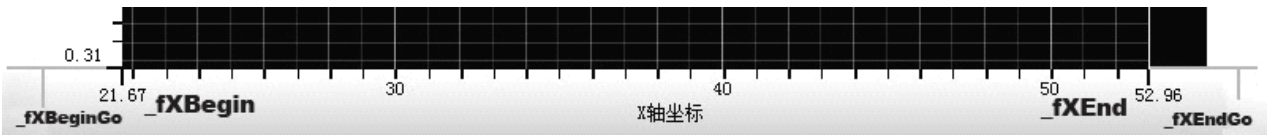
具体的意义如下图所示：

图 4- 1 坐标相关成员变量的意义



由上图可知，_fXBegin 和_fXEnd 代表的即为实际的坐标起始值和结束值（如上图中的起点值0和终点值60），而_fXBeginGo 和_fXEndGo 在用户使用波形显示控件的时候并不会显示出来，仅仅用于内部坐标标尺的计算，见下图：

图 4- 2 _fXBeginGo 和_fXEndGo 的意义



其具体表示的值通常在波形显示控件表示范围的外侧，上图中_fXBegin 的值为21.67，而_fXBeginGo 的值为10，_fXEnd 的值为52.96，而_fXEndGo 的值为60。这样设计坐标就能够根据_fXBeginGo 和_fXEndGo 设计出符合用户视觉的坐标线和坐标值，例如上图中显示的坐标值30、40、50，并不是简单地将_fXBegin 和_fXEnd 等分得到的。

4.1.2 坐标标定权值的概念

在确定了双坐标模式后（坐标值和坐标标定值）。用坐标值来描述实际的坐标轴的端点坐标，用坐标标定值来描述整个坐标标尺的坐标线和坐标轴上的子坐标。这样就需要保证坐标标定值要为简单的数据，例如-10、30、0.05等，即保证该浮点值中只有一位是非零的。于是引入了坐标标定权值的概念，定义如下：

```
private float _fXQuanBeginGO; //当前显示波形的 X 轴坐标标定起始权值

private float _fXQuanEndGO; //当前显示波形的 X 轴坐标标定结束权值

private float _fYQuanBeginGO; //当前显示波形的 Y 轴坐标标定起始权值

private float _fYQuanEndGO; //当前显示波形的 Y 轴坐标标定结束权值
```

坐标标定权值表示的是一个浮点数最高非零位对应的权，例如34的坐标标定权值为10，0.036的坐标标定权值为0.01，0的坐标标定权值设定为1。坐标轴刻度线和刻度值通过坐标标定值来实现。

因此，可以设计一个函数，来计算一个浮点数的坐标标定权值：

```
private float _getQuan(float m);
```

以 X 轴举例，_fXBegin 为21.67则计算得到的_fXQuanBeginGO 为10，_fXQuanEndGO 为52.96则计算得到的权值为10。之后就可以通过权值来修改_fXBeginGo 和_fXEndGo，即最后得到的_fXBeginGo 为10，_fXEndGo 为60。因此，在21.67到52.96范围内描绘出的子坐标值为30、40、50。同样的，如果有需要的话，可以方便得将30到40之间的坐标标尺继续分割下去。

4.1.3 坐标标尺的绘制

在绘制坐标标尺前，先判断起始坐标权值和结束坐标权值哪个大，并从权值大的方向往权值小的方向绘制标尺。计算出当前权值下可分多少段坐标，而后转换为实际要描绘的像素位置（在 pictureBoxBottom 和 pictureBoxLeft 中）。

```
if (_fXQuanBeginGO <= _fXQuanEndGO) //X 轴从右往左画

{ //.....

 linesQuan = _fXQuanEndGO; //获得两权的大值

 linesNum = (_fXEndGO - _fXBeginGO) / linesQuan; //可以分成的线段

 pxwidth = //所要画坐标的像素范围

 (float)(width - 100) / (_fXBegin - _fXEnd) * (_fXBeginGO - _fXEndGO);

 pxLine = pxwidth / linesNum; //每段坐标线间隔

 pxGO = //所要画坐标的起点像素位置

 (_fXEndGO - _fXBegin) / (_fXEnd - _fXBegin) * (width - 100) + 50;

//.....
```

4.1.4 子标尺线的选择性显示

根据每段坐标线间隔，可以设计出符合用户视觉的子标尺线，如下图所示：

图 4- 3子标尺线的选择性显示 A

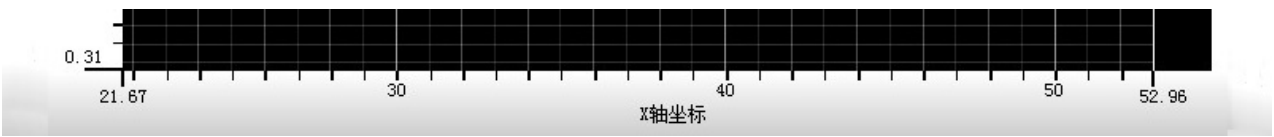
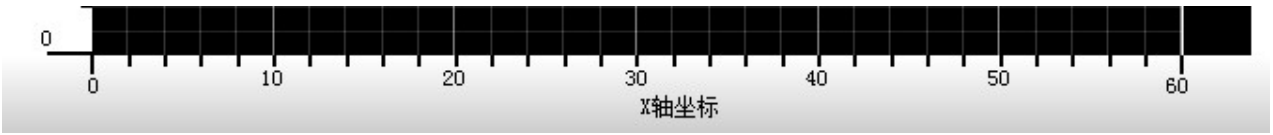


图 4- 4子标尺线的选择性显示 B



当波形显示控件被拉伸到足够大的时候，子标尺线能够选择性的显示。具体设计可以通过使用坐标标尺绘制时定义的每段坐标线间隔 pxLine 来实现。本课题的坐标设计为可以显示三层坐标线，两层坐标值。

当 pxLine 小于等于250时候，画第一层坐标，显示坐标值；

画第二层坐标,第一层坐标的基础上5等分，前提是间距大于10px，若间距大于50px 则显示坐标值。即当 pxLine 大于250，则将 pxLine 五等分，绘制第二层坐标线以及坐标值，当 pxLine 只大于50，则将 pxLine 五等分，绘制第二层坐标线。

继续判断若此时 pxLine 大于50，则将 pxLine 五等分，绘制第三层坐标线，若 pxLine 只大于20，则将 pxLine 二等分，并绘制第三层坐标线。

```
if (pxLine <= 250)

{ currentI = (int)linesNum + 1;

 showTextT = (decimal)_fXEndGO;

 currentDraw = pxGO;

 for (int i = 0; i < currentI; i++)

 { if (currentDraw > 50 && currentDraw < width - 50)

 { Grap.DrawLine(pe, currentDraw, 0, currentDraw, 6); //画坐标线

 Grap.DrawString //画坐标值

 (showTextT.ToString(), fo, brushString, currentDraw, 6, format);

 }

 showTextT -= (decimal)linesQuan; //更新要画的坐标值

 currentDraw -= pxLine; //更新坐标线和坐标值的位置

 }

}

//.....
```

4.2 数据点的描绘

4.2.1 数据值转换为坐标值

在数据存储结构设计的时候，通过两个 List<List<float>>获得的是要显示数据的数据值，而不是最后描绘到波形显示控件上的具体坐标值，因此需要从数据值到坐标值的转换。并将转换后得到的坐标值存储到成员变量中：

```
private List<PointF> _listDrawPoints = new List<PointF>();
```

默认情况下，GDI+是以像素单位来绘图的，初始情况下从控件的左上角沿 X 轴向右增长，沿 Y 轴向下增长，并且左上角第一个像素的坐标为(0,0)。为了计算方便，将坐标系变换为以左下角为零点，从左下角沿 X 轴向右增长，沿 Y 轴向上增长。然后设计一个函数，遍历要画的数据集合，并转换为坐标值：

```
private bool _changeToDrawPoints(

 int index, //要遍历的数据集合的编号

 ref List<PointF> listDrawPoints, //转后后的坐标集合

 int width, //画布像素宽度

 int height, //画布像素高度

 );
```

同时定义了一个私有成员变量，表示坐标精确度：

```
private float _fAccuracy = 0.05f;
```

若坐标起始和结束值之差小于精度范围，则函数返回 false，这样防止出现数据转换的过程中出现数据溢出：

```
if ((_fXEnd - _fXBegin) < _fAccuracy || (_fYEnd - _fYBegin) < _fAccuracy)

 return false;
```

在遍历数据的时候，出现非数字值则跳过：

```
if (float.IsNaN(_listX[index][i]) || float.IsNaN(_listY[index][i]))

 continue;
```

数据值根据当前波形显示控件大小和坐标范围转换为坐标值：

```
currentPointF.X =
```

```
(_listX[index][i] - _fXBegin) * (width - 1) / (_fXEnd - _fXBegin);

currentPointF.Y =

(_listY[index][i] - _fYBegin) * (height - 1) / (_fYEnd - _fYBegin);

listDrawPoints.Add(currentPointF);
```

4.2.2 溢出坐标范围的数据点的处理

当有要绘制的数据值超出的波形显示控件当前坐标的表示范围，若要实现自动调整坐标范围的功能，首先要实现坐标端点的自动改变。因为本课题设计的坐标为双坐标模式，因此可以添加一个函数来实现，根据溢出坐标范围的浮点数，改变 X 轴的坐标标定权值和坐标标定值：

```
private void _changXBegionOrEndGO(

float m, //溢出坐标范围的浮点数

bool isL //是否从左边溢出

);
```

同样的还有 Y 轴对应的函数：

```
private void _changYBegionOrEndGO(float m, bool isL);
```

函数内先通过_getQuan 函数获得溢出点的权，并根据该权值修改对应坐标轴的起点坐标权值和结束坐标权值，同时控制权差在10倍以内，然后根据新的权值修改坐标的标定值。具体设计如下：

```
float quan = _getQuan(m);

if(isL)

{

if (quan < _fXQuanEndGO)

_fXQuanBeginGO = _fXQuanEndGO / 10f;

else if (quan > _fXQuanEndGO)

{ _fXQuanBeginGO = quan;

_fXQuanEndGO = _fXQuanBeginGO / 10f;
```

```
}else _fXQuanBeginGO = _fXQuanEndGO;

if (m <= _fXQuanBeginGO && m >= -_fXQuanBeginGO)

_fXBeginGO = -_fXQuanBeginGO;

else

_fXBeginGO = ((int)(m / _fXQuanBeginGO) - 1) * _fXQuanBeginGO;

}

else .....
```

实际使用过程中，在 pictureBoxGraph 的 OnPaint 消息响应函数中在判断绘图模式为放大模式或坐标自动调整模式的时候，可以调用该函数以实现相应的功能。

4.2.3 遍历所有数据线并绘制出

绘制数据线的工作在 pictureBoxGraph 的 OnPaint 消息中完成。在该函数根据绘图模式调整完坐标范围并更新 pictureBoxBottom 和 pictureBoxLeft 的坐标标尺的显示后，开始遍历所有要显示的数据并绘制出。

每绘制 List<List<float>>中的一条数据集合，都先调用原先设计好的数据值转坐标值函数_changeToDrawPoints，若操作成功，则获取当前要绘制的数据集合的颜色、线宽、连接点样式、起始线帽，并加载到当前画笔中。

```
for (int i = 0; i < _listX.Count; i++)

{ _listDrawPoints.Clear();

if (!_changeToDrawPoints(i, ref _listDrawPoints, width, height))

continue;

pe.Color = _listColor[i]; //装载颜色

pe.Width = _listWidth[i]; //装载宽度

pe.LineJoin = _listLineJoin[i]; //装载连接点

pe.StartCap = _listLineCap[i]; //装载起始线帽
```

然后通过判断绘图样式（线、点、条形图），绘制出图形。注意在绘制线条的时候，设置连接点指定成斜角的连接，这样可以防止线宽在大于1的情况下导致的转折点不精确的问题：

```
if (_listDrawStyle[i] == DrawStyle.Line) //绘制线
```

```
{ f (_listDrawPoints.Count == 1) continue;

 pe.LineJoin = LineJoin.Bevel;

 Grap.DrawLines(pe, _listDrawPoints.ToArray());

}
```

由上面代码可知，实际使用控件时若绘图样式设置为线，则传入的线条连接点参数是不会被使用的。绘制点和条形图如下所示：

```
else if (_listDrawStyle[i] == DrawStyle.dot) //绘制方形点

{ foreach (PointF points in _listDrawPoints)

 { Grap.DrawRectangle(pe, points.X, points.Y, 1, 1); }

}

else

{ foreach (PointF points in _listDrawPoints) //绘制条形线

 { Grap.DrawLine(pe, points.X, points.Y, points.X, 0); }

}
```

4.3 波形显示区域网格的实现

4.3.1 网格相关的成员变量

网格的显示是同显示的坐标标尺的标尺线相对应的，因此在绘制坐标标尺的同时，可以保存绘制标尺的状态，然后在 pictureBoxGraph 中绘制数据点之前，将网格绘制到 pictureBoxGraph 中。因此需要的成员变量如下：

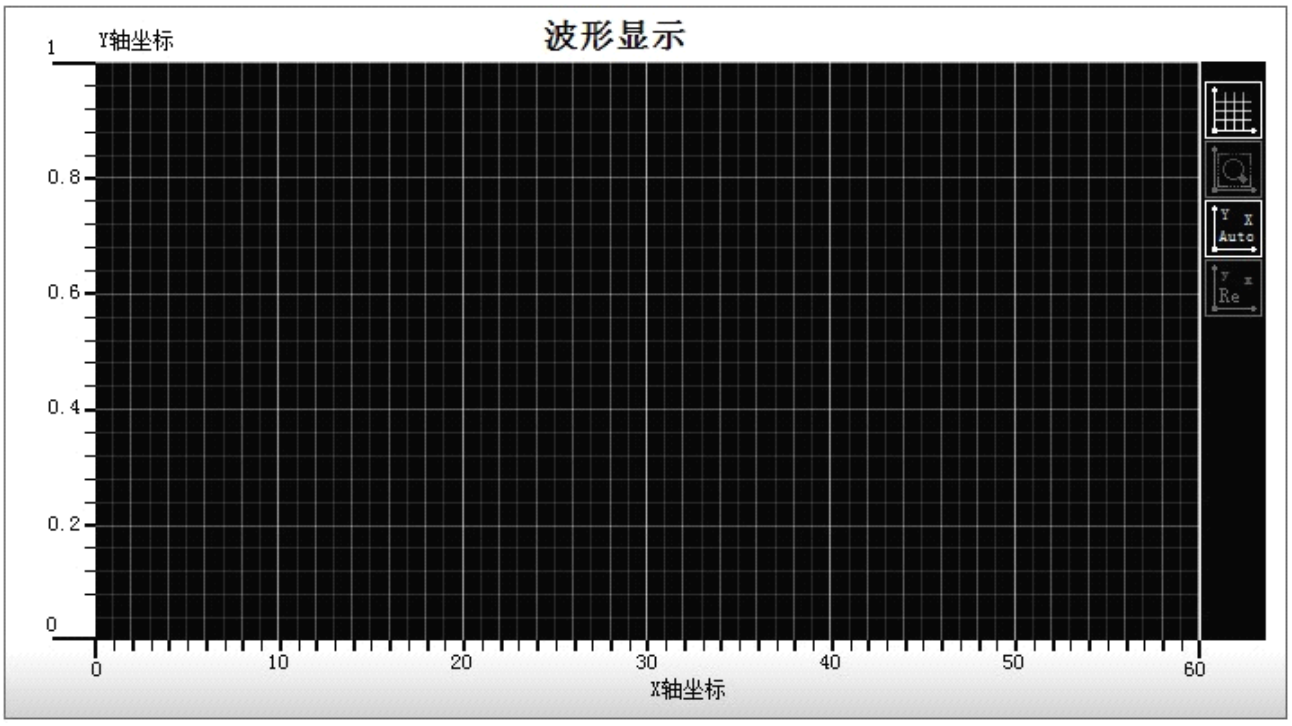
```
private bool _bXLinesLBegin; //X 轴网格线是否从左开始画

private bool _bYLinesLBegin; //Y 轴网格线是否从下开始画

private float _fXpxGO; //所要画 X 轴坐标的起点像素位置

private float _fYpxGO; //所要画 Y 轴坐标的起点像素位置
```

```
private float _fXLinesShowFirst = 0; //X 轴第一层网格线间隔

private float _fXLinesShowSecond = 0; //X 轴第二层网格线间隔

private float _fXLinesShowThird = 0; //X 轴第三层网格线间隔

private float _fYLinesShowFirst = 0; //Y 轴第一层网格线间隔

private float _fYLinesShowSecond = 0; //Y 轴第二层网格线间隔

private float _fYLinesShowThird = 0; //Y 轴第三层网格线间隔
```

当网格线间隔为0的时候，则不画出该层的网格线。为了增强用户体验，将各层的网格线使用不同的透明度绘制出：

```
pe.Color = Color.FromArgb(_iLineShowColorAlpha, _iLineShowColor);

pe.Color = Color.FromArgb(_iLineShowColorAlpha / 2, _iLineShowColor);

pe.Color = Color.FromArgb(_iLineShowColorAlpha / 3, _iLineShowColor);
```

图 4- 5　网格显示效果



4.3.2　网格的绘制

在 pictureBoxGraph 的 OnPaint 更新 pictureBoxLeft 和 pictureBoxBottom 后，开始绘制网格。绘制前判断当然用户是否选择了显示网格，通过私有成员变量_isLinesShowXY 判断，该变量能够通过工具栏按钮和右键菜单改变。

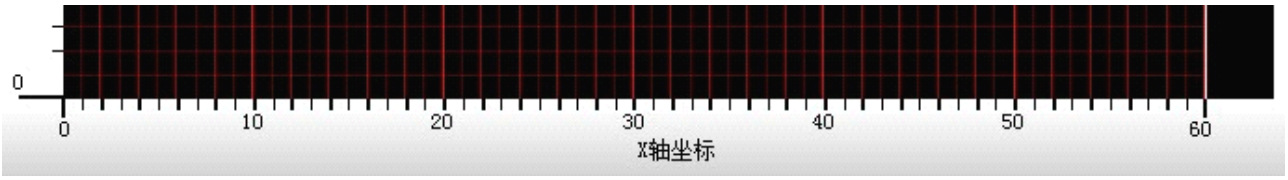
网格线采用1像素的宽度绘制，且从第三层网格线开始绘制。

```
if (_isLinesShowXY)
{ Grap.SmoothingMode = SmoothingMode.None; //保证网格线的清晰
 pe.Width = 1;
 float i = _fXpxGO; //临时，计数
 pe.Color = Color.FromArgb(_iLineShowColorAlpha / 3, _iLineShowColor);
 if (_fXLinesShowThird != 0) //画第三层网格
 { if (_bXLinesLBegin)
 { while (i < width)
 { Grap.DrawLine(pe, i, 0, i, height);
 i = i + _fXLinesShowThird; } }
 else
 { while (i > 0)
 { Grap.DrawLine(pe, i, 0, i, height);
 i = i - _fXLinesShowThird; } }
 }
……
```

两个私有成员变量 iLineShowColorAlpha 和 iLineShowColor 标识网格的透明度和颜色，并通过属性的方式作为接口公开给使用者：

```
public int m_iLineShowColorAlpha //网格线的透明度
{ set { _iLineShowColorAlpha = value; }
 get { return _iLineShowColorAlpha; } }
public Color m_iLineShowColor //网格线的颜色
{ set { _iLineShowColor = value; }
 get { return _iLineShowColor; } }
```

图 4- 6　设置为红色的网格



4.4　工具栏按钮的实现

4.4.1　工具栏按钮相互关系

本课题设计的工具栏按钮包括网格显示按钮、放大选取框功能按钮、坐标自动调整按钮、默认坐标范围按钮。

显示网格按钮同其他三个按钮不存在功能上的冲突，因此可以独立考虑。

放大选取框功能按钮默认处于未开启状态，若开启放大选取框功能，并成功的进行了局部放大操作，则自动关闭坐标自动调整功能。

坐标自动调整按钮默认处于开启状态，若开启坐标自动调整功能，则自动关闭放大选取框功能。

默认坐标范围按钮默认处于未开启状态，当用户点击默认坐标范围按钮后，自动将坐标范围调整为波形显示控件初始的坐标范围，并关闭坐标自动调整功能。

4.4.2　工具栏提示标签的实现

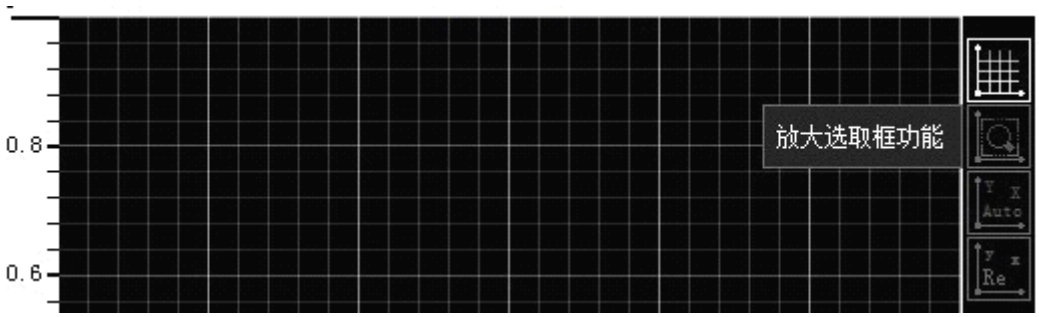
工具栏提示标签 labelItemShuoMing 默认情况下是处于隐藏状态的，只有当用户鼠标经过某一按钮的时候，按钮的 MouseEnter 消息被调用，并修改提示标签的位置和标签的内容，并设置 Visible 为 true：

```
private void buttonLinesShowXY_MouseEnter(object sender, EventArgs e)

{ Point po = new Point();

 po.X = panelControlItem.Location.X - 100;

 po.Y = panelControlItem.Location.Y + buttonLinesShowXY.Location.Y;

 labelItemShuoMing.Location = po; //更新标签说明坐标

 labelItemShuoMing.Text = "网格显示"; //更新标签说明文字

 labelItemShuoMing.Visible = true; //显示标签说明

}
```

并在鼠标离开按钮的时候重新设置 Visible 为 false。

具体效果如下图所示：

图 4- 7　工具栏提示标签

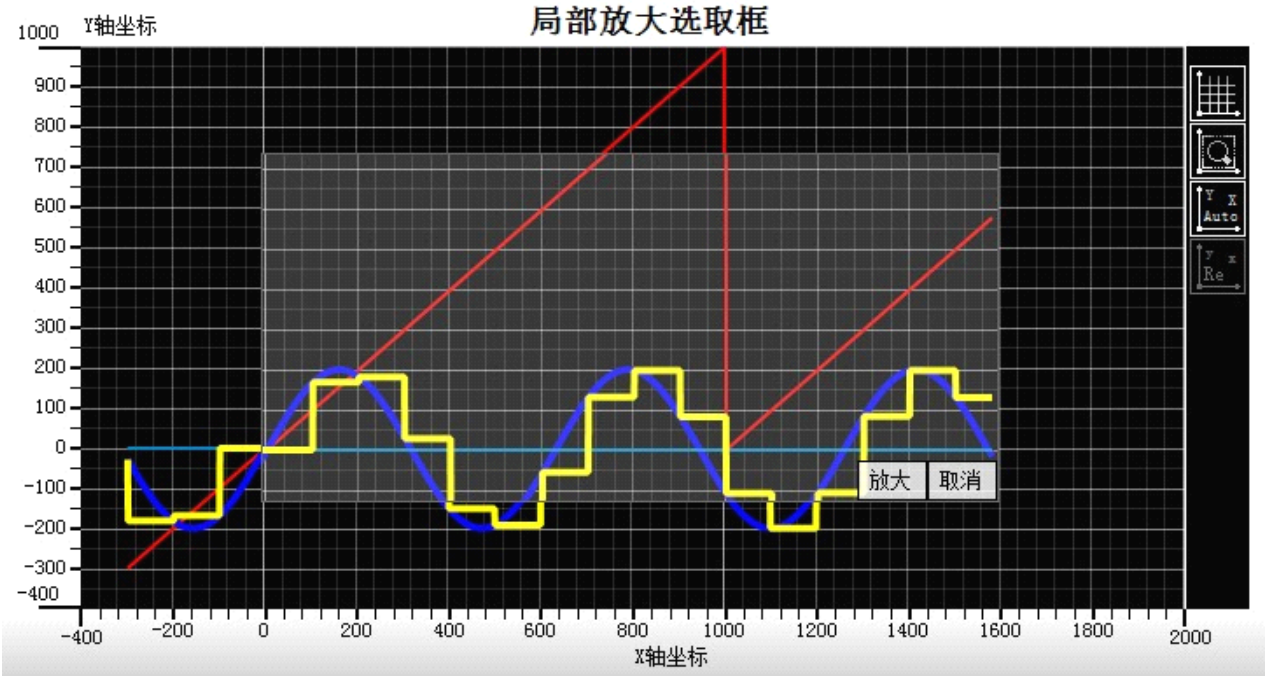


4.5　波形放大功能的实现

4.5.1　局部放大选择框的实现

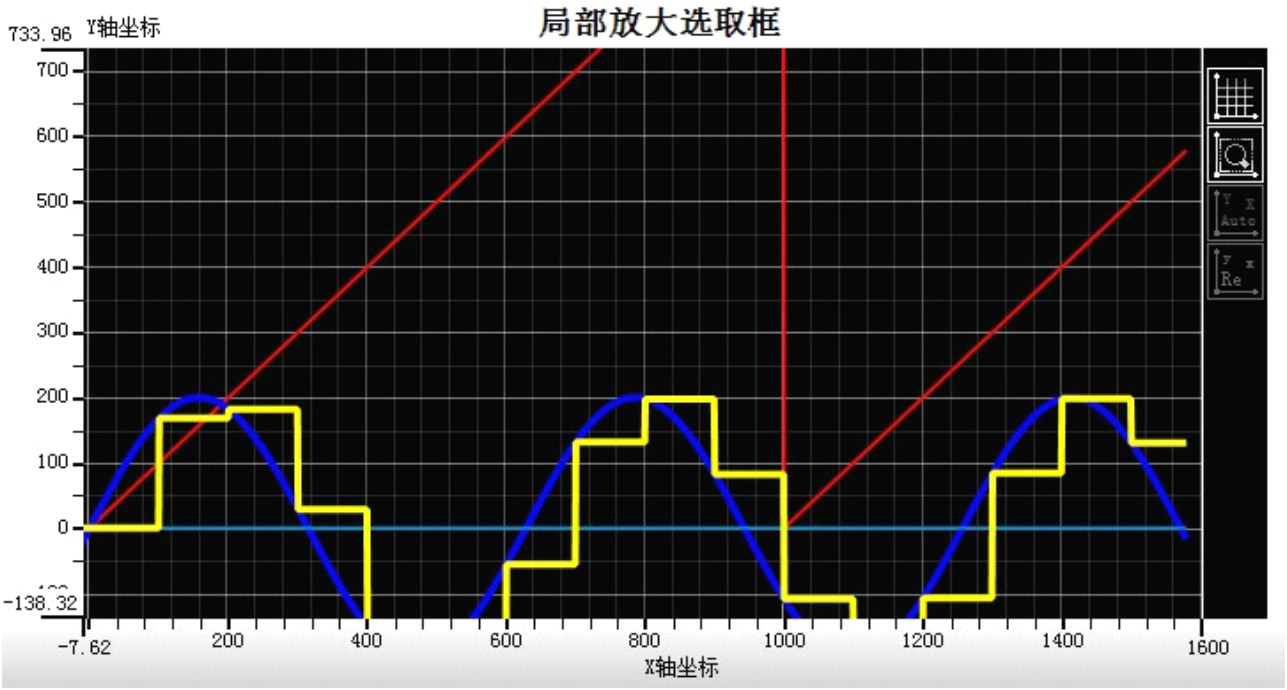
局部放大选取框通过一个半透明的 pictureBox 控件，作为选取框的背景，命名为 pictureBoxBigXY 并在选取框的右下角放置一个 Label，容纳放大和取消按钮，两个按钮的名称分别为 buttonBigXYBig 和 buttonBigXYQuit。最后运行的实际效果如下图所示：

图 4- 8　局部放大选择框



用户在点击放大按钮后，实现的效果如下：

图 4- 9　点击放大按钮后的效果



为了控制 pictureBoxBigXY 的显示，需要定义五个私有成员变量：

```
private Point _pictureBoxBigXY_L; //存放波形放大方框的起点坐标

private Point _pictureBoxBigXY_R; //存放鼠标移动时的坐标

private Point _pictureBoxBigXY_M; //存放最后调整后波形放大框的位置坐标

private float _labelXYNumX; //存放坐标显示 Label 的 X 值

private float _labelXYNumY; //存放坐标显示 Label 的 Y 值
```

需要响应的消息如下表所示：

表格 4- 1局部放大响应的消息

消息所属	消息类型	作用
pictureBoxGraph	鼠标按下	设置 pictureBoxBigXY 的父容器并更新其位置
pictureBoxGraph	鼠标移动	更新 pictureBoxBigXY 的大小并显示
pictureBoxGraph	鼠标抬起	判断并显示放大和取消两个按钮
buttonBigXYBig	单击	根据当前 pictureBoxBigXY 范围进行放大操作
buttonBigXYQuit	单击	隐藏 pictureBoxBigXY

4.5.2 放大选择框的鼠标操作

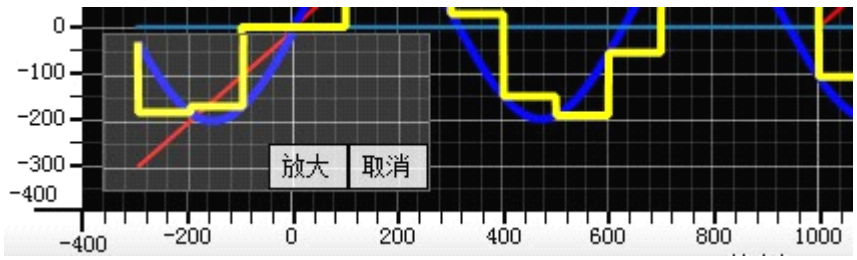
在 pictureBoxGraph 中按下鼠标左键，则根据_isShowBigSmallModeXY 标记判断是否进行放大选择框的初始化操作。修改鼠标的光标为十字光标，若先前已经显示过放大选择框，则设置为隐藏，并设置 pictureBoxGraph 作为 buttonBigXYBig 的父容器，这样方便坐标到数据值的转换：

```
private void pictureBoxGraph_MouseDown(object sender, MouseEventArgs e)

{ if (_isShowBigSmallModeXY && e.Button == MouseButtons.Left)

 { pictureBoxGraph.Cursor = Cursors.Cross; //十字光标

 pictureBoxBigXY.Visible = false; //隐藏波形放大框

 panelBigXY.Visible = false; //隐藏隐藏波形放大操作框

 pictureBoxBigXY.Parent = pictureBoxGraph; //父容器

 panelBigXY.Parent = pictureBoxBigXY; //父容器

 _pictureBoxBigXY_L.X = e.Location.X;

 _pictureBoxBigXY_L.Y = e.Location.Y;

 pictureBoxBigXY.Location = _pictureBoxBigXY_L; //更新位置

 }

 ……

}
```

左键按下后用户通过移动鼠标来选取需要放大的区域。因此必须在鼠标移动过程中不断获得当前鼠标的位置，并根据鼠标按下的位置和当前鼠标的位置，描绘出最后放大选取框的位置和大小。这里需要考虑到用户操作鼠标的方向可能是向右下角拖动，也可能向左上角拖动，因此鼠标按下时获得的坐标，并非完全代表 pictureBoxBigXY 的位置，需要通过重新计算获得。

这里还需要考虑一种情况：当用户按下鼠标左键并拖动鼠标到控件外边的时候，就需要保证 pictureBoxBigXY 始终显示在 pictureBoxGraph 中：

图 4- 10 放大选择框无法继续往左下角放大



```
if (e.Location.X > pictureBoxGraph.Width - 10)

{ _pictureBoxBigXY_R.X = pictureBoxGraph.Width - 10; }

else if (e.Location.X < 10) { _pictureBoxBigXY_R.X = 10; }

else { _pictureBoxBigXY_R.X = e.Location.X; }
```

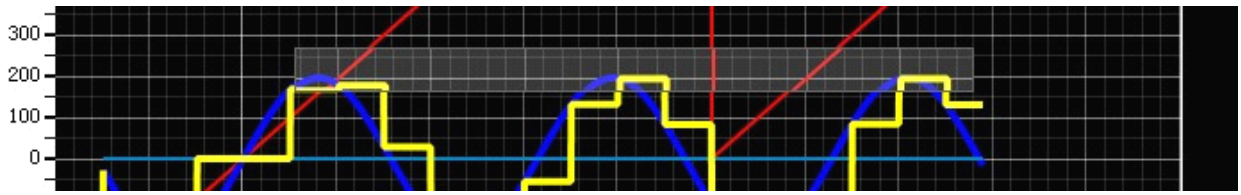
上面演示的代码是防止横坐标越界，同样的，纵坐标也要进行验证。之后通过判断设置 pictureBoxBigXY 新的位置：

```
_pictureBoxBigXY_M.X = (_pictureBoxBigXY_L.X < _pictureBoxBigXY_R.X) ?
_pictureBoxBigXY_L.X : _pictureBoxBigXY_R.X;

_pictureBoxBigXY_M.Y = (_pictureBoxBigXY_L.Y < _pictureBoxBigXY_R.Y) ?
_pictureBoxBigXY_L.Y : _pictureBoxBigXY_R.Y;

pictureBoxBigXY.Location = _pictureBoxBigXY_M;
```

pictureBoxBigXY 的大小则通过调用 Math.Abs 计算得到。

鼠标抬起的时候需判断当前的波形放大选择框是否足够大小显示放大和取消两个按钮，若不够显示则不显示，反之则调整位置并显示。

图 4- 11 放大选取框不够显示放大按钮的情况



4.5.3 放大选择框的按钮操作

放大选择框有放大和取消两个按钮。其中放大按钮的单击事件中，转换 pictureBoxBigXY 的四个角的坐标值为实际值，并调用 Refresh 更新显示。

函数中需要设置放大的范围，防止导致数据溢出的错误：

```
if (_fXEnd - _fXBegin < 1.0f || _fYEnd - _fYBegin < 1.0f){ return; }
```

设计一个函数，将波形显示中矩形区域的坐标转换为数据值，这里获得的坐标值是以左上角为原点的，而要转换到的数据值是从左下角向右 X 轴正向增长，向上 Y 轴正向增长，所以转换的时候需要特别注意，并且转换的过程中需要用临时变量来保存转换后的值，防止先转换完成的坐标值对下一个转换产生影响：

```
private void _changeXYPointsToNum(float xB, float xE, float yB, float yE,

 ref float outxB, ref float outxE, ref float outyB, ref float outyE)

{ float currentB, currentE;
```

```
currentB=xB/(pictureBoxGraph.Width-1)*(_fXEnd-_fXBegin) +_fXBegin;

currentE=xE/(pictureBoxGraph.Width - 1)*(_fXEnd-_fXBegin)+_fXBegin;

outxB = currentB; outxE = currentE;

currentB=_fYEnd-yB/(pictureBoxGraph.Height-1)*(_fYEnd-_fYBegin);

currentE=_fYEnd-yE/(pictureBoxGraph.Height-1)*(_fYEnd-_fYBegin);

outyE = currentB; outyB = currentE;

}
```

在调用该函数的时候传入的参数即为_fXBegin、_fXEnd、_fYBegin、_fYEnd。即通过调用该函数，可以改变 X 轴和 Y 轴的坐标值，在 pictureBoxGraph 的 OnPaint 消息函数中，会根据上面的四个值改变坐标的标定值，从而正确的实现放大效果。

4.5.4 更新数据显示范围为放大的范围

放大模式下坐标不自动调整，使用当前坐标范围修改标定权值和标定坐标范围，在 pictureBoxGraph 的 OnPaint 消息函数中：

```
_fXQuanBeginGO = _getQuan(_fXBegin);_fXQuanEndGO = _getQuan(_fXEnd);

_fYQuanBeginGO = _getQuan(_fYBegin);_fYQuanEndGO = _getQuan(_fYEnd);

_changXBegionOrEndGO(_fXBegin, true);

_changXBegionOrEndGO(_fXEnd, false);

_changYBegionOrEndGO(_fYBegin, true);

_changYBegionOrEndGO(_fYEnd, false);
```

4.6 坐标自动调整及恢复默认坐标的实现

4.6.1 坐标自动调整功能

在工具栏按钮 buttonAutoModeXY 的单击事件中，重新初始化坐标值和坐标标定值，使用事先保存好的默认的坐标范

围：

```
private float _fXBeginSYS = 0f; //X 轴起始坐标值

private float _fXEndSYS = 60f; //X 轴结束坐标值

private float _fYBeginSYS = 0f; //Y 轴起始坐标值

private float _fYEndSYS = 1f; //Y 轴结束坐标值
```

同时，需要标记_isBigModeXY 为 false。另外需要注意的是，之前的局部放大操作若达到放大的极限（横坐标首尾之差小于1.0或纵坐标首尾之差小于1.0），则会禁用放大按钮。因此需要在 buttonAutoModeXY 的单击事件中重新设置 buttonBigXYBig 的 Enabled 属性为 true。

上面的步骤实现了先恢复默认坐标的功能，然后再在 pictureBoxGraph 的 OnPaint 事件中，通过遍历当前要绘制的数据，调用_changXBegionOrEndGO 完成坐标标定值和坐标标定权值的更新。之后刷新 pictureBoxLeft 和 pictureBoxBottom 更新坐标标尺的显示，即实现了坐标自动调整的功能。

4.6.2 恢复默认坐标范围功能

恢复默认坐标范围实际上少了坐标自动调整在 pictureBoxGraph 中的操作。即直接更新坐标值和坐标标定值、以及坐标标定权值：

```
_fXBegin = _fXBeginGO = _fXBeginSYS;

_fYBegin = _fYBeginGO = _fYBeginSYS;

_fXEnd = _fXEndGO = _fXEndSYS; _fYEnd = _fYEndGO = _fYEndSYS;

_fXQuanBeginGO = _getQuan(_fXBeginGO);

_fXQuanEndGO = _getQuan(_fXEndGO);

_fYQuanBeginGO = _getQuan(_fYBeginGO);

_fYQuanEndGO = _getQuan(_fYEndGO);

pictureBoxGraph.Refresh(); //刷新界面

panelItemsIN.Refresh(); //刷新按钮显示
```

4.7 波形显示控件接口的实现

4.7.1 控件基本属性

控件的标题、X 轴名称、Y 轴名称、以及坐标的初始值通过属性的方式公开给用户：

表格 4- 2 控件基本属性

名称	数据类型	作用
m_SyStitle	string	波形显示控件标题
m_SySnameX	string	X 轴名称
m_SySnameY	string	Y 轴名称
m_fXBeginSYS	float	初始 X 轴起始坐标
m_fXEndSYS	float	初始 X 轴结束坐标
m_fYBeginSYS	float	初始 Y 轴起始坐标
m_fYEndSYS	float	初始 Y 轴结束坐标

在控件内部，控件的标题是在 pictureBoxTop 的 OnPaint 中绘制出的：

```
private void pictureBoxTop_Paint(object sender, PaintEventArgs e)

{ ……

 brush.Color = _titleBorderColor;

 Grap.DrawString(_SyStitle, foTitle, brush, pictureBoxTop.Width * _titlePosition+1, height+1,
format);

 Grap.DrawString(_SyStitle, foTitle, brush, pictureBoxTop.Width * _titlePosition-1, height-1,
format);

 brush.Color = _titleColor;

 Grap.DrawString(_SyStitle, foTitle, brush, pictureBoxTop.Width * _titlePosition, height,
format);

 ……

}
```

可得实际上控件的标题包括两层，分别通过_titleBorderColor 和_titleColor 绘制出，并且控件的通过_titlePosition 控制其在水平方向上绘制的位置。同样的，X 轴的标签在 pictureBoxBottom 的 OnPaint 中绘制出，Y 轴标签则是在 pictureBoxTop 的 OnPaint 中绘制出。

4.7.2 控件外观样式

控件的外观样式一共涉及到21个属性，具体如下标所示：

表格 4- 3 控件外观样式

名称	作用	名称	作用
m_titleSize	控件标题字体大小	m_GraphBackColor	波形显示区域背景色
m_titlePosition	控件标题位置	m_ControlItemBackColor	工具栏背景色
m_titleColor	控件标题颜色	m_ControlButtonBackColor	工具栏按钮背景颜色
m_titleBorderColor	控件标题描边颜色	m_ControlButtonForeColorL	工具栏按钮前景选中颜色
m_backColorL	背景色渐进起始颜色	m_ControlButtonForeColorH	工具栏按钮前景未选中颜色
m_backColorH	背景色渐进终止颜色	m_DirectionBackColor	标签说明框背景颜色
m_coordinateLineColor	坐标线颜色	m_DirectionForeColor	标签说明框文字颜色
m_coordinateStringColor	坐标值颜色	m_BigXYBackColor	放大选取框背景颜色
m_coordinateStringTitleColor	坐标标题颜色	m_BigXYButtonBackColor	放大选取框按钮背景颜色
m_iLineShowColorAlpha	网格线的透明度	m_BigXYButtonForeColor	放大选取框按钮文字颜色
m_iLineShowColor	网格线的颜色		

m_titleSize 为 int 类型、m_titlePosition 为 float 类型、m_iLineShowColorAlpha 为 ing 类型，其他的外观样式属性都为 Color 类型。其中某些属性是对应了多个私有成员变量的，这样可以统一的进行颜色的更新：

```
public Color m_ControlItemBackColor

{ set
```

```
{ ControlItemBackColor = value;

panelControlItem.BackColor = ControlItemBackColor;

buttonItemsDown.ForeColor = ControlItemBackColor;

buttonControlItemUP.ForeColor = ControlItemBackColor;

}

get{ return ControlItemBackColor; }

}
```

4.7.3 控件绘图接口

本课题设计的波形显示控件主要提供了五个函数供使用者调用。用来控制波形显示控件的绘图操作。

函数 f_ClearAllPix 用于清空所有加载的波形数据，具体设计如下：

```
public void f_ClearAllPix()

{ _listX.Clear(); _listY.Clear(); _listColor.Clear(); _listWidth.Clear();

_listLineJoin.Clear(); _listLineCap.Clear(); _listDrawStyle.Clear();

pictureBoxGraph.Refresh(); //更新

}
```

函数 f_reXY 用于重新初始化 X 轴和 Y 轴坐标，内部通过调用工具栏按钮的恢复默认坐标范围按钮和设置自动调整坐标按钮来实现，具体设计如下：

```
public void f_reXY()

{ buttonReXY_Click(null, null);

buttonAutoModeXY_Click(null, null);

}
```

函数 f_Refresh 用来更新显示，实际上再每次需要重新绘图的时候，只需要更新 pictureBoxGraph 即可，因为

pictureBoxGraph 的 OnPaint 函数中会调用 pictureBoxBottom 和 pictureBoxLeft 的更新显示，因此设计如下：

```
public void f_Refresh()

{ pictureBoxGraph.Refresh(); }
```

函数 f_LoadOnePix 用于清空原有数据并加载一条波形数据：

```
public void f_LoadOnePix(ref List<float> listX, ref List<float> listY, Color listColor, int listWidth, LineJoin listLineJoin, LineCap listLineCap,DrawStyle listDrawStyle)

{ f_ClearAllPix(); //重新初始化

 _listX.Add(listX); _listY.Add(listY);

 _listColor.Add(listColor); _listWidth.Add(listWidth);

 _listLineJoin.Add(listLineJoin); _listLineCap.Add(listLineCap);

 _listDrawStyle.Add(listDrawStyle);

}
```

函数 f_AddPix 则是在原有波形上添加一条波形数据，其实现的过程中少了清空原有波形数据这一步，因此数据是在原有数据的基础上继续添加上去的。

4.8 波形显示控件其他细节的处理

4.8.1 坐标值产生遮盖时的处理

在坐标标尺的调整过程中，会出现子坐标同两端的坐标重叠的现象，因此为了能够更友好的显示重叠部分的坐标，可以在绘制坐标轴两端坐标的时候，先填充一个背景颜色，用户遮盖可能和两端坐标重叠的子坐标：

```
Grap.FillRectangle(brush, 28, 8, 46, 16); //坐标背景色，用于遮盖

Grap.FillRectangle(brush, width - 63, 8, 50, 16); //坐标背景色，用于遮盖

Grap.DrawLine(pe, 50, 0, 50, 10); //X 轴起始坐标线条

Grap.DrawString(string.Format("{0}", Math.Round(_fXBegin, _iAccuracy)), fo, brushString, 51, 10, format); //X 轴起始坐标值
```

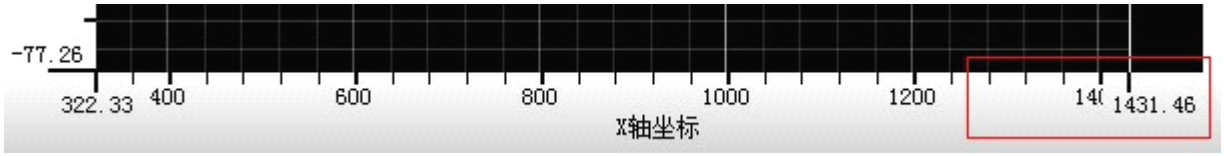
```
……
```

注意上面通过_iAccuracy 来控制两端坐标值能够显示的小数点位数，防止浮点数显示过多小数位数出现的重叠问题，具体定义如下：

```
private int _iAccuracy = 2; //坐标显示最多小数位数
```

最后实现的效果如下：

图 4- 12　坐标遮盖的问题



上图红色方框部分，X 轴的末尾坐标值1431.46和子坐标值1400重叠，通过遮盖子坐标，能够使用户更有效地观察坐标值，同样的，Y 轴的两端坐标和子坐标的显示也做类似的处理。

4.8.2　波形显示控件大小改变时的处理

实际使用中，若用户打开了放大局部放大功能，并在波形显示区域拖出了一个矩形，若此时用户在未点击放大和取消按钮之前，改变了波形显示控件的大小（或者说是使用该控件的软件界面的大小），就会导致之后的实际放大区域和用户期望的不相符合，因此，需要在控件大小改变时隐藏放大选取框：

```
private void ZGraph_Resize(object sender, EventArgs e)

{ pictureBoxBigXY.Visible = false; //隐藏放大选择框

 panelBigXY.Visible = false; //隐藏放大取消按钮

 _isShowBigSmallModeXY = false; //标记为隐藏

 ……
```

4.8.3　按钮点击时进行禁用操作

防止按钮再处理的过程中被再次点击，因此在按钮处理函数中进行按钮的禁用操作，另外重新设置按钮的焦点：

```
private void buttonBigModeXY_Click(object sender, EventArgs e)

{ buttonBigModeXY.Enabled = false; //禁用按钮

 buttonBigModeXY.Parent.Focus(); //取消焦点
```

```
……

buttonBigModeXY.Enabled = true; //启用按钮

……
```

4.8.4 右键菜单的显示

在使用过程中，用户能通过右键菜单实现工具栏中提供的功能。同时，右键菜单中还显示当然鼠标位置的数据值，在 pictureBoxGraph 的右键按下时获取：

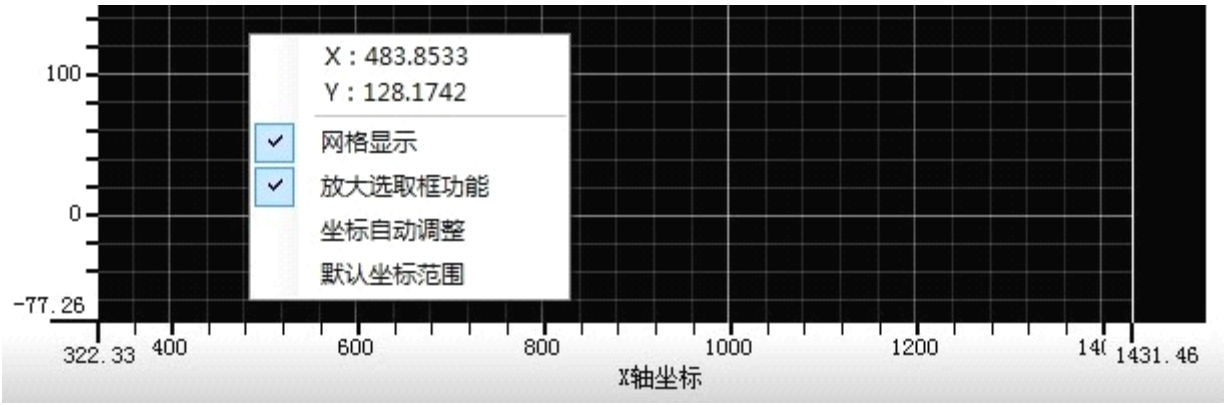
```
if (e.Button == MouseButtons.Right)

{ _changeXYPointsToNum(e.Location.X, e.Location.Y, ref _labelXYNumX, ref _labelXYNumY);

 toolStripTextBoxX.Text = //显示当前点的横坐标值

 string.Format(" X: {0}", Math.Round(_labelXYNumX, _iAccuracy + 2));

 toolStripTextBoxY.Text = //显示当前点的纵坐标值

 string.Format(" Y: {0}", Math.Round(_labelXYNumY, _iAccuracy + 2));

}
```

其中的_changeXYPointsToNum 函数将波形显示中一个点的坐标转换为数据值，具体实现如下：

```
private void _changeXYPointsToNum(float x,float y,ref float outX,ref float outY)

{ outX = x / (pictureBoxGraph.Width - 1) * (_fXEnd - _fXBegin) + _fXBegin;

 outY = _fYEnd - y / (pictureBoxGraph.Height - 1) * (_fYEnd - _fYBegin);}
```

右键菜单的效果如下图：

图 4- 13 右键菜单



4.8.5 XML 注释以及智能提示

Visual Studio 中能够为代码生成 XML 注释，并能利用该注释产生智能提示的功能，这样能够方便开发人员以及控件的使用者快速获得函数以及属性的信息。

图 4- 14 XML 风格的注释

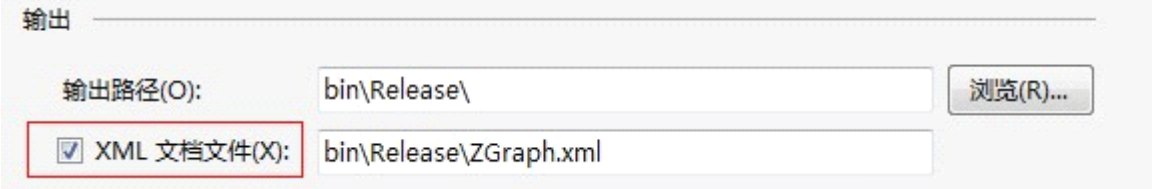
```
/// <summary>
/// 清空原有数据并加载一条波形数据，显示样式为线条
/// </summary>
/// <param name="listX">X轴</param>
/// <param name="listY">Y轴</param>
/// <param name="listColor">线条颜色</param>
/// <param name="listWidth">线条宽度</param>
public void f_LoadOnePix(ref List<float> listX, ref List<float> listY, Color listColor, int listWidth)
```

拥有 XML 风格注释的函数或属性，在使用的时候能够产生智能提示的效果：

图 4- 15 智能提示

对控件中需要公开的属性和方法都加上 XML 风格的注释后，要想控件在提供给控件使用者时，任然能产生智能提示的话，需要将 XML 文档到处，并在使用最终生成的 DLL 控件时，将该导出的 XML 文档放置在同一目录下，这样就能实现智能提示的功能了。具体需要在项目属性的生成选项卡中设置：

图 4- 16 XML 文档输出



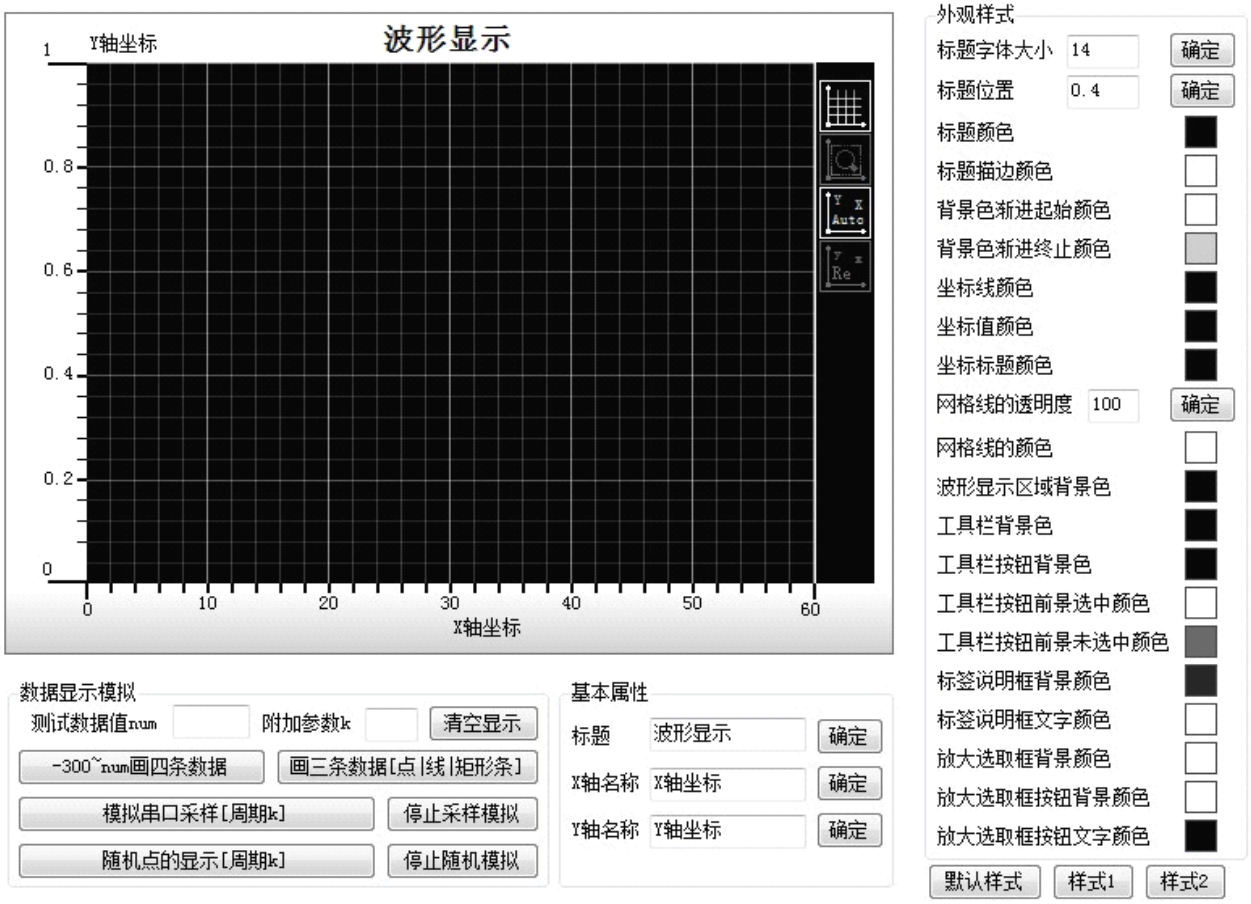
这样使用者就能够方便快捷地得知该控件的方法和属性的含义。

5 波形显示控件功能的演示和使用

5.1 波形显示控件演示程序的设计

为了配合波形显示控件，本课题还设计了一个波形显示控件的演示程序。设计最终界面如下图所示：

图 5- 1 波形显示控件演示程序



其中右侧一栏主要是对波形显示控件外观样式的控制。可以改变某一属性的参数，也可以点击下方的默认样式、样式1和样式2。其中样式1和样式2是在演示程序内部通过调用波形显示控件的各个接口属性来做相应的改变的。默认样式则是在演示控件初始化之后，将此时控件的各个属性保存在成员变量中，之后可以通过点击该按钮从新将成员变量中的参数值赋值给波形显示控件对应的属性。

左下方提供了简单的数据显示模拟和基本属性更改的操作。该演示程序主要模拟了多条曲线的绘制，以及模拟自动化检测软件中串口采样的过程，即实现波形的动态显示，随机点的显示实现过程同串口采样模拟类似。

5.2 波形显示控件功能的演示

5.2.1 外观样式的更改

用户可以通过对右边属性值的更改改变波形显示控件的外观。通过点击对应的属性的彩色框，可以更改颜色：

图 5- 2 改变控件的标题颜色

用户也可以点击下发的样式1和样式2，其在按钮的单击消息函数中修改了波形显示控件的样式：

图 5- 3 演示程序样式1

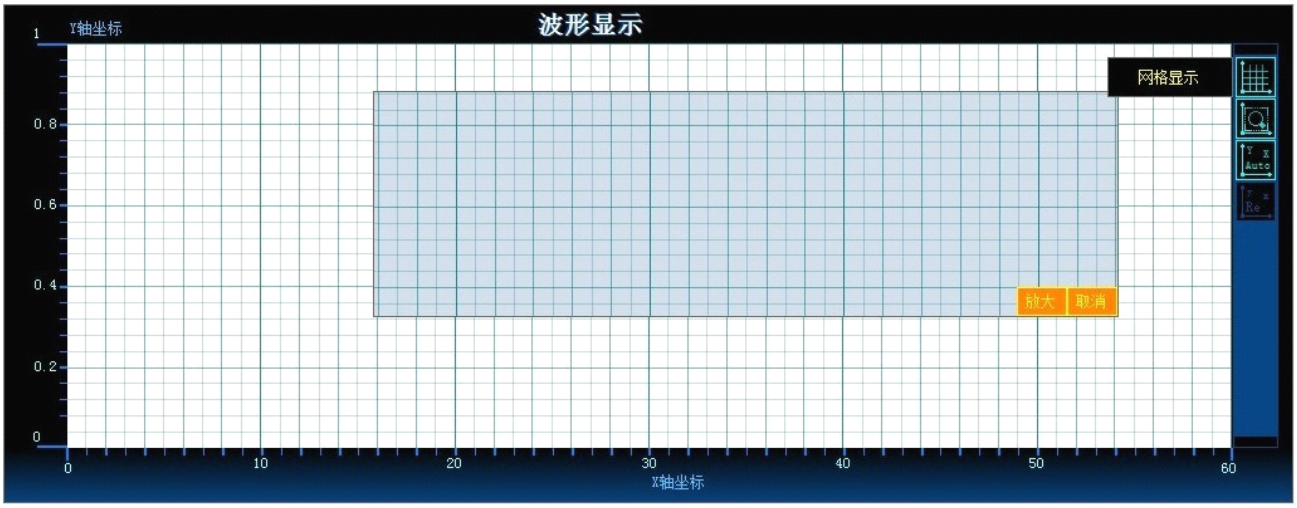
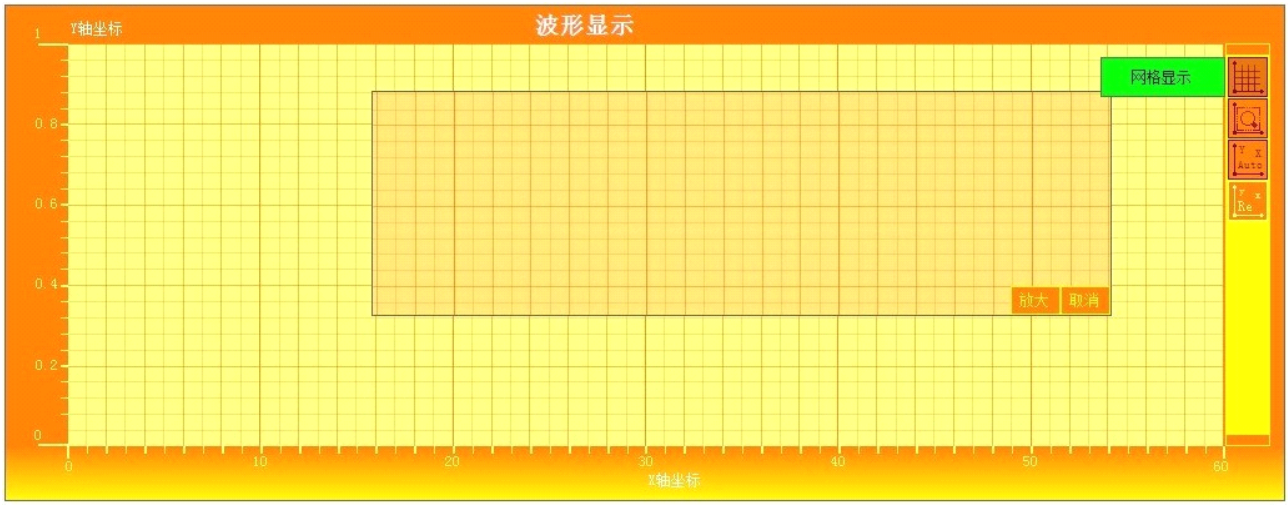


图 5- 4 演示程序样式2



用户可以根据自己喜欢的颜色，逐一设置符合软件整体风格的样式。

5.2.2 波形显示演示

（1） 绘制多条曲线

演示程序内部定义了8个 List<float>，即可以表示4条不同的数据集合，从 x1、y1命名到 x4、y4。可通过修改该4条数据集合的数据实现不同的演示效果。

点击左下角的-300~num 画四条数据按钮，可以在控件中绘制演示程序内设置的数据曲线，可以通过更改测试数据 值 num 来测试波形显示控件的显示效果。其具体内部实现通过遍历-300~num，一次添加数据值：

```
x1.Clear(); y1.Clear(); x2.Clear(); y2.Clear();

x3.Clear(); y3.Clear(); x4.Clear(); y4.Clear();

for (int i = -300; i < num; i++) //这里省略 num 小于-300的情况

{ x1.Add(i); y1.Add(i % 1000);

 x2.Add(i); y2.Add((float)Math.Sin(i / 100f) * 200);

 x3.Add(i); y3.Add(0);

 x4.Add(i); y4.Add((float)Math.Sin(i / 100) * 200);

}
```

然后调用波形显示控件的绘图函数绘制出：

```
zGraphTest.f_reXY(); //恢复默认坐标范围

zGraphTest.ff_LoadOnePix(ref x1, ref y1, Color.Red, 2);

zGraphTest.f_AddPix(ref x2, ref y2, Color.Blue, 4);

zGraphTest.f_AddPix(ref x3, ref y3, Color.FromArgb(0, 128, 192), 2);

zGraphTest.f_AddPix(ref x4, ref y4, Color.Yellow, 4);

zGraphTest.f_Refresh(); //更新并显示
```

注意在调用绘图函数绘制相应的曲线后，波形并不会马上显示出来，因此在4条曲线都添加完毕后，调用波形显示控件的 f_Refresh 更新显示。显示效果如下图所示：

图 5- 5　波形显示演示绘制多条曲线（num 为1580）

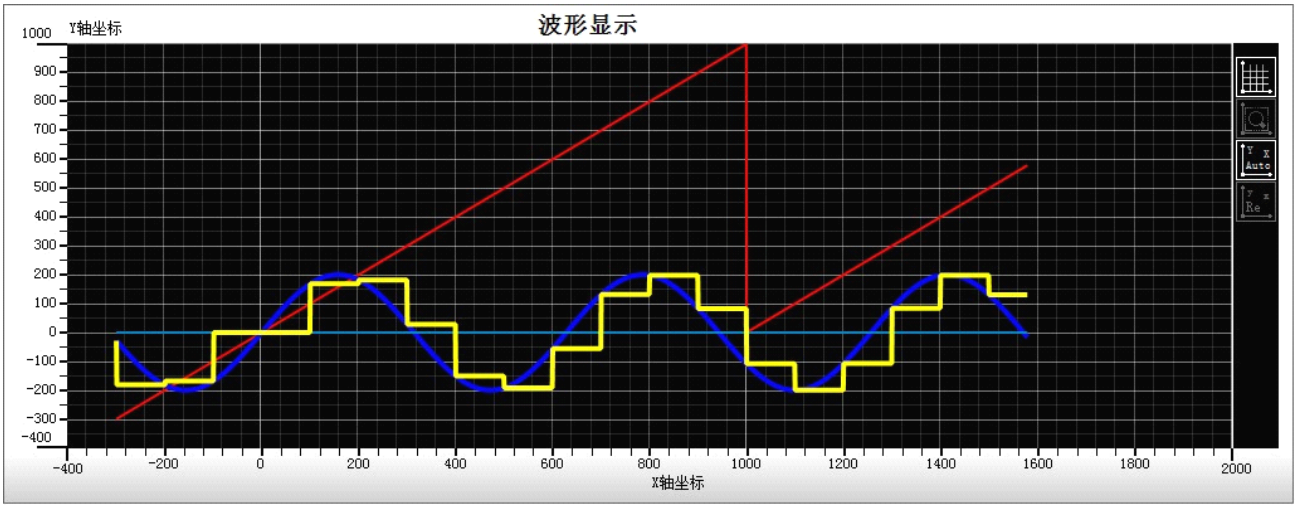


图 5- 6　波形显示控件演示绘制多条曲线（num 为-10000）

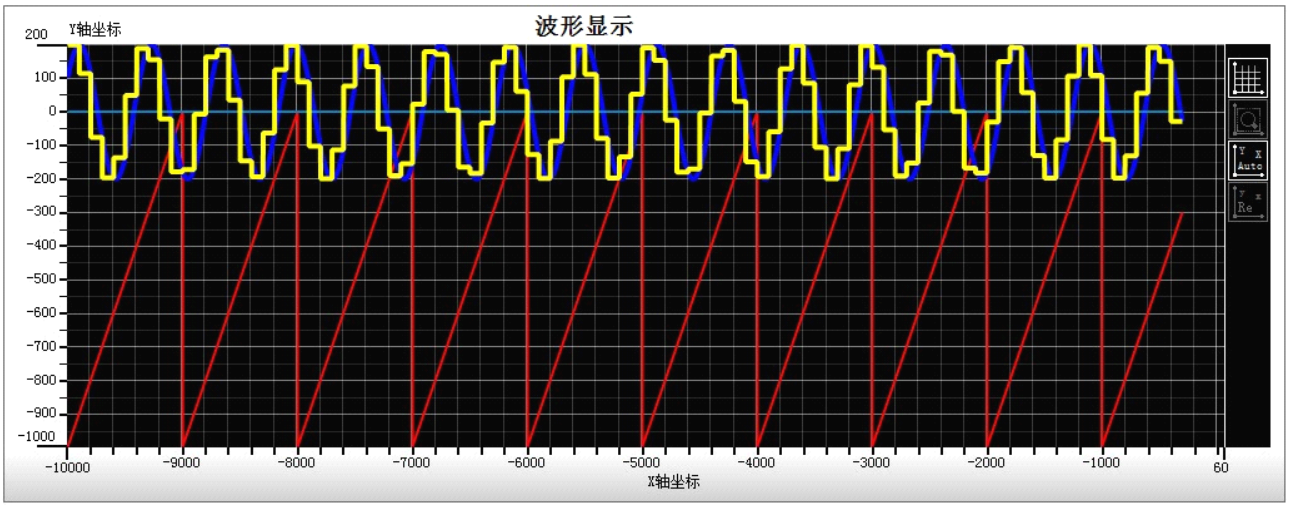
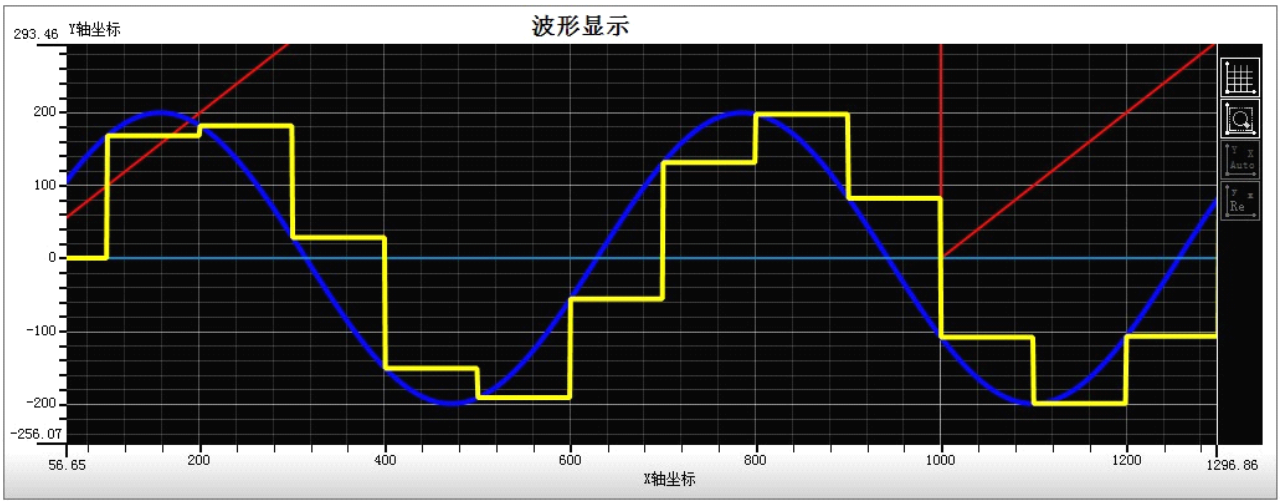


图 5- 7　波形显示控件绘制多条曲线（num 为1580时局部放大的效果）



（2）　绘图样式的选择

控件还提供了绘制样式的选择：点、线、矩形条。演示程序实现如下：

```
for (int i = 0; i < 18000; i += 1000)
```

```
{ x1.Add(i); y1.Add(i/4f);

 x2.Add(i); y2.Add(i/4f);

 x3.Add(i); y3.Add(i / 8f);

}

zGraphTest.f_LoadOnePix(ref x1, ref y1, Color.Red, 3); //绘制线

zGraphTest.f_AddPix(ref x2, ref y2, Color.Yellow, 5, LineJoin.Round,

 LineCap.Flat, ZhengJuyin.UI.ZGraph.DrawStyle.dot); //绘制点

zGraphTest.f_AddPix(ref x3, ref y3, Color.FromArgb(0,128,192), 12,

 LineJoin.MiterClipped, LineCap.NoAnchor,

 ZhengJuyin.UI.ZGraph.DrawStyle.bar); //绘制矩形条
```

绘图演示演示程序显示如下效果：

图 5- 8 波形显示控件绘制样式演示

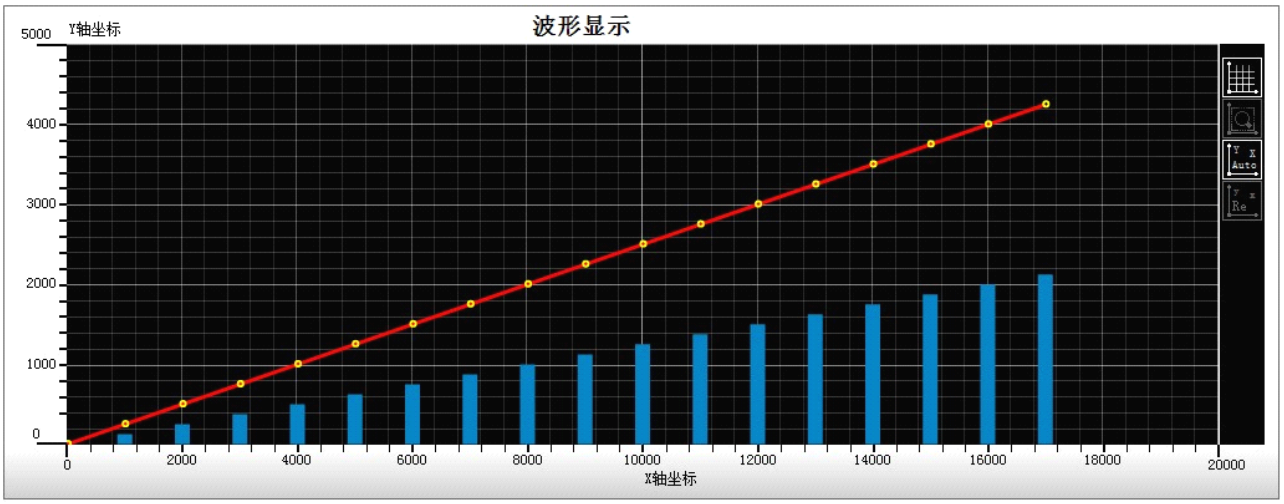
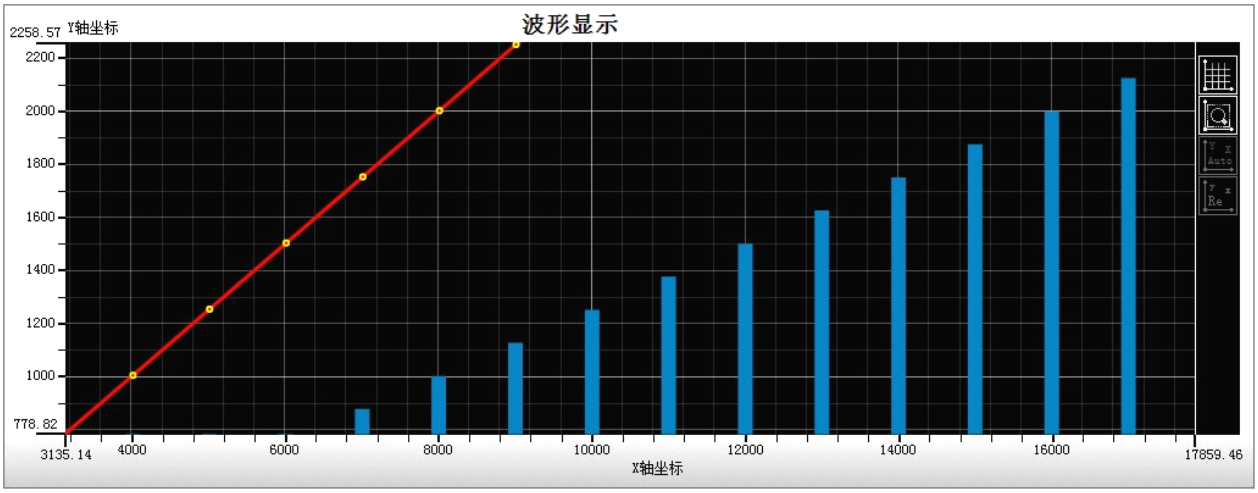


图 5- 9　波形显示控件绘制样式演示（局部放大的效果）



绘制线条的宽度都是像素为单位的，所以绘制的矩形条，即使进行了局部放大操作，矩形条的宽度任然是原先的宽度显示，可观察上图的局部放大效果。

（3）　模拟串口采样

波形显示控件在实际使用过程中，通常用户显示外界动态的波形数据，而在自动化方面，串口通讯尤为常见。演示程序内部通过开启一个 Timer 控件，按照特定的周期更新波形显示的数据，同时只需要刷新波形显示控件即可得到新的波形。串口采样按钮的单击事件只是简单得添加空数据的引用：

```
zGraphTest.f_LoadOnePix(ref x1, ref y1, Color.Red, 2);

zGraphTest.f_AddPix(ref x2, ref y2, Color.Blue, 3);

zGraphTest.f_AddPix(ref x3, ref y3, Color.FromArgb(0, 128, 192), 2);

zGraphTest.f_AddPix(ref x4, ref y4, Color.Yellow, 3);

f_timerDrawStart(); //开启 Timer 控件
```

Timer 控件的周期函数实现如下：

```
private int timerDrawI = 0;

private void timerDraw_Tick(object sender, EventArgs e)

{ x1.Add(timerDrawI); y1.Add(timerDrawI % 100);

 x2.Add(timerDrawI); y2.Add((float)Math.Sin(timerDrawI / 10f) * 200);

 x3.Add(timerDrawI); y3.Add(50);

 x4.Add(timerDrawI); y4.Add((float)Math.Sin(timerDrawI / 10) * 200);

 timerDrawI++;
```

```
zGraphTest.f_Refresh();

.....

}
```

模拟串口通讯最后实现的效果如下：

图 5- 10　模拟串口通讯的显示

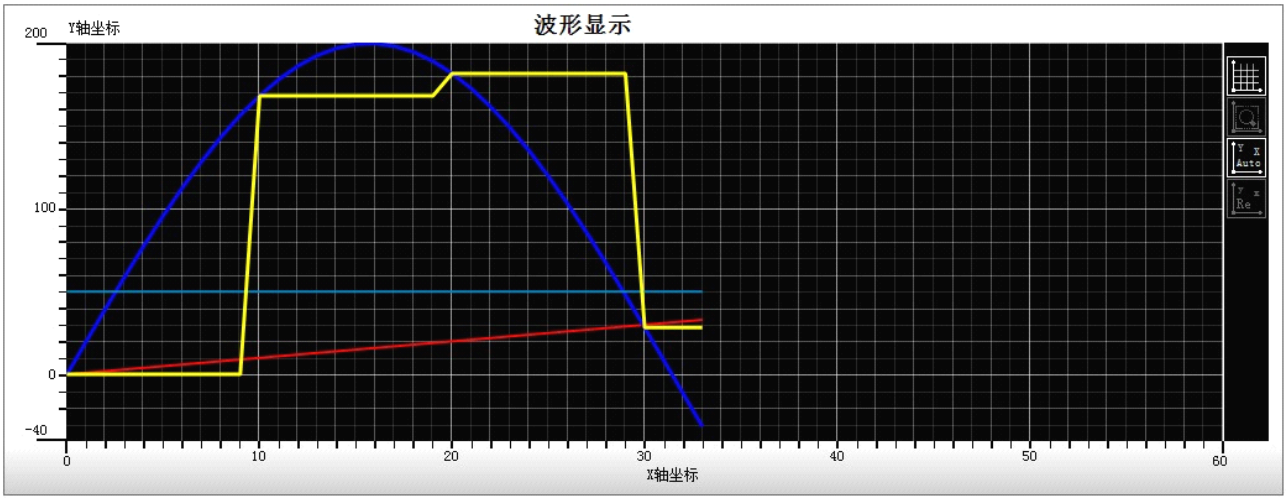
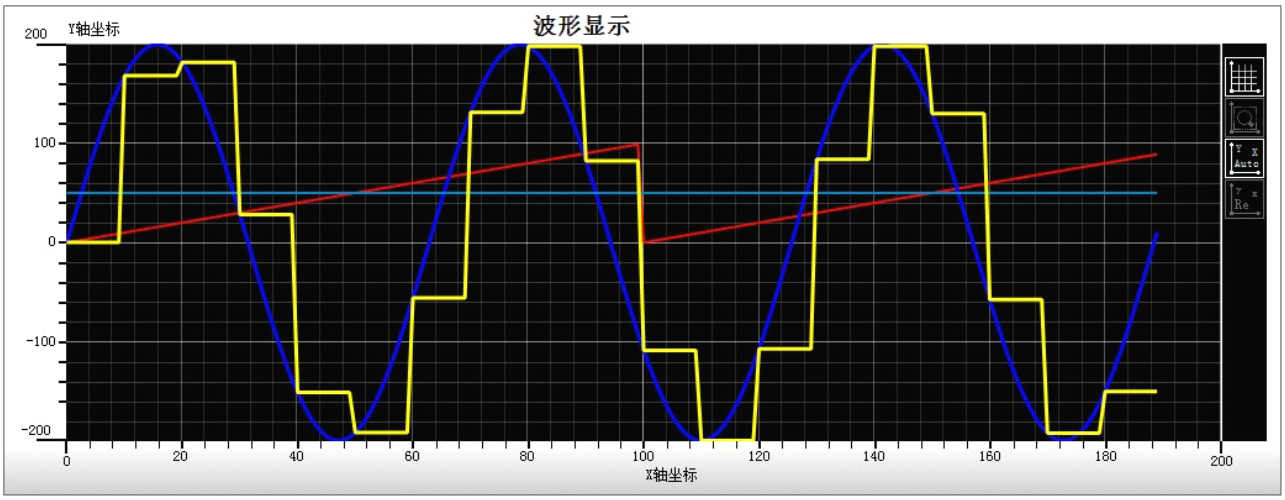


图 5- 11　模拟串口通讯的显示（一段时间后）



注意到在自动坐标模式下，波形显示的范围能够根据当前显示数据的范围自动调整，这样观察者能够观察数据的整体显示效果。

（4）　模拟随机点的显示

该演示程序模拟随即点的显示思路同模拟串口通讯，也是通过 Timer 实现的，只是数据是通过产生随机值获得的：

```
Random rand = new Random();

private void timerRandom_Tick(object sender, EventArgs e)

{ x1.Add(rand.Next(60)); //添加随机值
```

```
y1.Add((float)rand.NextDouble()); //添加随机值

zGraphTest.f_Refresh();

……

}
```

产生的效果如下：

图 5- 12　模拟随机点的显示

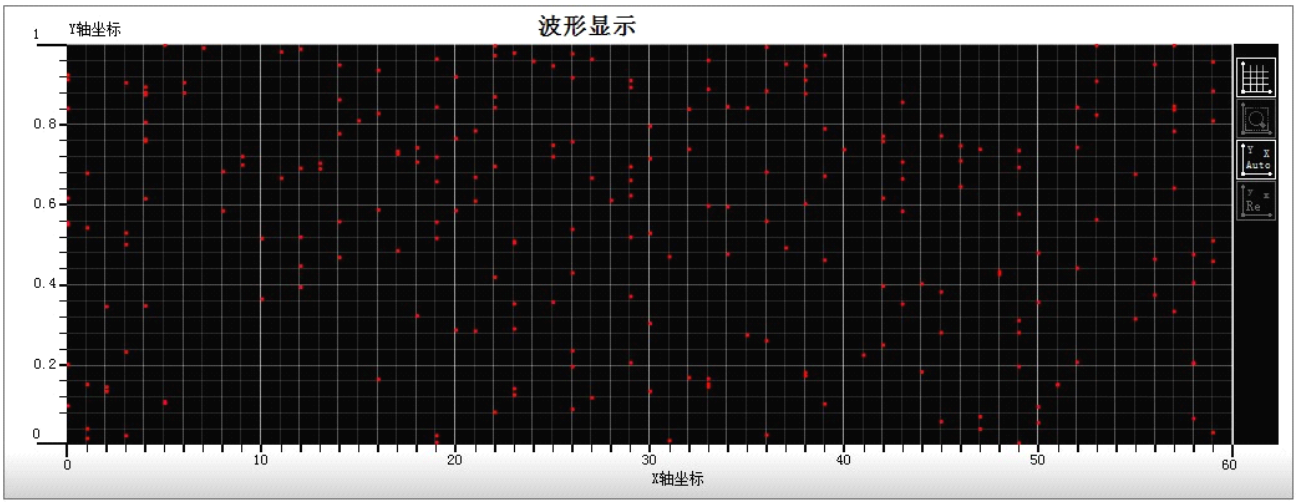
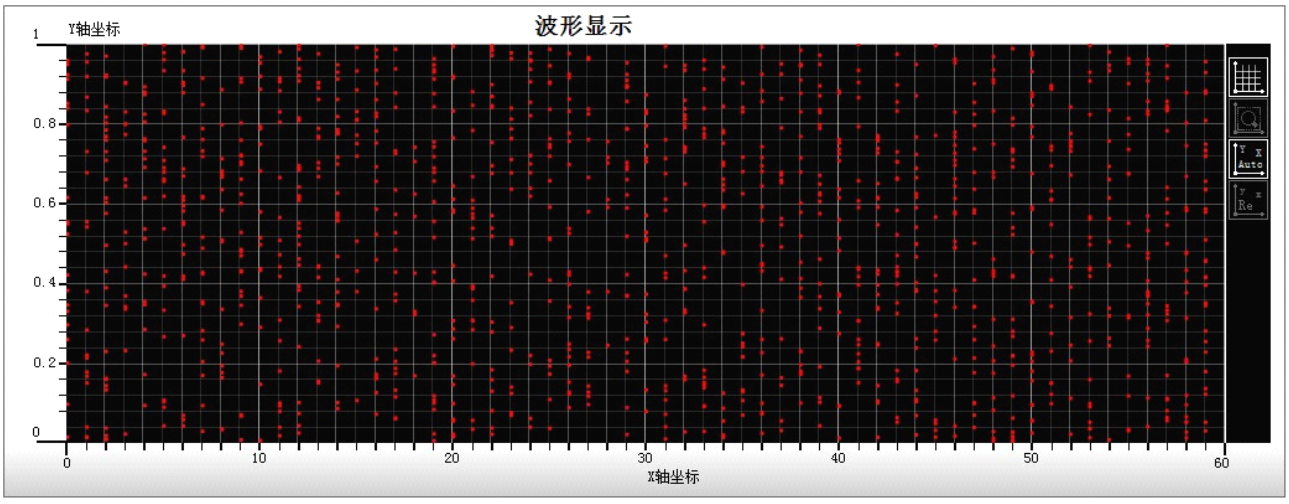


图 5- 13　模拟随机点的显示（一段时间后）

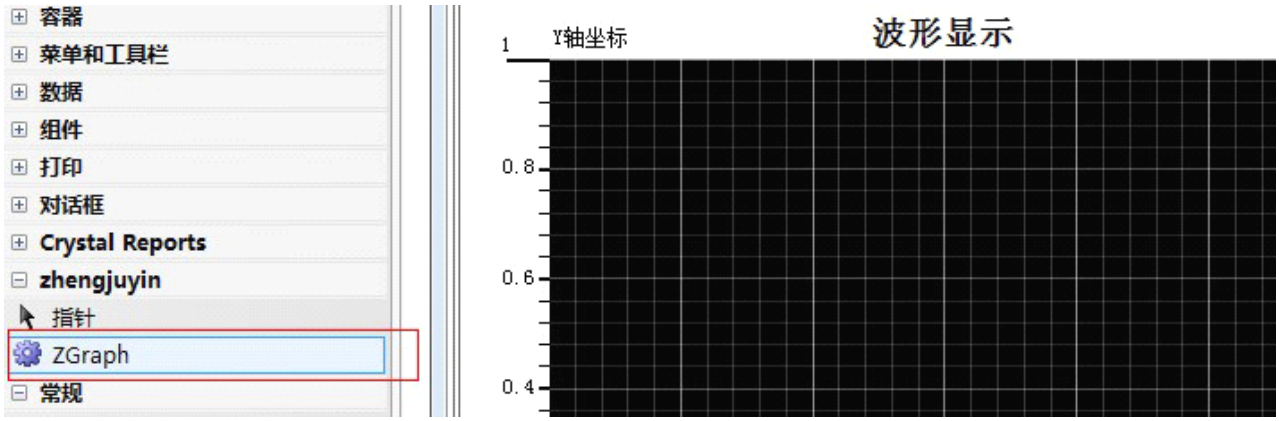


上面为该波形显示控件演示程序提供的演示内容，以及简单的使用方法。实际在使用过程中，因为 Timer 的时间间隔并不是精确的（收到消息队列的影响），通常通过多线程采样，将数据处理后更新给波形显示控件，之后显示出波形。

5.2.3 波形显示控件在实际项目中的使用

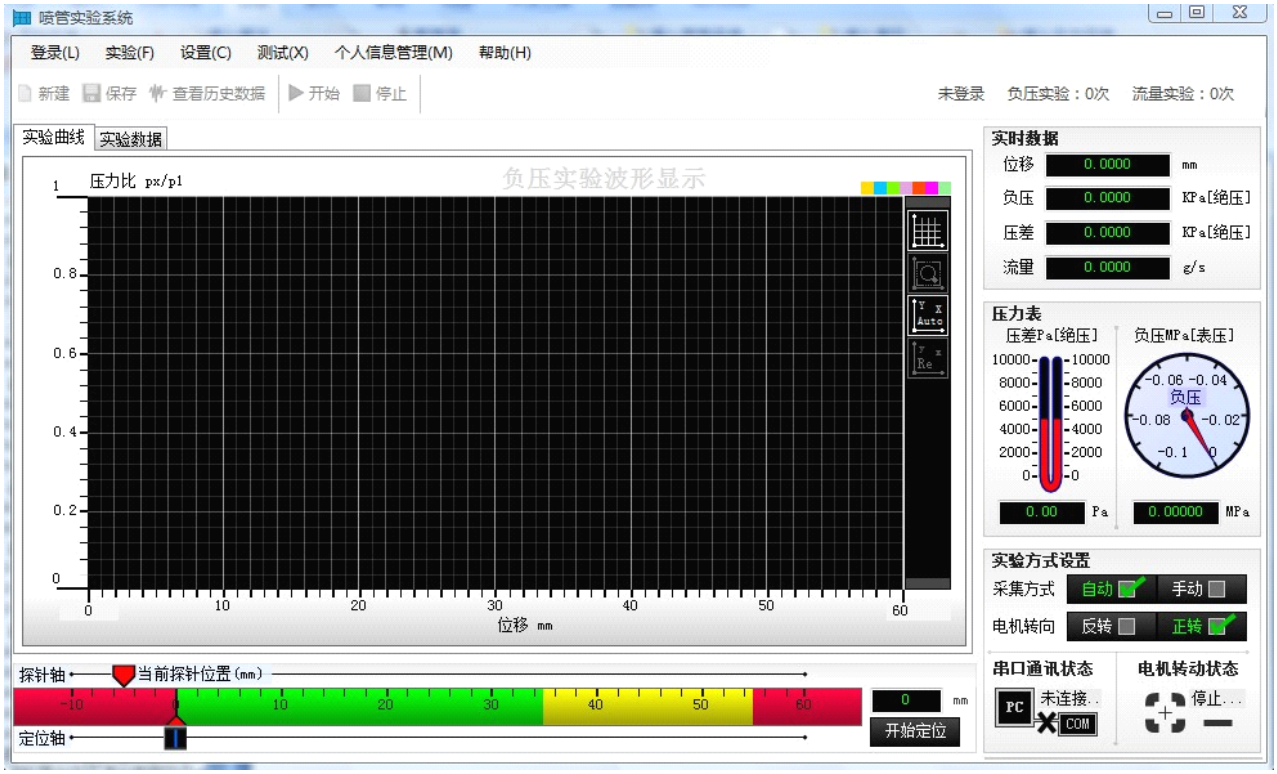
在实际使用过程中，设计者通过 Visual Studio 的工具箱，添加选项卡功能，将控件添加到工具箱中，之后通过拖拽的方式即可在设计视图添加该波形显示控件。

图 5- 14 波形显示控件的添加



下图为波形显示控件在喷管实验平台软件中的应用，经测试，完全能够满足需求：

图 5- 15 波形显示控件的实际应用



波形显示控件在数据采集过程中作为数据显示的主要窗口，另外在历史数据查询中可以用来显示历史实验数据，根据需求显示多条实验数据并进行直观的比较。

6 课题总结

本课题设计的波形显示控件通过双坐标模式，成功的实现了友好坐标标尺的显示，以及波形的动态范围自动调整。另外提供的显示网格功能、局部放大功能、以及波形显示控件的整体 UI 效果，都较为出色。并同时设计了波形显示控件演示程序，介绍控件的使用方法并观察显示效果。

在波形显示的设计上，采用的并不是将数据点绘制在内存位图上，然后通过映射位图的方式实现显示，而是直接通过坐标换算来实现显示和局部放大。因为显示的数据点集合存在较强的相互位置关系，若之前通过内存图像的方式存储，则不能保证数据量的大小，比如数据点集合遍历后的 X 轴范围为0~2^{10}，则内存中就要创建宽度为至少2^{10}像素的图像，否则会导致局部放大查看后数据显示丢失，且容易产生因为一个孤立的较远的点而浪费大量的内存空间的情况。波形显示控件在设计的初期，使用的便是 pictureBox 控件，其本身提供的缓冲后显示的效果能够使控件显示过程更为流畅。

波形显示控件在进行坐标转换时，数据溢出的处理显得尤为重要，因此限制了波形显示控件局部放大的范围，防止放大后的内部数据转换出现数据溢出。同时，坐标值转数据值和数据值转坐标值，都需要考虑到 GDI+默认坐标系的问题。

在控件测试的过程中，修改了不少遗漏的细节问题，例如放大选取框拖到控件外的情况，以及在点击放大按钮前修改软件大小，会导致放大的范围并非用户需要的范围。在设计控件接口的时候，将控件的颜色方案作为属性公开给用户，方便用户配置符合软件风格的显示方案。

最后，控件任然存在不足和需要改进的地方：

（1）本课题设计的波形显示控件并未提供平移波形的功能，实际上可利用设计局部放大功能的思路，实现波形拖拽移动的效果。

（2）局部放大功能是通过使用一个半透明的 pictureBox 实现的，这样在数据量较大的时候产生残影的效果，可以通过直接异或显示区域的颜色来改进。

（3）波形的显示过程中并为设置刷新时的裁剪区域，这样每次数据更新都会刷新整个波形显示区域，可以通过设置裁剪区域来提高绘图的效率。

（4）在显示标尺坐标值的时候，为防止浮点数加法过程中产生的显示值出错，采用了 decimal 数据类型，牺牲了绘图效率。

参考文献

[1] 闫宇晗，常鑫.在 C#中用 GDI+实现图形动态显示[J].计算机技术与发展，2006，16(12)：118.

[2] 高宏亮，王淑娟，翟国富，陈功军.采用 Visual C++实现的函数波形显示控件[J].电测与仪表，2006，43(12)：62-65.

[3] 张文，秦开宇，李志强. VC 环境下多波形显示 ActiveX 控件开发[J].中国测试，2009，35(2)：34-36.

[4] 陈本峰，苏琦. Windows GDI+的研究与应用[J].计算机应用研究，2003，20(3)：56-59.

[5] Archer T.C#技术内幕[M].侯晓霞，柴洪辉译.北京：清华大学出版社，2002.

[6] Nagel C，Evjen B.C#高级编程[M].李敏波译.第4版.北京：清华大学出版社，2006.

[7] Sells C，Weinhardt M.Windows Forms 2.0程序设计[M].汪泳译.北京：电子工业出版社，2008.

[8] SerBan I.GDI+ Custom Controls with Visual C# 2005[M].UK：Packt Publishing Ltd，2006.

[9] MacDonald M.Pro .NET 2.0 Windows Forms and Custom Controls in C#[M].Apress，2005：211-262.

[10] Troelsen A.Pro C# with .NET 3.0[M].Apress，2007：649-698.

[11] hite E.GDI+程序设计[M].杨浩，张哲峰译.北京：清华大学出版社，2002.