上海格西信息科技有限公司

自动化测试软件例子

版本 0.1

目录

1. 概述

电子设备在生产和制造测试、产品维护测试、产品检测过程中，需要各式各样的测试测量、控制软件，来满足产品研发、和生产和维护的需求。传统的研发方式是使用高级语言进行量身定制，从编写通信接口驱动、编写复杂通信协议交互流程、数据采集保存，到用户界面显示和采集数据结果报表等方面，都需要对软件开发有完全的掌控能力和精深的知识，才能够开发合适的功能以满足自身特定的测控要求。但是，如果您的开发团队跟不上需求，又会如何？如果您的开发预算不足，又会如何？

当产品测试要求的变化速度超过测试软件的更新速度时，就会出现问题，很快就会导致产品开发进度大大拖延。

在本文中，我们将使用商用测控软件开发环境——格西测控大师的一个例子来探讨如何解决自主研发测试软件遇到的问题。

本例子文件位于：<软件安装目录>\Examples\Solutions\AutomaticTestSystem。

文件说明：

✓ AutomaticTestSystem.gpj – 自动化测试软件演示项目 – 中文

例子自带仿真器，可以脱离设备仿真运行。

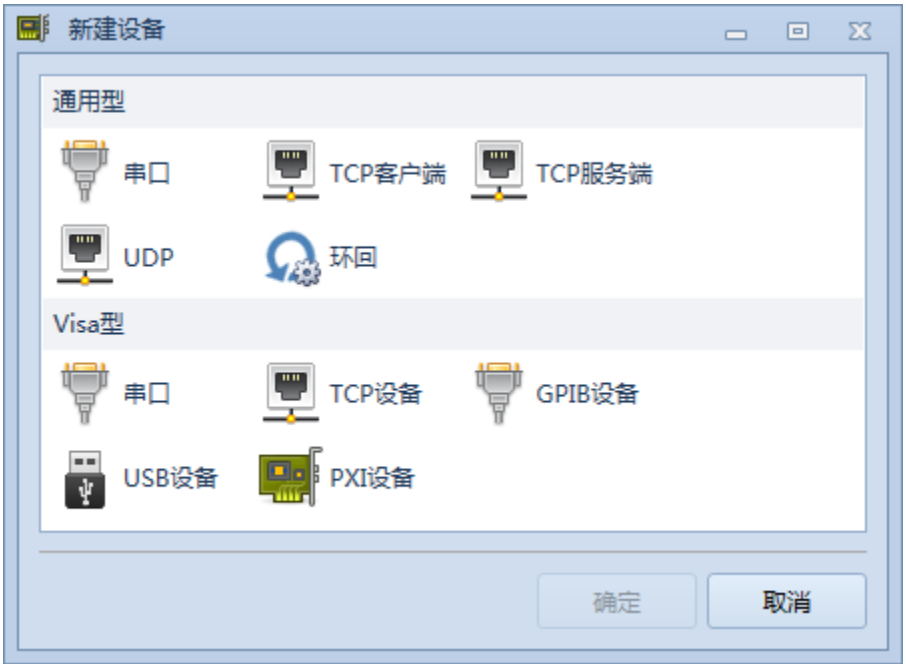
2. 便利地与任意设备互联互通

2.1 通过直接 IO 的方式与设备通信

借助软件内置的设备适配器，用户可以创建任意设备和接口组合，可以同时对不同的设备和接口进行通信，满足各种测控连接需求。

软件支持的设备和接口如下图所示，所有的设备驱动现成可用，用户无需考虑通信缓存、通信并发等问题，直接绑定协议型步骤即可进行通信收发，也可以通过脚本进行数据收发，方便灵活。

直接 IO 方式几乎可以用于任何通信场合，实现通信数据的精确控制，通常用于数据采集、通信协议监听与分析、通信协议测试、仪器仪表通信等。



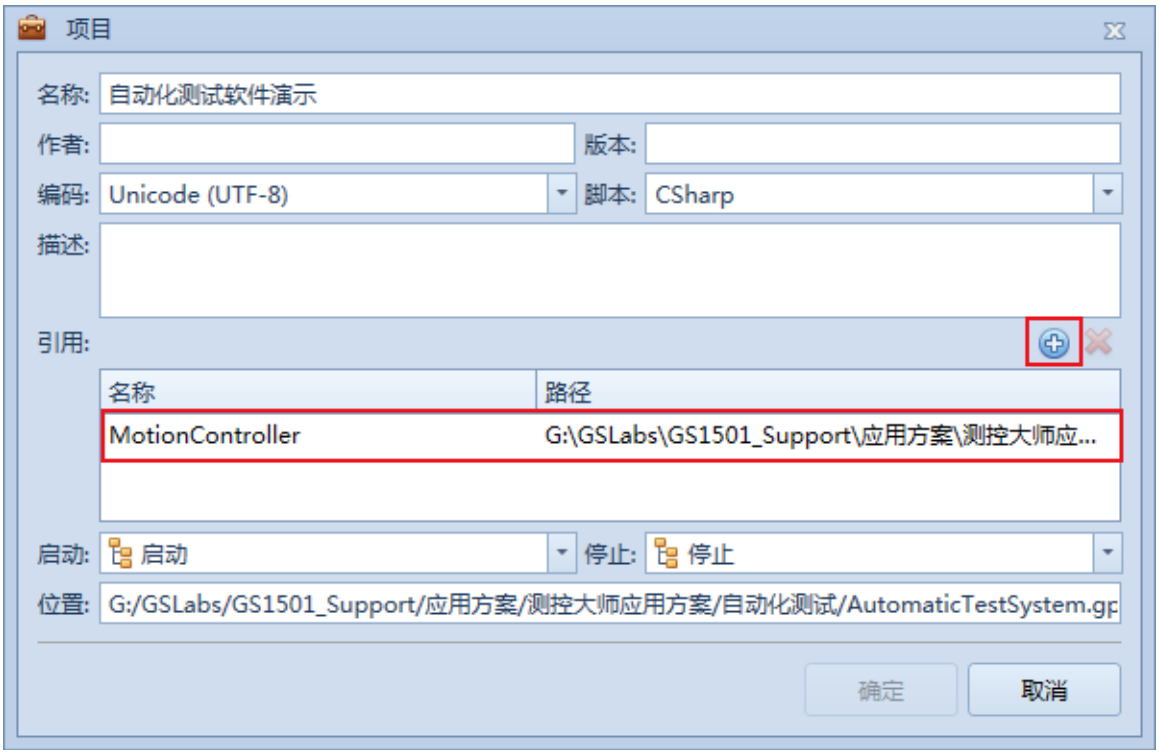
2.2 通过动态库的方式与设备通信

一般设备厂商，除了提供与设备通信的通信协议规范，供用户使用直接 IO 的方式和设备通信之外，也会提供配套的仪器控制动态库，供用户直接调用和设备通信，不需要用户处理底层的通信，方便用户进行二次开发。

本例子就是使用动态库（MotionController.dll，该库为仿真库，并未连接具体设备。）的方式对电机运动控制卡进行控制的。

引用动态库的唯一条件就是引用的动态库是一个.Net 托管动态库。如果设备厂商只提供非托管动态库，那么需要用.Net 平台包装一下，脚本便可以通过访问包装类库间接执行非托管代码库。

引用动态库功能在项目属性对话框中，如下图所示。



引用了动态库后，即可在步骤脚本或者画面脚本中调用动态库中的函数。

如：启动序列的脚本，调用动态库的 MotionManager 进行初始化。

```
using System;
using Genesis;
using Genesis.Scripting;
using Genesis.Sequence;
using Genesis.Workbench;
using MotionController; // 动态库命名空间

public class Step_16BE57B88F0D4B4CA4C73DDA665E8B45
{
  public ScriptContext Context { get; set; }
  //
  public Int32 BeginExecute(IStepContext context, IStep step)
  {
    // 配置文件
    // 打开项目配置文件，以便读取上一次保存的参数
    IMemento config = this.Context.OpenProjectConfiguration();
    IMemento motionConfig = config.GetChild("Motion");
    if (motionConfig != null)
    {
      MotionManager.LoadState(motionConfig); // 用配置文件数据配置动态库参数
    }
    IMemento manualTestConfig = config.GetChild("ManualTest");
    if (manualTestConfig != null)
```

```
    {
      ManualTest.Data.LoadState(manualTestConfig);
    }
    //
    MotorManager.Startup(this.Context);

    // 关闭工具栏、状态栏、项目管理器等
    this.Context.HideToolBar();
    this.Context.HideStatusBar();
    this.Context.HideEditorHeaders();
    this.Context.CloseAllViews();
    this.Context.CloseAllEditors();
    // 打开用户界面
    this.Context.OpenSchema("主界面");
    // 运行电机状态采集序列
    this.Context.StartStep("采集电机状态");

    return 0;
  }

  //
  public Int32 EndExecute(IStepContext context, IStep step)
  {
    return 0;
  }
}
```

如：手动测试界面的定点运动按钮事件脚本，调用动态库 MotionManager.MoveAsync 进行定点运动。

```
  // 定点运动
  public async void BtnMovePoint_Click(Object sender, RoutedEventArgs e)
  {
    double x =
Convert.ToDouble(this.Context.GetSchemaElement<TextEditBox>(sender,"TxtMovePointX").Edit
Value);
    double y =
Convert.ToDouble(this.Context.GetSchemaElement<TextEditBox>(sender,"TxtMovePointY").Edit
Value);
    double z =
Convert.ToDouble(this.Context.GetSchemaElement<TextEditBox>(sender,"TxtMovePointZ").Edit
Value);
    Point3D point = new Point3D(x, y, z);

    this.Context.Variants["电机变量/电机最新状态"].Purge();
    MotorManager.ClearUI();
    MotorManager.CommandRunning = true;
    MotorManager.CommandStarted = true;
```

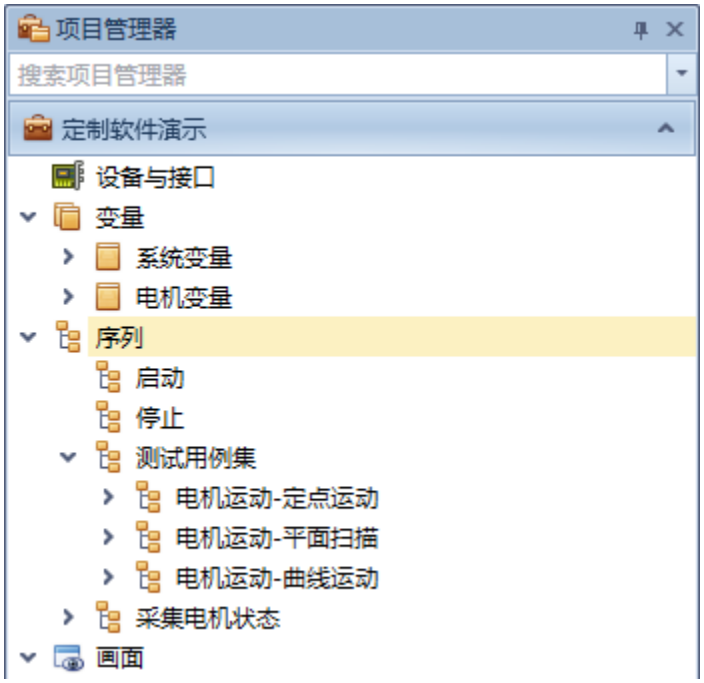
```
    while(MotorManager.CommandStarted)
    {
      System.Threading.Thread.Sleep(10); // 等待采样第一个点
    }
    MotionStatus result = await MotionManager.MoveAsync(point); // 动态库的定点运动接口
    if (result != MotionStatus.OK)
    {
      this.Context.ShowMessageBox("手动测试",string.Format("启动定点运动失败（{0}），请检查坐标是否越界！
",result.ToString()),System.Windows.MessageBoxButton.OK,System.Windows.MessageBoxImage.Error);
    }
    MotorManager.CommandEnded = true;
    MotorManager.CommandRunning = false;
  }
```

3. 灵活地编写复杂序列交互流程

借助软件内置的序列适配器，用户可以创建执行序列，实现任意逻辑的执行过程，满足各种测控自动化需求。

- 支持流程控制，如分支语句 If、Switch，循环语句 For、While，并行语句 Parallel。
- 支持同步控制，如等待（Wait）、通知（Notification）。
- 支持数值类型动作步骤（Value）、协议类型动作步骤（Protocol）、进程类型动作步骤（Process）。
- 支持序列嵌套，支持复杂的层次结构。
- 支持脚本，脚本可以无缝调用.Net Framework 类库，调用第三方托管库来实现执行逻辑。

本例子使用的序列如下图所示。



3.1 启动和停止序列

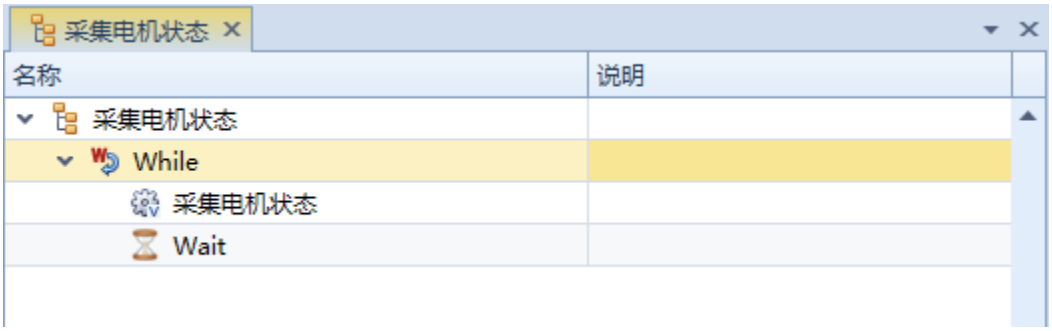
启动和停止序列：分别控制项目启动运行和停止运行阶段的动作，一般是通过脚本进行控制，启动运行脚本中加载配置文件，初始化设备，初始化界面等，停止运行脚本中保存配置文件，停止设备等。

```
using System;
using System.IO;
using Genesis;
using Genesis.Scripting;
using Genesis.Sequence;
using Genesis.Workbench;
using MotionController;

// 停止序列脚本
public class Step_EC68C9AD9F50466CB4A384D2124EE6EF
{
  public ScriptContext Context { get; set; }

  //
  public Int32 BeginExecute(IStepContext context, IStep step)
  {
    // 创建和保存项目配置文件
    IMemento config = this.Context.CreateProjectConfiguration();
    IMemento motionConfig = config.CreateChild("Motion");
    MotionManager.SaveState(motionConfig);

    IMemento manualTestConfig = config.CreateChild("ManualTest");
    ManualTest.Data.SaveState(manualTestConfig);

    this.Context.SaveProjectConfiguration(config);

    //
    MotorManager.Shutdown();

    // 备份 MotorData.db
    string dataFilename =
System.IO.Path.Combine(this.Context.ProjectDirectory,"MotorData.db");
    string dataFilenameDes = AppendFileNameSuffix(dataFilename,
DateTime.Now.ToString("yyyyMMddHHmmss"));
    File.Copy(dataFilename,dataFilenameDes);
    File.Delete(dataFilename);

    return 0;
  }

  //
  public Int32 EndExecute(IStepContext context, IStep step)
```

```
  {
    return 0;
  }

  private string AppendFileNameSuffix(string fileName, string suffix)
  {
    if (fileName.LastIndexOf('.') > 0)
    {
      return fileName.Substring(0, fileName.LastIndexOf('.')) + suffix +
fileName.Substring(fileName.LastIndexOf('.'));
    }
    return fileName + suffix;
  }
}
```

3.2 采集电机状态序列

采集电机状态序列：通过 While 步骤无限循环，在 Value 型步骤的脚本中进行采集，每次采集后采用 Wait 步骤延时 500ms 后进行下一次采集。



采集电机状态步骤的脚本如下，调用动态库 MotionManager.GetStatus 接口函数进行状态采集，采集后的数据保存到变量中，自动进行数据存储和用户界面显示。

```
using System;
using Genesis;
using Genesis.Scripting;
using Genesis.Sequence;
using Genesis.Workbench;
using MotionController;

public class Step_ECE8F3BE0F9943F29A088C5D670D9A47
{
  public ScriptContext Context { get; set; }

  //
  public Int32 BeginExecute(IStepContext context, IStep step)
  {
    if (!MotorManager.Opened)
    {
      return 0;
```

```
    }
    //
    Int32 run = 0;
    Int32 dir = 0;
    Int32 negative = 0;
    Int32 positive = 0;
    Int32 zero = 0;
    Int32 mode = 0;
    double velocityX = 0;
    double velocityY = 0;
    double velocityZ = 0;
    double positionX = 0;
    double positionY = 0;
    double positionZ = 0;
    MotionManager.GetStatus(MotionAxis.X,ref run,ref dir,ref negative,ref positive,ref
zero,ref mode,ref velocityX,ref positionX);
    this.Context.Variants["电机变量/X 轴状态/Run"] = run;
    this.Context.Variants["电机变量/X 轴状态/Mode"] = mode;
    this.Context.Variants["电机变量/X 轴状态/Direction"] = dir;
    this.Context.Variants["电机变量/X 轴状态/Zero"] = zero;
    this.Context.Variants["电机变量/X 轴状态/PositiveLimit"] = positive;
    this.Context.Variants["电机变量/X 轴状态/NegativeLimit"] = negative;
    this.Context.Variants["电机变量/X 轴状态/Velocity"] = velocityX;
    this.Context.Variants["电机变量/X 轴状态/Position"] = positionX;

    //this.Context.LogMessage(string.Format("X:Run={0} Mode={1}",run, mode));
    //
    MotionManager.GetStatus(MotionAxis.Y,ref run,ref dir,ref negative,ref positive,ref
zero,ref mode,ref velocityY,ref positionY);
    this.Context.Variants["电机变量/Y 轴状态/Run"] = run;
    this.Context.Variants["电机变量/Y 轴状态/Mode"] = mode;
    this.Context.Variants["电机变量/Y 轴状态/Direction"] = dir;
    this.Context.Variants["电机变量/Y 轴状态/Zero"] = zero;
    this.Context.Variants["电机变量/Y 轴状态/PositiveLimit"] = positive;
    this.Context.Variants["电机变量/Y 轴状态/NegativeLimit"] = negative;
    this.Context.Variants["电机变量/Y 轴状态/Velocity"] = velocityY;
    this.Context.Variants["电机变量/Y 轴状态/Position"] = positionY;
    //
    MotionManager.GetStatus(MotionAxis.Z,ref run,ref dir,ref negative,ref positive,ref
zero,ref mode,ref velocityZ,ref positionZ);
    this.Context.Variants["电机变量/Z 轴状态/Run"] = run;
    this.Context.Variants["电机变量/Z 轴状态/Mode"] = mode;
    this.Context.Variants["电机变量/Z 轴状态/Direction"] = dir;
    this.Context.Variants["电机变量/Z 轴状态/Zero"] = zero;
    this.Context.Variants["电机变量/Z 轴状态/PositiveLimit"] = positive;
    this.Context.Variants["电机变量/Z 轴状态/NegativeLimit"] = negative;
    this.Context.Variants["电机变量/Z 轴状态/Velocity"] = velocityZ;
```

```
    this.Context.Variants["电机变量/Z 轴状态/Position"] = positionZ;

    ////////////////////////////////////////////////////////////////////////////////
////////////////////////
    DateTime time = DateTime.Now;
    // 电机变量/电机状态 是一个数据库存储型变量，每次更新后自动存储一条记录。
    this.Context.Variants["电机变量/电机状态/Time"] = time;
    this.Context.Variants["电机变量/电机状态/XPosition"] = positionX;
    this.Context.Variants["电机变量/电机状态/YPosition"] = positionY;
    this.Context.Variants["电机变量/电机状态/ZPosition"] = positionZ;
    this.Context.Variants["电机变量/电机状态/XVelocity"] = velocityX;
    this.Context.Variants["电机变量/电机状态/YVelocity"] = velocityY;
    this.Context.Variants["电机变量/电机状态/ZVelocity"] = velocityZ;
    //
    if (MotorManager.CommandRunning || MotorManager.CommandStarted ||
MotorManager.CommandEnded)
    {
      this.Context.Variants["电机变量/电机最新状态/Time"] = time;
      this.Context.Variants["电机变量/电机最新状态/XPosition"] = positionX;
      this.Context.Variants["电机变量/电机最新状态/YPosition"] = positionY;
      this.Context.Variants["电机变量/电机最新状态/ZPosition"] = positionZ;
      this.Context.Variants["电机变量/电机最新状态/XVelocity"] = velocityX;
      this.Context.Variants["电机变量/电机最新状态/YVelocity"] = velocityY;
      this.Context.Variants["电机变量/电机最新状态/ZVelocity"] = velocityZ;
      //
      MotorManager.CommandStarted = false;
      MotorManager.CommandEnded = false;
    }

    return 0;
  }

  //
  public Int32 EndExecute(IStepContext context, IStep step)
  {
    return 0;
  }
}
```

3.3 测试用例集序列

测试用例集序列：该序列是自动测试的用例集，可以根据需求增加测试用例。本软件系统提供了一个自动化测试框架，供用户快速构建基于模版的自动化测试程序，该框架的主要类是 StepExecutionScheme 和 StepExecutionCase，StepExecutionScheme 是测试方案类，保存测试方案设置和测试用例设置，运行测试用例等；StepExecutionCase 是测试用例类，保存测试用例设置。

自动测试总体脚本位于“自动测试界面”的脚本中，如下：

```
//
// 自动测试全局类
//
public static class AutoTest
{
  private static StepExecutionScheme s_scheme = null;
  private static StepExecutionCase s_selectedCase = null;
  private static object s_selectedCaseParameter = null;

  static AutoTest()
  {
    InitializeCommands();
    InitializeTemplateCases();
  }

  public static ScriptContext Context { get; set; }

  public static StepExecutionScheme Scheme
  {
    get
    {
      return s_scheme;
    }
    set
    {
      if (s_scheme != null)
      {
```

```
        s_scheme.Started -= Scheme_Started;
        s_scheme.Stopped -= Scheme_Stopped;

        s_scheme.CaseStarted -= Scheme_CaseStarted;
        s_scheme.CaseStopped -= Scheme_CaseStopped;
      }
      s_scheme = value;
      if (s_scheme != null)
      {
        s_scheme.Started += Scheme_Started;
        s_scheme.Stopped += Scheme_Stopped;

        s_scheme.CaseStarted += Scheme_CaseStarted;
        s_scheme.CaseStopped += Scheme_CaseStopped;
      }
    }
  }

  public static StepExecutionCase SelectedCase
  {
    get
    {
      return s_selectedCase;
    }
    set
    {
      if (s_selectedCase != value)
      {
        s_selectedCase = value;
      }
    }
  }

  public static object SelectedCaseParameter
  {
    get
    {
      return s_selectedCaseParameter;
    }
    set
    {
      if (s_selectedCaseParameter != value)
      {
        s_selectedCaseParameter = value;
      }
    }
  }
```

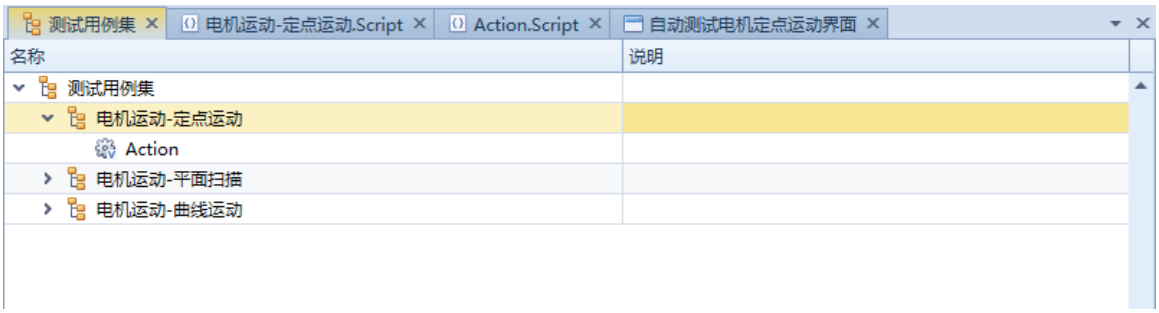
```
  //
  // 测试用例模版
  //
  private static List<StepExecutionCase> s_templateCases = new
List<StepExecutionCase>();
  public static List<StepExecutionCase> TemplateCases
  {
    get
    {
      return s_templateCases;
    }
  }

  private static void InitializeTemplateCases()
  {
    // 模版添加在此
    StepExecutionCase c = new StepExecutionCase() {Name="电机运动-定点运动",
Description="三轴运动到指定的 X、Y、Z 坐标点。"};
    c.Step = "测试用例集/电机运动-定点运动";
    c.SetParameterObject(new TCMotorMovePoint());
    c.Editor = "自动测试电机定点运动界面";
    s_templateCases.Add(c);

    c = new StepExecutionCase() {Name="电机运动-平面扫描", Description="任意两维的蛇形
平面扫描运动。"};
    c.Step = "测试用例集/电机运动-平面扫描";
    c.SetParameterObject(new TCMotorMovePoint());
    c.Editor = "自动测试电机平面扫描界面";
    s_templateCases.Add(c);

    c = new StepExecutionCase() {Name="电机运动-曲线运动", Description="任意三维曲线运
动。"};
    c.Step = "测试用例集/电机运动-曲线运动";
    c.SetParameterObject(new TCMotorMoveCurve());
    c.Editor = "自动测试电机曲线运动界面";
    s_templateCases.Add(c);
  }
  //
  //
  //
  public static void ShowStatusMessage(string message)
  {
    Context.Variants["系统变量/状态消息"] = message;
  }

  public static void ShowPageMessage()
  {
```

```
    Context.Variants["系统变量/页面消息"] = "自动测试";
  }
  //
  // 相关命令
  //
  public static ICommand NewSchemeCommand { get; private set; }
  public static ICommand OpenSchemeCommand { get; private set; }
  public static ICommand SaveSchemeCommand { get; private set; }

  public static ICommand StartupCommand { get; private set; }
  public static ICommand ShutdownCommand { get; private set; }


  private static void InitializeCommands()
  {
    NewSchemeCommand = new DelegateCommand(NewScheme_Execute,NewScheme_CanExecute);
    OpenSchemeCommand = new DelegateCommand(OpenScheme_Execute,OpenScheme_CanExecute);
    SaveSchemeCommand = new DelegateCommand(SaveScheme_Execute,SaveScheme_CanExecute);

    StartupCommand = new DelegateCommand(Startup_Execute,Startup_CanExecute);
    ShutdownCommand = new DelegateCommand(Shutdown_Execute,Shutdown_CanExecute);
  }

  private static bool NewScheme_CanExecute()
  {
    return AutoTest.Scheme == null || (AutoTest.Scheme != null
&& !AutoTest.Scheme.Power);
  }

  private static void NewScheme_Execute()
  {

  }

  private static bool OpenScheme_CanExecute()
  {
    return AutoTest.Scheme == null || (AutoTest.Scheme != null
&& !AutoTest.Scheme.Power);
  }

  private static void OpenScheme_Execute()
  {

  }

  private static bool SaveScheme_CanExecute()
  {
    return AutoTest.Scheme != null;
```

```
  }

  private static void SaveScheme_Execute()
  {
    if (AutoTest.SelectedCase != null)
    {
      AutoTest.SelectedCase.SetParameterObject(AutoTest.SelectedCaseParameter);
    }
    AutoTest.Scheme.Save();
  }

  private static bool Startup_CanExecute()
  {
    return AutoTest.Scheme != null && !AutoTest.Scheme.Power;
  }

  private static void Startup_Execute()
  {
    if (!MotorManager.Opened)
    {
      MotorManager.Open();
    }
    if (MotorManager.Opened)
    {
      AutoTest.Scheme.Startup();
    }
  }

  private static bool Shutdown_CanExecute()
  {
    return AutoTest.Scheme != null && AutoTest.Scheme.Power;
  }

  private static void Shutdown_Execute()
  {
    MotionManager.Stop(); // 主动停止需要调用，否则步骤结束，运动还在继续。
    //
    AutoTest.Scheme.Shutdown();
  }
}
```

测试用例模版是预先构建的，一个测试用例模版分为三部分，分别为执行序列、用例参数类和用例参数设置界面。以“电机运动-定点运动”用例为例说明。

1）执行序列

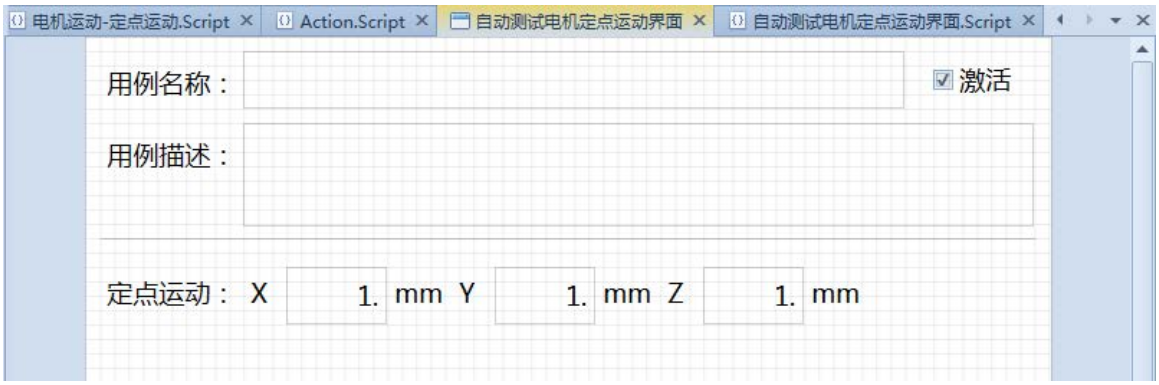


2）用例参数类，本例子的参数类创建在“电机运动-定点运动”序列的脚本中。

```
using System;
using System.Text;
using Genesis;
using Genesis.Scripting;
using Genesis.Sequence;
using Genesis.Workbench;

// 电机运动-定点运动参数类
public class TCMotorMovePoint
{
  public TCMotorMovePoint()
  {
    X = 0;
    Y = 0;
    Z = 0;
  }

  public double X {get;set;}
  public double Y {get;set;}
  public double Z {get;set;}
}

public class Step_5F60ECE08ED3443F84BF2694B1BE2F4D
{
  public ScriptContext Context { get; set; }

  //
  public Int32 BeginExecute(IStepContext context, IStep step)
  {
    return 0;
  }

  //
  public Int32 EndExecute(IStepContext context, IStep step)
  {
    return 0;
  }
```

```
}
```

3）用例参数设置界面，本用例的参数设置界面为 “自动测试电机定点运动界面”。



“自动测试电机定点运动界面”的脚本主要功能是将测试用例的参数和界面控件绑定，以便控件的数值修改时自动修改测试用例参数的值。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Windows;
using System.Windows.Data;
using System.Windows.Input;
using System.Windows.Controls;
using Genesis;
using Genesis.Scripting;
using Genesis.Sequence;
using Genesis.Workbench;
using Genesis.Workbench.Schema;
using Genesis.Workbench.Sequence;
using Genesis.Windows.Input;
using Genesis.Windows.Controls;

public class Schema_FB9E71324FD24FBC88DF8129228E35B4
{
  public ScriptContext Context { get; set; }

  //
  public void 自动测试电机定点运动界面_Loaded(Object sender,
System.Windows.RoutedEventArgs e)
  {
    InitializeControls(sender);
  }

  // 使用绑定机制，初始化控件
  private void InitializeControls(Object sender)
  {
```

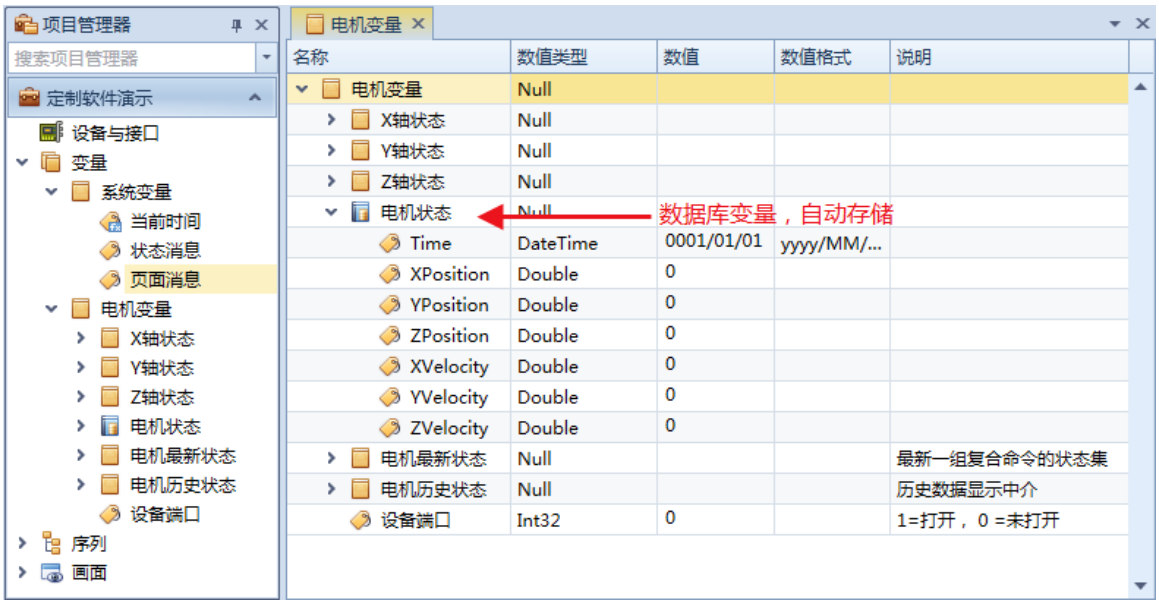
```
    StepExecutionScheme scheme = AutoTest.Scheme;
    StepExecutionCase selectedCase = AutoTest.SelectedCase;
    if (scheme == null || selectedCase == null || selectedCase.Editor != "自动测试电机定点运动界面")
    {
      return;
    }
    // 绑定用例属性
    this.Context.GetSchemaElement<TextEditBox>(sender,"TxtCaseName").SetBinding(TextEditBox.EditValueProperty,new Binding("Name") {Source = selectedCase, Mode = BindingMode.TwoWay});
    this.Context.GetSchemaElement<Genesis.Windows.Controls.CheckBox>(sender,"ChkCaseActive").SetBinding(Genesis.Windows.Controls.CheckBox.IsCheckedProperty,new Binding("Active") {Source = selectedCase, Mode = BindingMode.TwoWay});
    this.Context.GetSchemaElement<TextEditBox>(sender,"TxtCaseDescription").SetBinding(TextEditBox.EditValueProperty,new Binding("Description") {Source = selectedCase, Mode = BindingMode.TwoWay});

    TCMotorMovePoint parameters = selectedCase.GetParameterObject<TCMotorMovePoint>();
    AutoTest.SelectedCaseParameter = parameters;

    this.Context.GetSchemaElement<TextEditBox>(sender,"TxtMovePointX").SetBinding(TextEditBox.EditValueProperty,new Binding("X") {Source = parameters, Mode = BindingMode.TwoWay});
    this.Context.GetSchemaElement<TextEditBox>(sender,"TxtMovePointY").SetBinding(TextEditBox.EditValueProperty,new Binding("Y") {Source = parameters, Mode = BindingMode.TwoWay});
    this.Context.GetSchemaElement<TextEditBox>(sender,"TxtMovePointZ").SetBinding(TextEditBox.EditValueProperty,new Binding("Z") {Source = parameters, Mode = BindingMode.TwoWay});
  }

}
```

4. 强大的数据处理和存储

软件系统是通过变量来进行各种测控数据传递、呈现和存储的，变量肩负着媒介的作用。例如，序列步骤通过设备接口采集到数据，存入数据库变量，然后数据库变量把数据自动存入数据库，同时变量的值直接显示到绑定的用户界面控件中。

本例子用到的变量分为两类，一是系统变量，用一个变量容器保存，有当前时间、状态消息等；一是电机变量，不需要持久化的用变量容器，需要持久化的用一个数据库变量容器，如下图所示。

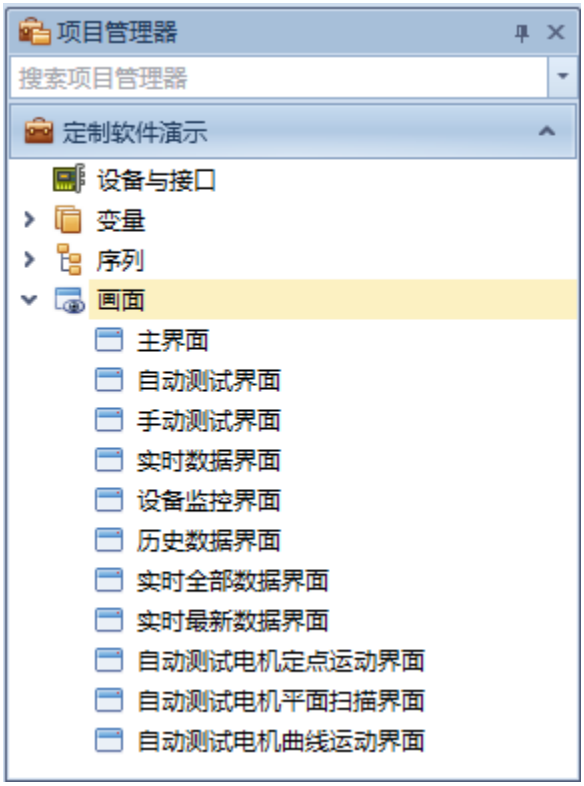
采集数据和更新变量通过脚本的方式进行，详情参考 3.2 节。



5. 快速地构建现代化的用户界面

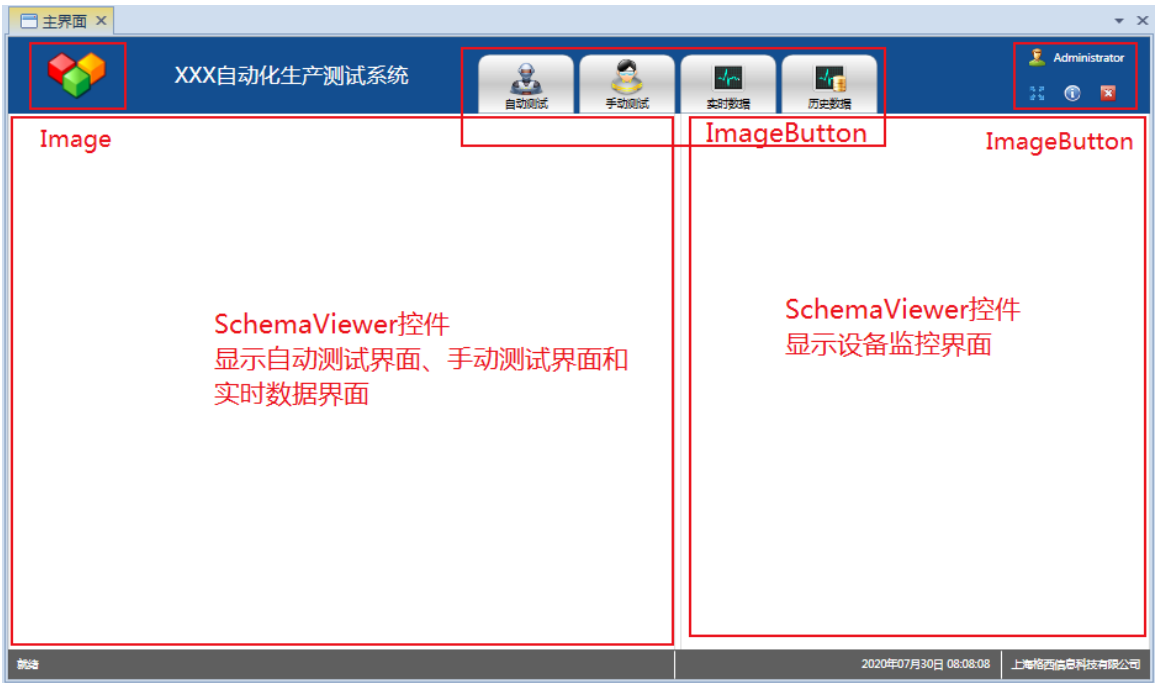
借助软件内置的画面编辑器，提供丰富的图元和控件库，用户可以很快速地创建和编辑现代化的用户界面，实现各种测控操作和显示画面。

本例子项目包含的用户画面如下图所示。



5.1 主界面

项目主界面通过 SchemaViewer 控件划分区域，构成用户界面的框架。



主界面的脚本主要功能是通过工具栏按钮加载具体用户界面和实现一些全局功能。

```
using System;
using System.Windows;
using System.Windows.Controls;
using Genesis;
using Genesis.Scripting;
using Genesis.Sequence;
using Genesis.Workbench;
using Genesis.Windows.Controls;
using Genesis.Workbench.Schema;

//
// 软件主全局类
//
public static class Global
{
  static Global()
  {
    FullScreen = false;
  }

  public static ScriptContext Context { get; set;}

  //
  public static bool FullScreen
  {
    get;
    set;
```

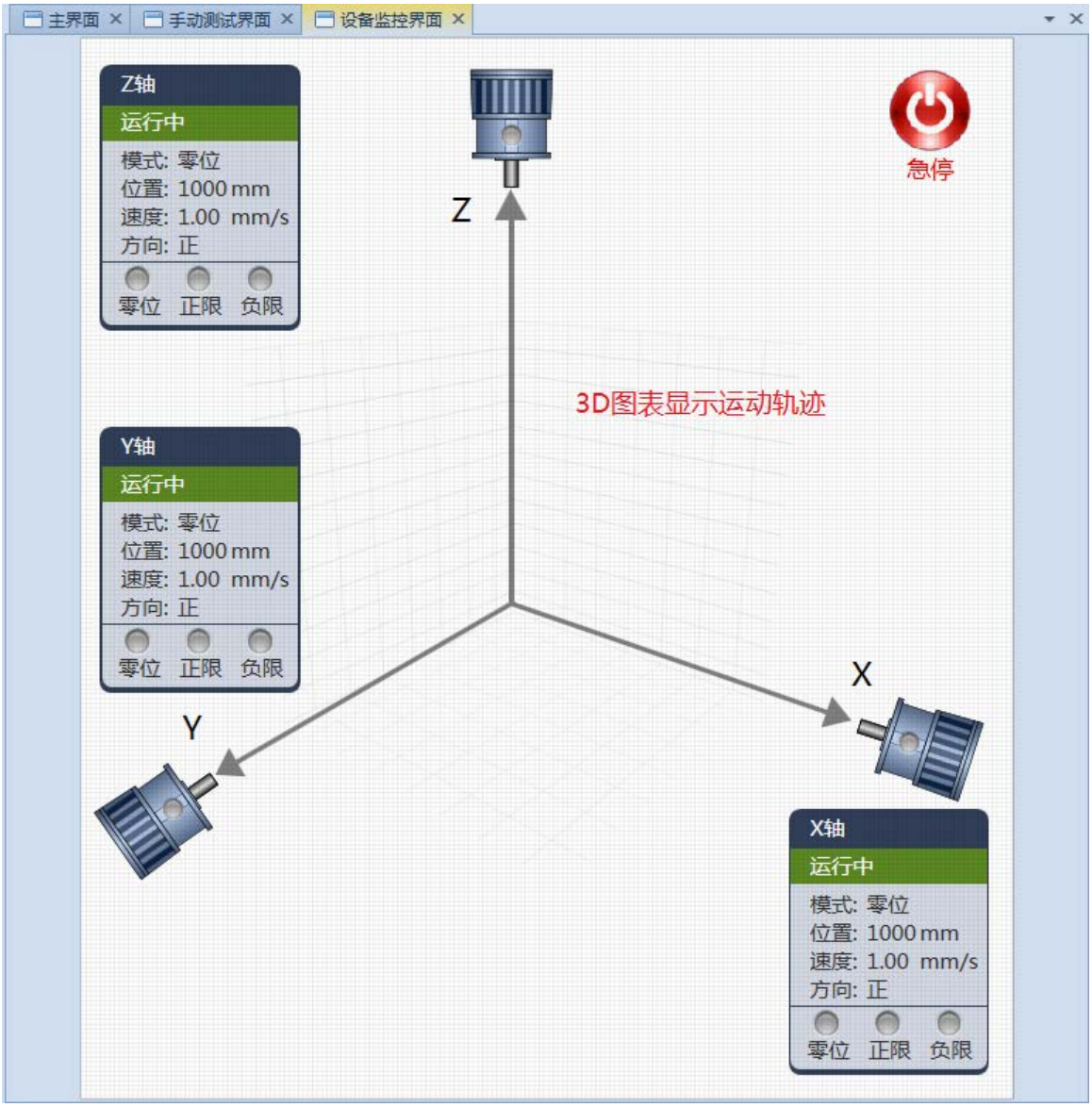
```
  }
}

public class Schema_D467469ED3F345BC993D4C3AFBB131B7
{
  public ScriptContext Context { get; set; }

  //
  public void ButtonLogout_Click(Object sender, System.Windows.RoutedEventArgs e)
  {
    this.Context.Logout();
  }

  //
  public void ButtonAbout_Click(Object sender, RoutedEventArgs e)
  {
    this.Context.ShowAboutDialog();
  }
  //
  public void BtnShowFullScreen_Click(Object sender, RoutedEventArgs e)
  {
    Global.FullScreen = !Global.FullScreen;
    //
    Context.ShowFullScreen(Global.FullScreen);
  }
  //
  public void BtnExitSystem_Click(Object sender, RoutedEventArgs e)
  {
    if (Context.ShowMessageBox("系统","是否要关闭软件系统?
",System.Windows.MessageBoxButton.YesNo,System.Windows.MessageBoxImage.Question) ==
MessageBoxResult.Yes)
    {
      Context.Shutdown();
    }
  }
  // 主界面画面加载成功后执行的功能
  public void 主界面_Loaded(Object sender, System.Windows.RoutedEventArgs e)
  {
    Global.Context = this.Context;
    //
    TextBlock tbCurrentUser =
this.Context.GetSchemaElement<TextBlock>(sender,"TextBlockCurrentUser");
    tbCurrentUser.Text = this.Context.GetUserName();

    // 加载功能界面
    this.Context.GetSchemaElement<SchemaViewer>(sender,"SchViewerController").Load("手
动测试界面");
```

```
    this.Context.GetSchemaElement<SchemaViewer>(sender,"SchViewerMonitor").Load("设备
监控界面");

  }
  //
  public void ButtonAutoTest_Click(Object sender, RoutedEventArgs e)
  {
    SchemaViewer viewer
=this.Context.GetSchemaElement<SchemaViewer>(sender,"SchViewerController");
    viewer.Width = 1122;
    viewer.Load("自动测试界面");
    //
    ShowStatusMessage("自动测试");
  }
  //
  public void ButtonManualTest_Click(Object sender, RoutedEventArgs e)
  {
    SchemaViewer viewer
=this.Context.GetSchemaElement<SchemaViewer>(sender,"SchViewerController");
    viewer.Width = 1122;
    viewer.Load("手动测试界面");
    //
    ShowStatusMessage("手动测试");
  }
  //
  public void ButtonLiveData_Click(Object sender, RoutedEventArgs e)
  {
    SchemaViewer viewer
=this.Context.GetSchemaElement<SchemaViewer>(sender,"SchViewerController");
    viewer.Width = 1122;
    viewer.Load("实时数据界面");
    //
    ShowStatusMessage("实时数据");
  }
  //
  public void ButtonHistoricalData_Click(Object sender, RoutedEventArgs e)
  {
    SchemaViewer viewer
=this.Context.GetSchemaElement<SchemaViewer>(sender,"SchViewerController");
    viewer.Width = 1920;
    viewer.Load("历史数据界面");
    //
    ShowStatusMessage("历史数据");
  }
  private void ShowStatusMessage(string message)
  {
    this.Context.Variants["系统变量/状态消息"] = message;
```

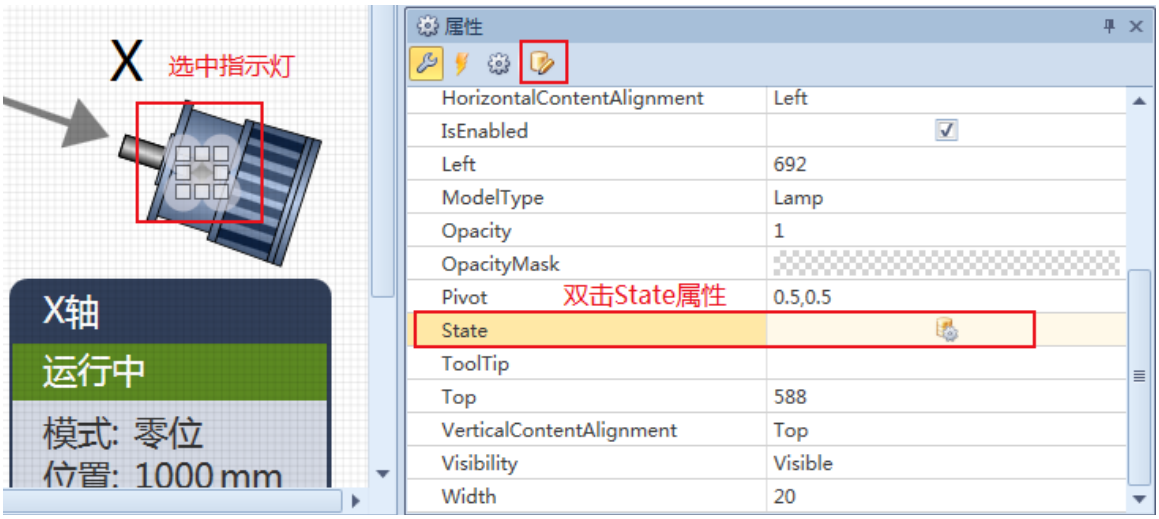
```
    }
}
```

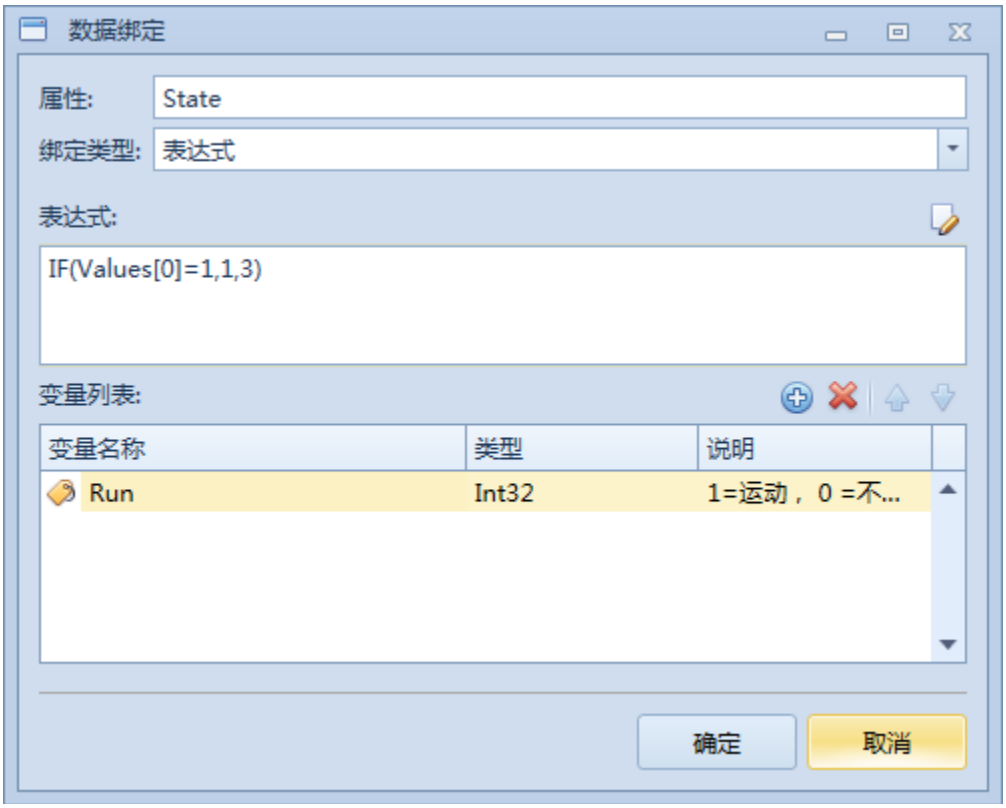
5.2 设备监控界面

设备监控界面的设计类似传统工业监控组态软件，通过软件系统提供的图库、控件库，构建界面，然后通过数据绑定的方式，把数据显示到对应的控件中。



数据绑定的操作方法比较简单，例如本例子 X 轴电机上的指示灯，需要绑定到变量“电机变量/X 轴状态/Run”，先选中指示灯控件，然后在属性面板双击“State”属性，弹出数据绑定对话框，选择绑定类型为“表达式”（因为变量 Run 的值 0 表示停止，1 表示运行，而指示灯绿灯是 1，红灯是 3，故需要用表达式 IF 进行转换，表达式为 IF(Value[0]=1,1,3)，即变量值等于 1 时返回 1，否则返回 3。），在对话框的变量列表中添加变量“电机变量/X 轴状态/Run”，最后在表达式文本框中填入表达式即可。





6. 总结

随着产品复杂度越来越高，开发周期越来越短，市场竞争越来越激烈，自主研发测试软件越来越不能满足企业的需求。现代测控软件开发环境克服了许多自主研发软件的限制。 Carnegie Mellon SEI3 关于标准化测控软件开发过程的调查显示，测控软件开发过程的标准化使得：

- 产品上市时间缩短 38%
- 工作量减轻了 76%
- 产品质量提高了 80%；降低发货不合格率

格西测控大师作为一款现代测控软件开发环境，基于模块化技术的测控开发管理软件，可帮助您快速开发自动化测试和控制软件系统，可帮助企业统一化、标准化测试和控制软件的开发和管理，减少企业在测试和控制软件方面的开发成本、学习成本和维护成本！