第 8 章“组态，为王”

应用软件工具之一

8.1 组态软件简介[1]

组态软件，又称组态监控系统软件。译自英文 SCADA,即 Supervisory Control and Data Acquisition（数据采集与监视控制）。它是指一些数据采集与过程控制的专用软件。它们处在自动控制系统监控层一级的软件平台和开发环境，使用灵活的组态方式，为用户提供快速构建工业自动控制系统监控功能的、通用层次的软件工具。组态软件的应用领域很广，可以应用于电力系统、给水系统、石油、化工等领域的数据采集与监视控制以及过程控制等诸多领域。

组态软件在国内是一个约定俗成的概念，并没有明确的定义，它可以理解为“组态式监控软件”。“组态（Configure）”的含义是“配置”、“设定”、“设置”等意思，是指用户通过类似“搭积木”的简单方式来完成自己所需要的软件功能，而不需要编写计算机程序，也就是所谓的“组态”。它有时候也称为“二次开发”，组态软件就称为“二次开发平台”。“监控（Supervisory Control）”，即“监视和控制”，是指通过计算机信号对自动化设备或过程进行监视、控制和管理。

组态软件是有专业性的。一种组态软件只能适合某种领域的应用。组态的概念最早出现在工业计算机控制中，如：DCS(采集控制系统）组态、PLC（可编程控制器）梯形图组态；人机界面生成软件就叫工控组态软件。在其他行业也有组态的概念，如 AutoCAD，PhotoShop 等。不同之处在于，工业控制中形成的组态结果是用在实时监控的。从表面上看，组态工具的运行程序就是执行自己特定的任务。工控组态软件也提供了编程手段，一般都是内置编译系统，提供类 BASIC 语言，有的支持 VB，现在有的组态软件甚至支持 C#高级语言。

组态软件大都支持各种主流工控设备和标准通信协议，并且通常应提供分布式数据管理和网络功能。对应于原有的 HMI（人机接口软件，Human Machine Interface）的概念，组态软件还是一个使用户能快速建立自己的 HMI 的软件工具或开发环境。在组态软件出现之前，工控领域的用户通过手工或委托第三方编写 HMI 应用，开发时间长，效率低，可靠性差；或者购买专用的工控系统，通常是封闭的系统，选择余地小，往往不能满足需求，很难与外界进行数据交互，升级和增加功能都受到严重的限制。组态软件的出现使用户可以利用组态软件的功能，构建一套最适合自己的应用系统。随着它的快速发展，实时数据库、实时控制、SCADA、通讯及联网、开放数据接口、对 I/O 设备的广泛支持已经成为它的主要内容，监控组态软件将会

不断被赋予新的内容。

“组态”的概念是伴随着集散型控制系统（Distributed Control System 简称 DCS）的出现才开始被广大的生产过程自动化技术人员所熟知的。在工业控制技术不断发展和应用的过程中，PC（包括工控机）相比以前的专用系统具有的优势日趋明显。这些优势主要体现在：PC 技术保持了较快的发展速度，各种相关技术已经成熟；由 PC 构建的工业控制系统具有相对较低的拥有成本；PC 的软件资源和硬件资源丰富，软件之间的互操作性强；基于 PC 的控制系统易于学习和使用，可以容易地得到技术方面的支持。在 PC 技术向工业控制领域的渗透中，组态软件占据着非常特殊而且重要的地位。

常见的国外组态软件主要有：1.InTouch 2.IFix 3.Citech 4.WinCC 5.ASPEN-tech 6.Movicon 7.GENESIS 64 其中最早进入国内的是 InTouch，也是最具代表性的组态软件之一。

国内常用的组态软件有：1.世纪星 2.三维力控 3.组态王 KingView 4.紫金桥 Realinfo 5.MCGS 6.态神 7.uScada 8.Controx（开物）9.E-Form++组态源码解决方案 10.iCentroView 11.QTouch 其中组态王 KingView 的市场占有率最高。

随着工业自动化水平的迅速提高，计算机在工业领域的广泛应用，人们对工业自动化的要求越来越高，种类繁多的控制设备和过程监控装置在工业领域的应用，使得传统的工业控制软件已无法满足用户的各种需求。在开发传统的工业控制软件时，当工业被控对象一旦有变动，就必须修改其控制系统的源程序，导致其开发周期长；已开发成功的工控软件又由于每个控制项目的不同而使其重复使用率很低，导致它的价格非常昂贵；在修改工控软件的源程序时，倘若原来的编程人员因工作变动而离去时，则必须同其他人员或新手进行源程序的修改，因而更是相当困难。通用工业自动化组态软件的出现为解决上述实际工程问题提供了一种崭新的方法，因为它能够很好地解决传统工业控制软件存在的种种问题，使用户能根据自己的控制对象和控制目的的任意组态，完成最终的自动化控制工程。

组态（Configuration）为模块化任意组合。通用组态软件主要特点：

（1）延续性和可扩充性。用通用组态软件开发的应用程序，当现场（包括硬件设备或系统结构）或用户需求发生改变时，不需作很多修改而方便地完成软件的更新和升级；

（2）封装性（易学易用），通用组态软件所能完成的功能都用一种方便用户使用的方法包装起来，对于用户，不需掌握太多的编程语言技术（甚至不需要编程技术），就能很好地完成一个复杂工程所要求的所有功能；

（3）通用性，每个用户根据工程实际情况，利用通用组态软件提供的底层设备（PLC、智能仪表、智能模块、板卡、变频器等）的 I/O Driver、开放式的数据库和画面制作工具，就能完成一个具有动画效果、实时数据处理、历史数据和曲线并存、具有多媒体功能和网络功能的工程，不受行业限制。

组态软件指一些数据采集与过程控制的专用软件，它们是在自动控制系统监控层一级的软件平台和开发环境，能以灵活多样的组态方式（而不是编程方式）提供良好的 用户开发界面和简捷的使用方法，它解决了控制系统通用性问题。其预设置的各种软件模块可以非常容易地实现和完成监控层的各项功能，并能同时支持各种硬件厂 家的计算机和 I/O 产品，与高可靠的工控计算机和网络系统结合，可向控制层和管理层提供软硬件的全部接口，进行系统集成。

组态软件通常有以下几方面的功能：

（1）强大的界面显示组态功能。目前，工控组态软件大都运行于 Windows 环境下，充分利用 Windows 的图形功能完善界面美观的特点，可视化的 m 风格界面、丰富的工具栏，操作人员可以直接进人开发状态，节省时间。丰富的图形控件和工况图库，既提供所需的组件，又 是界面制作向导。提供给用户丰富的作图工具，可随心所欲地绘制出各种工业界面，并可任意编辑，从而将开发人员从繁重的界面设计中解放出来，丰富的动画连接 方式，如隐含、闪烁、移动等等，使界面生动、直观。

（2）良好的开放性。社会化的大生产，使得系统构成的全部软硬件不可能出自一家公司的产品，“异构”是当今控制系统的主要特点之一。开放性是指组态软件能与多种通信协议互联，支持多种硬件设备。开放性是衡量一个组态软件好坏的重要指标。组态软件向下应能与低层的数据采集设备通信，向上能与管理层通信，实现上位机与下位机的双向通信。

（3） 丰富的功能模块。提供丰富的控澜功能库，满足用户的测控要求和现场荽求。利用各种功能模块，完成实时监控 产生功能报表 业示历史曲线、实时曲线、提侠报警等功能，使系统具有良好的人机界面，易于操作，系统既叫适用于单机集中式控制、DCS 分布式控制，也可以是带远程遇信能 力的远程测控系统．

（4）强大的数据库。配有实时数据库，可存储各种数据，如模拟量、离散童、字符型等，实现与外部设备的数据交换。

（5）可编程的命令语言。有可编程的命令语言，使用户可根据自己的需要编鸾程序，蹭强图形界面

（6）周密的系统安全防范，对不同的操作者，赋予不同的操作权眼，保证整个系统的安全可靠运行。

（7）仿真功能．捉供强大的仿真功能使系统并行设计，从而缩短开发周期。

8.2 组态软件与 CAN 总线——强强结合

一方面组态软件得到了广泛的应用，另一方面 CAN 现场总线成本降低而逐步普及，使其有能力替代传统的 485＋modbus 应用模式，因此将 CAN 现场总线融入组态软件中是发展的大趋势，其对应用系统的性能提升起着非常积极的作用，是强强结合的产物。

然而，现实却与我们美好的愿望存在着很大差距，由于 CAN 应用层协议中的两个巨无霸 CANopen 和 deviceNet 解析起来非常困难，使组态软件中相应的“驱动”也更加难于实现。目前国内主流的组态软件对 CANopen 和 deviceNet 都没有有效的支持，国外组态软件有限的部分支持 CANopen 和 deviceNet 协议，且价格不菲。这给组态软件与 CAN 现场总线的结合带来了巨大的屏障。

TTCANopen 应用层协议的出现使我们看到了曙光。我们知道所有的组态软件对串口和 Modbus 的支持是与生俱来的，由于 TTCANopen 应用层协议的直读性和其类 Modbus 特性以及“转换模块”的“虚拟串口”特性，使 TTCANopen 的解析和组态软件驱动的开发的难度与 Modbus 站在了同一“起跑线”上，由此铺平了组态软件与 CAN 现场总线结合的道路，大

凡能够开发串口设备、Modbus 驱动的程序员便可非常容易的开发出 TTCANopen 驱动。

为了给用户提供一个样例，起着抛砖引玉的作用，我们聘请了北京中铁航机电公司的工程师，尝试开发了一个“组态王”组态软件驱动。

8.3 组态软件驱动程序的开发

组态软件驱动主要分为“设备驱动”和“协议驱动”。“设备驱动”是针对某些具体设备开发的驱动程序，它只对这些设备有效；“协议驱动”是针对某种通用协议编写的驱动，凡是符合协议规范的设备都会被组态软件所支持，如“Modbus”驱动，我们要开发的样例驱动属于后者，是针对 TTCANopen 初级篇中的协议规范的。

组态软件驱动与我们通常所说的系统设备驱动是不同的。一般“通讯设备”首先是通过系统设备驱动完成其硬件与操作系统之间的交互关系，再上面一层才是组态软件驱动，完成通讯数据的解析使其能够被组态软件所识别和使用。确切的说组态软件驱动是一个“协议解析器”，它将底层通讯设备按某种协议传输的数据解析为组态软件的特征变量，反之亦然。

在开发组态软件驱动前，我们给国内几个主要组态软件开发商打通了电话，索取其组态软件驱动开发包，其中，只有北京亚控公司发来了“组态王”的驱动开发包，还包括了一个长达 2 小时的视频开发教程及相关文档，这对我们的帮助很大。这是我们第一次组织开发组态软件驱动程序，必然存在很多的经验不足和偏差，在这里我们将开发的主要过程展现给用户，也是一个相互学习的过程，当然我们更希望将来的驱动不是由我们来开发，而是由各组态软件公司开发，因为只有生产商才最了解自己产品的内核，开发出来的驱动才会更加稳定。

8.3.1 组态王驱动开发环境

操作系统	windows xp（sp3）
编程语言	visual studio.net 2003
开发软件包	组态王驱动开发包 3.0.0.7（中文）
面向协议	TTCANopen 初级协议

8.3.2 安装组态王开发软件包

在安装组态王驱动开发软件包之前，我们应当安装好 visual studio.net 2003，然后运行“组态王驱动开发包 3.0.0.7（中文）.exe”见图 8－1，然后点击“下一步”选择安装目标文件

夹见图 8－2，安装完成。组态王驱动开发工具能够自动生成驱动代码框架。

图 8－1 启动安装组态王驱动开发包

图 8－2 选择安装目标文件夹

8.3.3 生成基本的驱动程序代码

安装完组态王驱动开发包后，就可以启动 visual studio.net 2003，生成基本的驱动程序框架。其具体步骤如下：

（1）启动 visual studio.net 2003；

（2）新建一个工程

“项目类型”选择“Visual C++项目”，

“模板”选择 “KingView Wizard”，

在下面的编辑框中输入工程的名字以及存储路径，

点击“确定”；

（3）生成一个驱动创建向导，该向导分为三页，第一页见图 8－3，是一个简要说明，没有任何选项和输入内容；

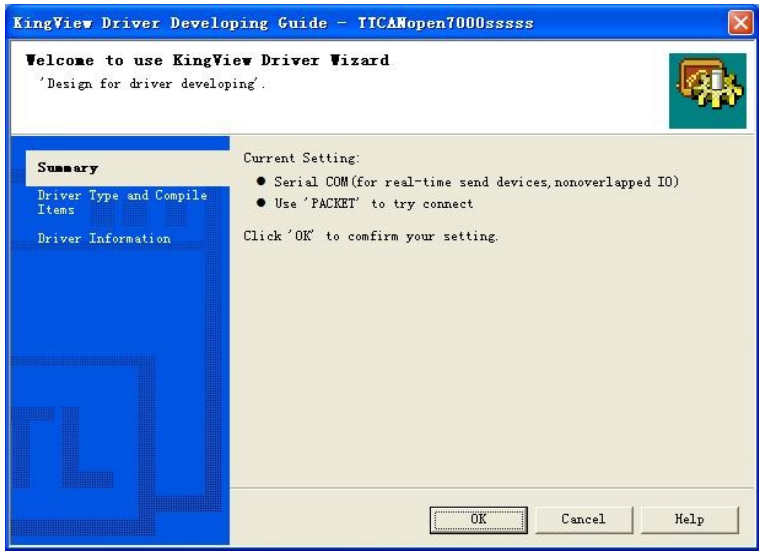

图 8－3 驱动简要说明

第二页见图 8－4，是驱动类型和编译选项，其中驱动类型有两个串口可选项和两个网络可选项，在这里我们选择 Serial COM(for real-time send devices, nonoverlapped IO)这种方式，其适用于下位设备实时上发数据的情况，驱动中需要创建一个线程来监视串口事件的通讯类型，采用非重叠 IO 方式，该方式能够支持 TTCANopen 中的事件触发和主动上报信息模式，而不

仅仅是轮询这种简单方式。

设备名称：默认的设备名称是“Name1”，可以自由填写，但须注意应该与设备列表中的设备名称保持一致，这里我们使用“TTCANopen7000sssss”这个名字。

尝试连接：默认选用 use“PACKET”即可。

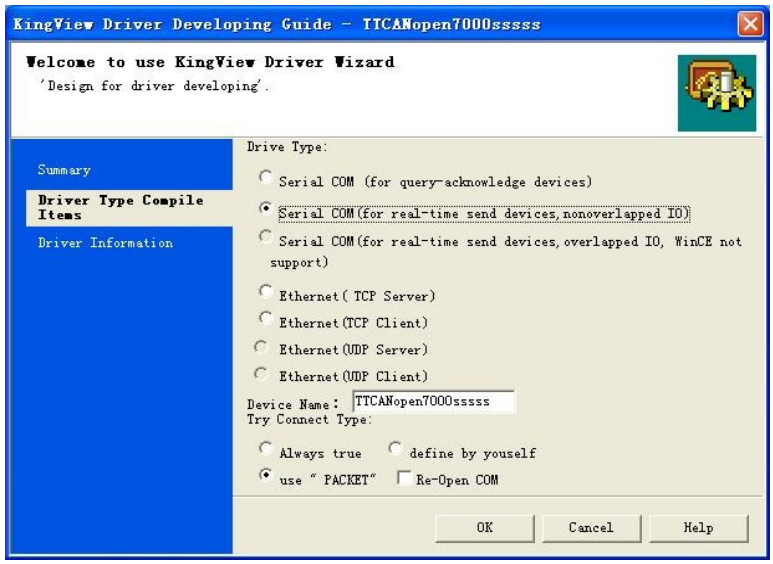

图 8－4 驱动类型和编译选项

第三页见图 8－5，是驱动基本信息，主要包括：驱动名称、版本、描述、程序员和 项目经理；其他支持：USB 通讯和 ADO 数据库操作；

点击“OK”即生成了基本的驱动框架，在这个框架中包含了许多的文件见图 8－6 中所示，其中“DevTTCANopen7000sssss.cpp”这个文件非常重要，我们编写的驱动程序需要“填充”和“修改”的 80%的内容都在这个文件中完成。

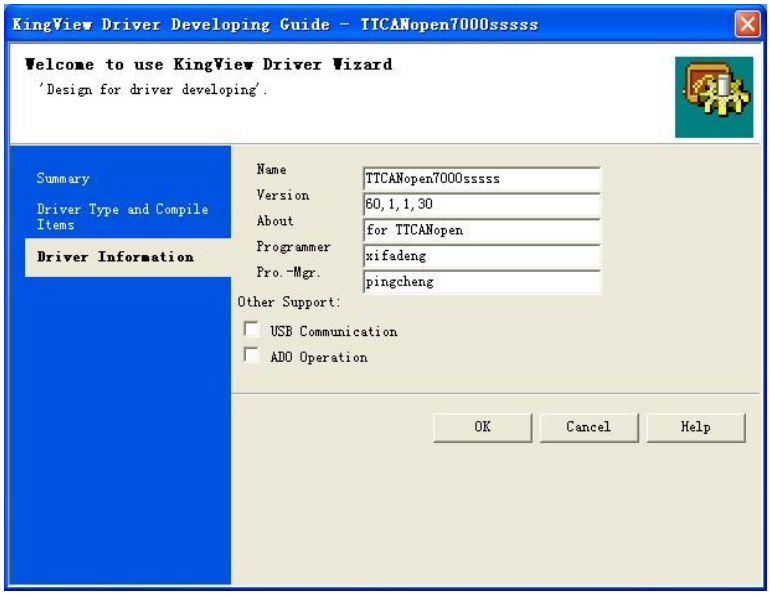

图 8－5 驱动基本信息

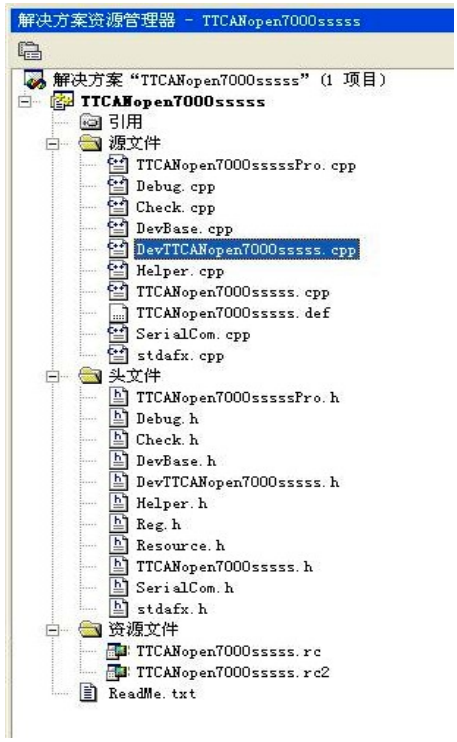

图 8－6 驱动框架中的文件

8.3.4 DevTTCANopen7000sssss.cpp 文件的“填充”和“修改”

（1）组态软件中注册变量

在组态软件中根据应用的不同会用到各种变量，如：“位”变量、“字节”变量、“整数”变量、“长整数”变量和“浮点”变量等，在 DevTTCANopen7000sssss.cpp 文件中专门有一个数据结构用于定义用户用到的变量类型、寄存器范围和变量属性（读/写）见图 8－7。

```
static REG_INFO gsRegInfo[]=
{
//add your code  here, the following for your reference
    {_T("FLOW"), 0, 0, FLOAT_DATATYPE, PT_READ },       // the current flow register, read only
    {_T("PARA"), 0, 2, FLOAT_DATATYPE, PT_READ }        //  parameter register, read only

};
```

图 8－7 自动生成的用户变量结构列表

在上面自动生成的用户变量结构列表中只有两个变量类型，用户可以根据自己的应用环境进行添加或修改，图 8－8 是我们根据 TTCANopen 协议工程应用修改后的列表。

在这个结构中我们定义了一些种类的变量，下面给出这些变量的一些具体说明：

① 双引号“ ”中的内容是各种变量寄存器名的前缀，后跟一个寄存器地址值，由于在 TTCANopen 协议中所有的变量都放在一个十六位寻址的寄存器序列中，因此这里我们习惯性的使用十六进制数表示这个地址（在所有变量名前缀的后端为_0x 是为了提醒用户后面跟着的应当是一个十六进制地址，如：SI_0x81C0 等。）。

② “ ”内_0x 前的字母含义：变量的操作属性

I － 只读变量，组态王读取设备变量，主要是数据采集（轮询读）；

O － 读写变量，组态王可对该设备变量进行读或写操作（轮询读，单次写）；

W － 只写变量，组态王对该设备变量只能进行写操作（单次）；

G － 设备主动上报变量，组态王被动接收（由设备触发或周期发送）；

T － 设备主动请求的数据，组态王对其进行响应填充（“索取/输送”方式）；

以上是常用的五种变量“访问”属性，另外我们还定义了三种用于位操作的属性，

Y－ 位“与”变量；

H－ 位“或”变量；

F－ 位“非”变量；

这三种属性的变量，其使用方法比较特殊，这里暂不进行介绍。

```
static REG_INFO gsRegInfo[]=
{
//add your code  here, the following for your reference
    {_T("BI_0x"),    0x8000, 0x9FFF, BIT_DATATYPE, PT_READ},          //寄存器下限为0x8000   上限为0x9FFF
    {_T("BO_0x"),    0x8000, 0x9FFF, BIT_DATATYPE, PT_READWRITE},     //寄存器下限为0x8000   上限为0x9FFF
    {_T("BW_0x"),    0x8000, 0x9FFF, BIT_DATATYPE, PT_WRITE},         //寄存器下限为0x8000   上限为0x9FFF
    {_T("BG_0x"),    0x8000, 0x9FFF, BIT_DATATYPE, PT_READ},          //寄存器下限为0x8000   上限为0x9FFF
    {_T("BT_0x"),    0x8000, 0x9FFF, BIT_DATATYPE, PT_WRITE},         //寄存器下限为0x8000   上限为0x9FFF

    {_T("B8I_0x"),   0x8000, 0x9FFF, BYTE_DATATYPE, PT_READ},         //寄存器下限为0x8000   上限为0x9FFF
    {_T("B8O_0x"),   0x8000, 0x9FFF, BYTE_DATATYPE, PT_READWRITE},    //寄存器下限为0x8000   上限为0x9FFF
    {_T("B8W_0x"),   0x8000, 0x9FFF, BYTE_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF
    {_T("B8G_0x"),   0x8000, 0x9FFF, BYTE_DATATYPE, PT_READ},         //寄存器下限为0x8000   上限为0x9FFF
    {_T("B8T_0x"),   0x8000, 0x9FFF, BYTE_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF

    {_T("CI_0x"),    0x8000, 0x9FFF, BYTE_DATATYPE, PT_READ},         //寄存器下限为0x8000   上限为0x9FFF
    {_T("CO_0x"),    0x8000, 0x9FFF, BYTE_DATATYPE, PT_READWRITE},    //寄存器下限为0x8000   上限为0x9FFF
    {_T("CW_0x"),    0x8000, 0x9FFF, BYTE_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF
    {_T("CG_0x"),    0x8000, 0x9FFF, BYTE_DATATYPE, PT_READ},         //寄存器下限为0x8000   上限为0x9FFF
    {_T("CT_0x"),    0x8000, 0x9FFF, BYTE_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF

    {_T("KCI_0x"),   0xF000, 0xFFFF, BYTE_DATATYPE, PT_READ},         //控制用寄存器0xF000   至0xFFFF
    {_T("KCO_0x"),   0xF000, 0xFFFF, BYTE_DATATYPE, PT_READWRITE},    //控制用寄存器0xF000   至0xFFFF
    {_T("KCW_0x"),   0xF000, 0xFFFF, BYTE_DATATYPE, PT_WRITE},        //控制用寄存器0xF000   至0xFFFF
    {_T("KCG_0x"),   0xF000, 0xFFFF, BYTE_DATATYPE, PT_READ},         //控制用寄存器0xF000   至0xFFFF
    {_T("KCT_0x"),   0xF000, 0xFFFF, BYTE_DATATYPE, PT_WRITE},        //控制用寄存器0xF000   至0xFFFF

    {_T("SI_0x"),    0x8000, 0x9FFE, INT_DATATYPE, PT_READ},          //寄存器下限为0x8000  上限为0x9FFF-1
    {_T("SO_0x"),    0x8000, 0x9FFE, INT_DATATYPE, PT_READWRITE},     //寄存器下限为0x8000  上限为0x9FFF-1
    {_T("SW_0x"),    0x8000, 0x9FFE, INT_DATATYPE, PT_WRITE},         //寄存器下限为0x8000  上限为0x9FFF-1
    {_T("SG_0x"),    0x8000, 0x9FFE, INT_DATATYPE, PT_READ},          //寄存器下限为0x8000  上限为0x9FFF-1
    {_T("ST_0x"),    0x8000, 0x9FFE, INT_DATATYPE, PT_WRITE},         //寄存器下限为0x8000  上限为0x9FFF-1

    {_T("USI_0x"),   0x8000, 0x9FFE, UINT_DATATYPE, PT_READ},         //寄存器下限为0x8000   上限为0x9FFF-1
    {_T("USO_0x"),   0x8000, 0x9FFE, UINT_DATATYPE, PT_READWRITE},    //寄存器下限为0x8000   上限为0x9FFF-1
    {_T("USW_0x"),   0x8000, 0x9FFE, UINT_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF-1
    {_T("USG_0x"),   0x8000, 0x9FFE, UINT_DATATYPE, PT_READ},         //寄存器下限为0x8000   上限为0x9FFF-1
    {_T("UST_0x"),   0x8000, 0x9FFE, UINT_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF-1

    {_T("KUSI_0x"),  0xF000, 0xFFFE, UINT_DATATYPE, PT_READ},         //控制用寄存器 0xF000   至0xFFFE
    {_T("KUSO_0x"),  0xF000, 0xFFFE, UINT_DATATYPE, PT_READWRITE},    //控制用寄存器 0xF000   至0xFFFE
    {_T("KUSW_0x"),  0xF000, 0xFFFE, UINT_DATATYPE, PT_WRITE},        //控制用寄存器 0xF000   至0xFFFE
    {_T("KUSG_0x"),  0xF000, 0xFFFE, UINT_DATATYPE, PT_READ},         //控制用寄存器 0xF000   至0xFFFE
    {_T("KUST_0x"),  0xF000, 0xFFFE, UINT_DATATYPE, PT_WRITE},        //控制用寄存器 0xF000   至0xFFFE

    {_T("BCDI_0x"),  0x8000, 0x9FFE, BCD_DATATYPE, PT_READ},          //寄存器下限为0x8000  上限为0x9FFF-1
    {_T("BCDO_0x"),  0x8000, 0x9FFE, BCD_DATATYPE, PT_READWRITE},     //寄存器下限为0x8000  上限为0x9FFF-1
    {_T("BCDW_0x"),  0x8000, 0x9FFE, BCD_DATATYPE, PT_WRITE},         //寄存器下限为0x8000  上限为0x9FFF-1
    {_T("BCDG_0x"),  0x8000, 0x9FFE, BCD_DATATYPE, PT_READ},          //寄存器下限为0x8000  上限为0x9FFF-1
    {_T("BCDT_0x"),  0x8000, 0x9FFE, BCD_DATATYPE, PT_WRITE},         //寄存器下限为0x8000  上限为0x9FFF-1

    {_T("LI_0x"),    0x8000, 0x9FFC, LONG_DATATYPE, PT_READ},         //寄存器下限为0x8000  上限为0x9FFF-3
    {_T("LO_0x"),    0x8000, 0x9FFC, LONG_DATATYPE, PT_READWRITE},    //寄存器下限为0x8000  上限为0x9FFF-3
    {_T("LW_0x"),    0x8000, 0x9FFC, LONG_DATATYPE, PT_WRITE},        //寄存器下限为0x8000  上限为0x9FFF-3
    {_T("LG_0x"),    0x8000, 0x9FFC, LONG_DATATYPE, PT_READ},         //寄存器下限为0x8000  上限为0x9FFF-3
    {_T("LT_0x"),    0x8000, 0x9FFC, LONG_DATATYPE, PT_WRITE},        //寄存器下限为0x8000  上限为0x9FFF-3

    {_T("LBCDIOx"),  0x8000, 0x9FFC, LONGBCD_DATATYPE, PT_READ},      //寄存器下限为0x8000   上限为0x9FFF-3
    {_T("LBCDOOx"),  0x8000, 0x9FFC, LONGBCD_DATATYPE, PT_READWRITE},//寄存器下限为0x8000   上限为0x9FFF-3
    {_T("LBCDWOx"),  0x8000, 0x9FFC, LONGBCD_DATATYPE, PT_WRITE},     //寄存器下限为0x8000   上限为0x9FFF-3
    {_T("LBCDGOx"),  0x8000, 0x9FFC, LONGBCD_DATATYPE, PT_READ},      //寄存器下限为0x8000   上限为0x9FFF-3
    {_T("LBCDTOx"),  0x8000, 0x9FFC, LONGBCD_DATATYPE, PT_WRITE},     //寄存器下限为0x8000   上限为0x9FFF-3

    {_T("FI_0x"),    0x8000, 0x9FFC, FLOAT_DATATYPE, PT_READ},        //寄存器下限为0x8000   上限为0x9FFF-3
    {_T("FO_0x"),    0x8000, 0x9FFC, FLOAT_DATATYPE, PT_READWRITE},   //寄存器下限为0x8000   上限为0x9FFF-3
    {_T("FW_0x"),    0x8000, 0x9FFC, FLOAT_DATATYPE, PT_WRITE},       //寄存器下限为0x8000   上限为0x9FFF-3
    {_T("FG_0x"),    0x8000, 0x9FFC, FLOAT_DATATYPE, PT_READ},        //寄存器下限为0x8000   上限为0x9FFF-3
    {_T("FT_0x"),    0x8000, 0x9FFC, FLOAT_DATATYPE, PT_WRITE},       //寄存器下限为0x8000   上限为0x9FFF-3


    {_T("CY_0x"),    0x8000, 0x9FFF, BYTE_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF    专用于“位与”操作
    {_T("CH_0x"),    0x8000, 0x9FFF, BYTE_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF    专用于“位或”操作
    {_T("CF_0x"),    0x8000, 0x9FFF, BYTE_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF    专用于“位非”操作

    {_T("USY_0x"),   0x8000, 0x9FFE, UINT_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF-1   专用于“位与”操作
    {_T("USH_0x"),   0x8000, 0x9FFE, UINT_DATATYPE, PT_WRITE},        //寄存器下限为0x8000   上限为0x9FFF-1   专用于“位或”操作
    {_T("USF_0x"),   0x8000, 0x9FFE, UINT_DATATYPE, PT_WRITE}         //寄存器下限为0x8000   上限为0x9FFF-1   专用于“位非”操作

};
```

图 8－8 修改后的用户变量结构列表

③ “ ”内操作属性前面字母含义：变量的数值类型和长度

B — 位变量（1 位）；

B8 — 位变量，以字节为操作单位（1＊8 位）；

C — 字节变量（8 位）；

KC — 字节变量，仅用于 TTCANopen 协议的设备标识空间（8 位）；

S — 整数变量（16 位）；

US — 无符合整数（16 位）；

KUS — 无符合整数，仅用于 TTCANopen 协议的设备标识空间（16 位）；

BCD — BCD 计数方式（16 位）；

L — 长整型（32 位）；

LBCD— 长 BCD 计数方式（32 位）；

F — 浮点数（32 位）；

其中： B8、 BCD、 LBCD、 KUS 用的不多。

④ 后面三个字段分别为“可用寄存器起始地址”、“可用寄存器终止地址”和“变量读写属性”。TTCANopen 初级协议中定义 0x8000～0x8FFF 为过程交换变量的存储空间，0x9000～0x9FFF 为设备工作参数存储空间，0xF000～0xFFFF 为设备标识存储空间。

（2）索引注册变量

仅仅注册变量还不够，还要为这些变量类型提供索引，方能被驱动程序识别，图 8－9 是自动生成的用户变量索引，它为每一种变量类型提供了一个唯一的索引编号。

```
//Register variable define, add your code  here, the following for your reference
#define     FLOW_REG   0
#define     PARA_REG   1
```

图 8－9 自动生成的用户变量结构列表

图 8－10 为修改后的变量类型索引，一共定义了 60 个变量类型的索引。

```
//Register variable define, add your code  here, the following for your reference
#define     REG_TYPE_BI             0           //访问单个开关量
#define     REG_TYPE_BO             1           //访问单个开关量
#define     REG_TYPE_BW             2           //访问单个开关量
#define     REG_TYPE_BG             3           //访问单个开关量
#define     REG_TYPE_BT             4           //访问单个开关量

#define     REG_TYPE_B8I            5           //访问8个一组开关量，用TTCANOPEN 位“与”“或”“非”指令实现
#define     REG_TYPE_B8O            6           //访问8个一组开关量，用TTCANOPEN 位“与”“或”“非”指令实现
#define     REG_TYPE_B8W            7           //访问8个一组开关量，用TTCANOPEN 位“与”“或”“非”指令实现
#define     REG_TYPE_B8G            8           //访问8个一组开关量，用TTCANOPEN 位“与”“或”“非”指令实现
#define     REG_TYPE_B8T            9           //访问8个一组开关量，用TTCANOPEN 位“与”“或”“非”指令实现

#define     REG_TYPE_CI             10
#define     REG_TYPE_CO             11
#define     REG_TYPE_CW             12
#define     REG_TYPE_CG             13
#define     REG_TYPE_CT             14

#define     REG_TYPE_KCI            15          //用于控制寄存器
#define     REG_TYPE_KCO            16          //用于控制寄存器
#define     REG_TYPE_KCW            17          //用于控制寄存器
#define     REG_TYPE_KCG            18          //用于控制寄存器
#define     REG_TYPE_KCT            19          //用于控制寄存器

#define     REG_TYPE_SI             20
#define     REG_TYPE_SO             21
#define     REG_TYPE_SW             22
#define     REG_TYPE_SG             23
#define     REG_TYPE_ST             24

#define     REG_TYPE_USI            25
#define     REG_TYPE_USO            26
#define     REG_TYPE_USW            27
#define     REG_TYPE_USG            28
#define     REG_TYPE_UST            29
```

```
#define     REG_TYPE_KUSI           30
#define     REG_TYPE_KUSO           31
#define     REG_TYPE_KUSW           32
#define     REG_TYPE_KUSG           33
#define     REG_TYPE_KUST           34

#define     REG_TYPE_BCDI           35
#define     REG_TYPE_BCDO           36
#define     REG_TYPE_BCDW           37
#define     REG_TYPE_BCDG           38
#define     REG_TYPE_BCDT           39

#define     REG_TYPE_LI             40
#define     REG_TYPE_LO             41
#define     REG_TYPE_LW             42
#define     REG_TYPE_LG             43
#define     REG_TYPE_LT             44

#define     REG_TYPE_LBCDI          45
#define     REG_TYPE_LBCDO          46
#define     REG_TYPE_LBCDW          47
#define     REG_TYPE_LBCDG          48
#define     REG_TYPE_LBCDT          49

#define     REG_TYPE_FI             50
#define     REG_TYPE_FO             51
#define     REG_TYPE_FW             52
#define     REG_TYPE_FG             53
#define     REG_TYPE_FT             54

#define     REG_TYPE_CY             55          //专用于“位与”操作
#define     REG_TYPE_CH             56          //专用于“位或”操作
#define     REG_TYPE_CF             57          //专用于“位非”操作

#define     REG_TYPE_USY            58          //专用于“位与”操作
#define     REG_TYPE_USH            59          //专用于“位或”操作
#define     REG_TYPE_USF            60          //专用于“位非”操作
//(注意：位指令是通过定义四个指向相同的寄存器变量实现的，Y为与参数，K为或参数，F为非参数)
//(用G“设备主动上报只读变量”读取，位操作后的结果，虽然位操作在TTCANOPNE中是一个高效的操作)
//(但在组态王的实现中却打了折扣，即浪费了变量资源，又损失了部分事实性)
```

图 8－10 修改后的注册变量类型索引

在我们编译安装完驱动后，建立工程，并运行组态王开发环境，在数据词典中选择新建变量，出现的定义变量对话框，其中的“寄存器”选项下拉菜单中将会显示这 60 种变量见图 8－11，选择一种变量然后添加寄存器地址如：SI_0x81C0 见图 8－12。

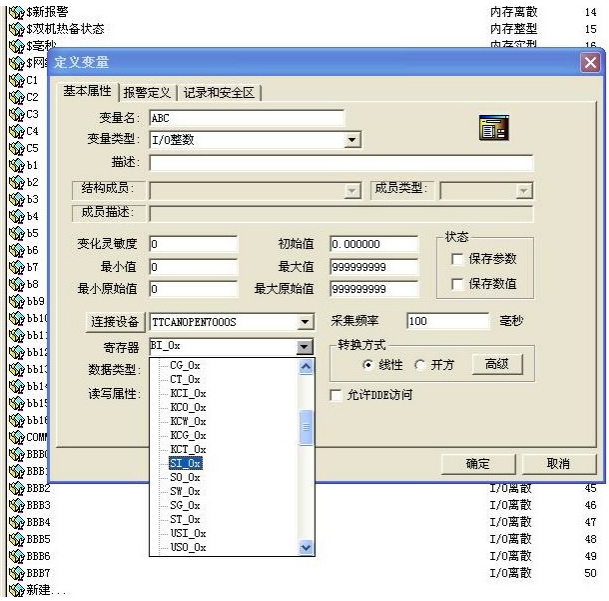

图 8－11 定义变量对话框寄存器选项下拉菜单

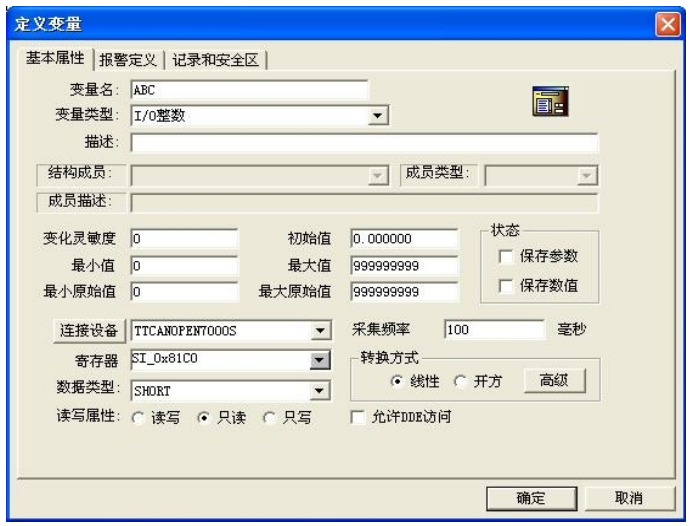

图 8－12 定义变量对话框

（3）修改 ConvertUserConfigToVar( )函数

该函数主要是对“定义变量对话框”中的变量输入的合法性进行检查，如：输入变量寄存器字符串长度检查、输入变量寄存器字符合法性检查、输入变量寄存器地址越界检查等。如果检查出错误将会弹出“错误提示对话框”。图 8－13 是我们修改后的函数，由于我们定义变量寄存器地址使用十六进制表示，因此我们在函数中定义“0123456789ABCDEF”这十六个字符为合法字符，输入其他字符将提示出错，对于“位”变量我们也给出了相关的界定。

```
WORD CDevTTCANopen7000ssssss::ConvertUserConfigToVar( LPVOID lpDbItemItem, LPVOID lpVar)  //
{

    ASSERT(lpDbItemItem != NULL);
    ASSERT(lpVar != NULL);
                LPVOID lpVar
    MiniDbItem * pDbItem = (MiniDbItem*)lpDbItemItem;
    PPLCVAR pPlcVar = (PPLCVAR) lpVar;
    /*
    Notes:
    This function is to convert the variable string to KINGVIEW's structural variables
    we advise you not to modify for common drivers.
    */

    REG_INFO * pRegInfo = NULL;
    int nRegNum = 0 ;
    GetRegisters( (LPVOID*)&pRegInfo, &nRegNum );
    //Initialization
    CString szRegister(pDbItem->szRegister);
    //Delete blank characters
    szRegister.TrimLeft();
    szRegister.TrimRight();
    if (szRegister.IsEmpty())
    {
        m_pPro->m_nLastErrorCode = ERR_REG_NAME;
        return 1;
    }
    //Not to access the Maximum length supported.
    if (szRegister.GetLength() > MAX_CONFIG_STRING_LENGTH)
    {
        //m_pPro->m_nLastErrorCode = ERR_CONFIG_MAXLEN;
        m_pPro->m_nLastErrorCode = ERR_REG_NAME;
        return 1;
    }

    //if found
    BOOL bMatched = FALSE;
    //get the index number of register
    int  nRegIndex = 0;
    do
    {
        //if found the register string
        if (szRegister.Find(pRegInfo[nRegIndex].sRegName) == 0 )
        {
            bMatched = TRUE;
        }

    }while (!bMatched && ++nRegIndex < nRegNum );

    if (!bMatched)
    {
        m_pPro->m_nLastErrorCode = ERR_REG_NAME;
        return 1;
    }
```

```
    //get channel string
    CString szChannelConfig = szRegister.Right(szRegister.GetLength() \
        - (int)_tcslen(pRegInfo[nRegIndex].sRegName));

    int szChannelConfig_long = szChannelConfig.GetLength();

    if( (szChannelConfig_long  != 4) && (szChannelConfig_long != 6) )
    {
        m_pPro->m_nLastErrorCode = ERR_REG_NAME;
        return 1;
    }

    if( (szChannelConfig_long == 4) && (pRegInfo[nRegIndex].wDataType == BIT_DATATYPE))   ////add by dxf  ++
    {
        m_pPro->m_nLastErrorCode = ERR_REG_NAME;
        return 1;
    }
```

```
    //check the string format
    if(pRegInfo[nRegIndex].nLowIndex == pRegInfo[nRegIndex].nUpperIndex)
    {
        if (pRegInfo[nRegIndex].nLowIndex != 0)
        {
            if(szChannelConfig.IsEmpty())
            {
                m_pPro->m_nLastErrorCode = ERR_REG_CH;
                return 1;
            }
        }
    }

    //get the channel(offset) number
//  int nNo = _ttoi(szChannelConfig);

    BYTE  bM[5];

    bM[0] = szChannelConfig.GetAt(2);
    bM[1] = szChannelConfig.GetAt(3);
    bM[2] = szChannelConfig.GetAt(0);
    bM[3] = szChannelConfig.GetAt(1);

    int nNo =  CHelper::ASCToWord( bM );
```

```
    if(szChannelConfig_long == 6)
    {
      if(szChannelConfig.GetAt(4) != '.' || szChannelConfig.GetAt(5) < '0'    || szChannelConfig.GetAt(5) > '7' )
      {
            m_pPro->m_nLastErrorCode = ERR_REG_NAME;
            return 1;
      }
      else
      {
        pPlcVar -> nSubType1 = szChannelConfig.GetAt(5) - 0x30;  //该信息可在ID_NO2->BYTE Special[0]中读取
      }
    }

    //Allowed characters : 0123456789ABCDEF.:
//  CString szAllow(_T("0123456789ABCDEF.:"));
    CString szAllow(_T("0123456789ABCDEF"));


    //Check all characters.
//  for (int nIndex = 0; nIndex < szChannelConfig.GetLength(); nIndex++)
    for (int nIndex = 0; nIndex < 4; nIndex++)                              ////change by dxf ++
    {

        if (szAllow.Find(szChannelConfig[nIndex]) == -1)
        {
            m_pPro->m_nLastErrorCode = ERR_REG_NAME;
            return 1;
        }
    }
```

```
    //if low the index number less than the high index number
    if(pRegInfo[nRegIndex].nLowIndex < pRegInfo[nRegIndex].nUpperIndex)
    {

        if(nNo < pRegInfo[nRegIndex].nLowIndex)
        {
            //if not in the range
            m_pPro->m_nLastErrorCode = ERR_REG_CH;
            return 1;
        }
        else if(nNo >pRegInfo[nRegIndex].nUpperIndex)
        {
```

```
            m_pPro->m_nLastErrorCode = ERR_REG_CH;
            return 1;
        }
        else
        {
            //success
            pPlcVar -> nNo = nNo;
        }
    }

    else if (pRegInfo[nRegIndex].nLowIndex ==  0 && pRegInfo[nRegIndex].nUpperIndex == 0)
    {
        if(nNo > 0)
        {
            m_pPro->m_nLastErrorCode = ERR_REG_CH;
            return 1;
        }
        else if(nNo < 0)
        {
            m_pPro->m_nLastErrorCode = ERR_REG_CH;
            return 1;
        }
        //default
        pPlcVar->nNo = 0;
    }
    else
    {
        m_pPro->m_nLastErrorCode = ERR_REG_FORMAT;
        return 1;
    }
```

```
    pPlcVar->nRegType = nRegIndex;
    pPlcVar->pszRegName = pRegInfo[nRegIndex].sRegName;

    if(pDbItem->nDataType & pRegInfo[nRegIndex].wDataType)
    {
        pPlcVar->nDataType = pDbItem->nDataType;
        return 0;
    }
    else
    {
        m_pPro->m_nLastErrorCode =  ERR_REG_DATATYPE;
        return 1;
    }

    return 0;
}
```

图 8－13 修改后的 ConvertUserConfigToVar( )函数

（3）修改 AddVarToPacket( )**函数**

该函数是一个组包函数，其主要功能是将组态王中自动扫描读取的变量打包，然后形成读取指令以提高读取数据的效率（目前组态王并不支持“写数据”的打包）。虽然 TTCANopen 协议的读取指令支持 1～255 个字节数据的读取，但当读取字节数大于 8 时，设备将分成多条指令进行响应，为了稳妥，在该函数中我们还是限制了读指令打包的读取字节数，其上限为 8 个字节。图 8－14 是修改后的 AddVarToPacket( )函数。

```
BOOL CDevTTCANopen7000ssss::AddVarToPacket( LPVOID lpVar, int nVarAccessType, LPVOID lpPacket)
{
    CDebug::ShowFunMessage(_T("CDevTTCANopen7000ssss::AddVarToPacket"));
    PPACKET pPac = (PACKET *)lpPacket;
    PPLCVAR pVar = (PLCVAR *)lpVar;
    /*
    Add you code here
    the following code for your reference.
    */

    //total vars in the packet
    int nTotalNo;
    //only if var type, address, data types are all the same
    if((nVarAccessType == pPac->nPacketType)&&(pPac->nUnitNo == pVar->nUnitNo)
        && (pPac->nRegType == pVar->nRegType))
    {
        //write
        if ( nVarAccessType == PT_WRITE)       //对写包进行组包，只有相邻变量才能组在一个包内，目前组态王不支持写数据组包
        {

        }

        //read   //对读包进行组包，没有限制为必须是相邻的变量组包  目前，组态王只支持对读数据的组包
        //if var channel less than the packet start No
        if ( pVar->nNo < pPac->nStartNo )
        {//chart:       -----pVar->nNo-------pPac->Start================pPac->End----------
            //get the channel rangle
            switch(pPac->nRegType)
                {
                    case REG_TYPE_BI:
                    case REG_TYPE_BO:
                    case REG_TYPE_B8I:
                    case REG_TYPE_B8O:
                    case REG_TYPE_CI:
                    case REG_TYPE_CO:
                    case REG_TYPE_KCI:
                    case REG_TYPE_KCO:
                        {
                            nTotalNo = pPac->nEndNo - pVar->nNo + 1;    //单字节
                        }
                        break;
                    case REG_TYPE_SI:
                    case REG_TYPE_SO:
                    case REG_TYPE_USI:
                    case REG_TYPE_USO:
                    case REG_TYPE_KUSI:
                    case REG_TYPE_KUSO:
                    case REG_TYPE_BCDI:
                    case REG_TYPE_BCDO:
                        {
                            nTotalNo = pPac->nEndNo - pVar->nNo + 2;    //双字节
                        }
                        break;
                    case REG_TYPE_LI:
                    case REG_TYPE_LO:
                    case REG_TYPE_LBCDI:
                    case REG_TYPE_LBCDO:
                    case REG_TYPE_FI:
                    case REG_TYPE_FO:
                        {
                            nTotalNo = pPac->nEndNo - pVar->nNo + 4;     //四字节
                        }
                        break;
                    default:
                        return FALSE;
                }//switch
```

```
//          if ( nTotalNo <= MAX_PACKET_NUM )
            if ( nTotalNo <= 8 )                //一个CAN帧最多可传输 8 个字节
            {
                pPac->nStartNo = pVar->nNo;
                return TRUE;
            }
        }

        //if var channel larger than the packet start No
        else if ( pVar->nNo > pPac->nEndNo )
        {//chart:------------pPac->Start================pPac->End----pVar->nNo------
            //get the channel rangle
            switch(pPac->nRegType)
            {
                case REG_TYPE_BI:
                case REG_TYPE_BO:
                case REG_TYPE_B8I:
                case REG_TYPE_B8O:
                case REG_TYPE_CI:
                case REG_TYPE_CO:
                case REG_TYPE_KCI:
                case REG_TYPE_KCO:
                    {
                        nTotalNo = pVar->nNo - pPac->nStartNo + 1;    //单字节
                    }
                    break;
                case REG_TYPE_SI:
                case REG_TYPE_SO:
                case REG_TYPE_USI:
                case REG_TYPE_USO:
                case REG_TYPE_KUSI:
                case REG_TYPE_KUSO:
                case REG_TYPE_BCDI:
                case REG_TYPE_BCDO:
                    {
                        nTotalNo = pVar->nNo - pPac->nStartNo + 2;    //双字节
                    }
                    break;
                case REG_TYPE_LI:
                case REG_TYPE_LO:
                case REG_TYPE_LBCDI:
                case REG_TYPE_LBCDO:
                case REG_TYPE_FI:
                case REG_TYPE_FO:
                    {
                        nTotalNo = pVar->nNo - pPac->nStartNo + 4;    //四字节
                    }

                    break;
                default:
                    return FALSE;
            }//switch

//          if ( nTotalNo <= MAX_PACKET_NUM )
            if ( nTotalNo <= 8 )                //一个 CAN 帧最多可传输 8 个字节
            {
                pPac->nEndNo = pVar->nNo;
                return TRUE;
            }
        }
        else
        {//chart:------------pPac->Start====pVar->nNo============pPac->End----------
            return TRUE;
        }
        //
        //chart:------------pPac->Start====pVar->nNo============pPac->End----------
    }
    return FALSE;
}
```

图 8－14 修改后的 AddVarToPacket( )函数

（4）修改 ProcessPacket2( )函数

图 8－15 是自动框架生成的 **ProcessPacket2**( )函数，该函数是被组态王自动调用的包处理函数，该函数通过调用几个相关处理函数完成对包的处理过程。当然用户可以根据应用进行修改，根据 TTCANopen 处理进程的特点，简化函数的处理过程，我们对该函数进行了大幅度的删节，只保留了 SendDataToKingView( )函数的调用，修改后的函数见图 8－16。

```
BOOL CDevTTCANopen7000ssssss::ProcessPacket2( LPVOID lpPacket )
{
    CDebug::ShowFunMessage(_T("CDevTTCANopen7000ssssss::ProcessPacket2"));

    ASSERT(lpPacket != NULL);
    PPACKET pPac = (PPACKET)lpPacket;
    ZeroMemory(m_bySndBuf, sizeof(m_bySndBuf));
    ZeroMemory(m_byRecBuf, sizeof(m_byRecBuf));
    if( !AssertPacketProperty(pPac) )
    {
        return FALSE;
    }

/*Notes:
In this function, we call the following functions to access finish processing data packet.

GetSendTimes:For some devices, we need send and receive repetitious to get the expected data frame,
             this function is to get the alternation times.

GetSendString:To build the send frame according to communication protocol.

Transmission: Send data to communication port and receive data from communication port.

PreProcessData:Pre-process the received data packet, such as to find the frame head flag and check the data.

SendDataToKingView:Send the disposed value to KingView's variables.

you can continue to use these functions structure, or you can modify the functions structure freely.
*/
```

```
    int nSendLen = 0;                           //send length
    int nExpectedLen = 0;                       //expect length to receive
    int nRecLen = 0;                            //received lenght

    if (!m_pSerialCom->m_bSuccess)
    {
        return FALSE;
    }
    if(pPac->nRegType == FLOW_REG)
    {
        //
        memcpy(m_byRecBuf, m_pSerialCom->m_AsynData, MAX_BUFFER);
        if (!PreProcessData(pPac, nRecLen, nExpectedLen, 1))
        {
            CDebug::ShowErroMessage(_T("PreProcessData error!"));
            return FALSE;
        }
        //send correct data to KingView
        if (pPac->nPacketType == PT_READ)
        {
            return SendDataToKingView(pPac, nExpectedLen);
        }
    }
    else if(pPac->nRegType == PARA_REG)
    {
        //use write event to deal with query-ack ,
        //it's different from the real-time send data
        if (! ResetEvent(m_pSerialCom->m_hThreadWriteEvent) )
        {
            return FALSE;
        }
```

```
        if (!GetSendString(pPac, nSendLen, nExpectedLen, 1))
        {
            CDebug::ShowErroMessage(_T("Get send string error!"));
            return FALSE;
        }
        if(!m_pSerialCom->PhysicalSend(m_bySndBuf, nSendLen))
        {
            return FALSE;
        }
        DWORD r = WaitForSingleObject(m_pSerialCom->m_hThreadWriteEvent, m_pSerialCom->m_dwTimeOut);
        if( r == WAIT_TIMEOUT || !m_pSerialCom->m_bSuccess )
        {
            return FALSE;
        }
        memcpy(m_byRecBuf, m_pSerialCom->m_SynData, MAX_BUFFER);

        if (!PreProcessData(pPac, nRecLen, nExpectedLen, 1))
        {
            CDebug::ShowErroMessage(_T("PreProcessData error!"));
            return FALSE;
        }
        //send correct data to KingView
        if (pPac->nPacketType == PT_READ)
        {
            return SendDataToKingView(pPac, nExpectedLen);
        }

    }
    return TRUE;
}
```

图 8－15 自动框架生成的 ProcessPacket2( )函数

```
BOOL CDevTTCANopen7000ssssss::ProcessPacket2( LPVOID lpPacket )
{
    CDebug::ShowFunMessage(_T("CDevTTCANopen7000ssssss::ProcessPacket2"));

    ASSERT(lpPacket != NULL);
    PPACKET pPac = (PPACKET)lpPacket;
    ZeroMemory(m_bySndBuf, sizeof(m_bySndBuf));
    ZeroMemory(m_byRecBuf, sizeof(m_byRecBuf));
    if( !AssertPacketProperty(pPac) )
    {
        return FALSE;
    }

    int nSendLen = 0;                               //send length
    int nExpectedLen = 0;                           //expect length to receive
    int nRecLen = 0;                                //received lenght

    //send correct data to KingView
//  if (pPac->nPacketType == PT_READ)   //注意：根据需要进行条件设置!!!!!!!
    {

        return SendDataToKingView(pPac,nExpectedLen);
    }
    return TRUE;
}
```

图 8－16 修改后的 ProcessPacket2( )函数

（5）修改传输错误重发机制

组态王对串口传输出错提供了一个出错重发机制，而我们建立的系统只是借道“虚拟串口”，而实际的现场总线是 CAN 总线，大家知道 CAN 的底层协议提供了全套的检测机制，因此组态王提供的这套机制不再被需要，只简单的屏蔽即可。

（6）修改 SendDataToKingView( )函数

图 8－17 是自动框架生成的 SendDataToKingView( )函数，它几乎是一个空的函数框架，而我们的任务就是“填充”这个函数，我们前面所做的工作可以说都是一些辅助性的工作，驱动程序绝大部分的处理都是在这个函数中完成的。

由于该函数全部内容比较长，不易在本书中展示，好在多数内容是重复的对多种变量类型进行处理，在这里我们只举证一种变量（即：S 整型）的处理过程，其他照此类推即可。这里需要提醒用户的是 TTCANopen 原生并不支持单独的“位”变量操作，而是通过整字节的“位与”“位或”操作间接完成。

在驱动中我们建立了两个非常重要的 map［64］数组，一个是 ztw_maps，用于记录组态王上层软件对相关“寄存器”的操作指令和数据内容；另一个就是 ttc_maps，用于记录 CAN 总线上 TTCANopen 对相关“寄存器” 的操作指令和数据内容。我们用数组的序数与设备编号对应，map 索引序数与寄存器地址对应，在每个 map 记录中我们定义了三个元素，Special、order 和 data，其中 Special 用于“位”操作时的“位置”标识，order 是操作指令，data 是寄存器中的数据。

通过使用这两个 map，大大简化了驱动程序中的相关逻辑推演的复杂度，这是整个驱动程序的精髓所在，图 8－18 为简化后的 SendDataToKingView( )函数。

```
BOOL CDevTTCANopen7000ssssss::SendDataToKingView(PPACKET pPac, int nLen)
{
    CDebug::ShowFunMessage(_T("CDevTTCANopen7000ssssss::SendDataToKingView"));
    ASSERT(pPac != NULL);
    //
    POSITION pos =  pPac->varList.GetHeadPosition();

    while(pos)
    {
        ID_NO2 * pIdNo = static_cast<ID_NO2*>(pPac->varList.GetNext(pos));

/*
Add your  handler code here,
The pIdNo is the pointer to  KingView variables in the packet.
About the variable's data types, see "datatype.h"
You can also call the SendDataToIdNo function to realize this,
you can see the demo example for reference.
*/
    }

    return TRUE;
}
```

图 8－17 自动框架生成的 SendDataToKingView( )函数

```
*********************************************************************/
////////////////////////////////////////////////////////////////////////////////
//
//   在这个函数中，如果“包”为PT_READ，ztw_MAP无法直接从组态王获取对应变量值，当在ttc_MAP中返回了查询数据，则用ttc_MAP中的数据填充到ztw_MAP
//   如果“包”为PT_WRITE，则直接将组态王变量值写入ztw_MAP 当获取到对应ttc_MAP的数据后，再用ttc_MAP中的数据填充到ztw_MAP中，
//   注意：在ztw_MAP中总是应当拥有组态王对应数据的记录，
//   注意：在驱动中一般不删除ztw_MAP和ttc_MAP中的记录，只是通过改变记录中order的状态，表征记录的状态，如order=0xEE,则表示该记录已经被使用过了
//
//   该函数， 首先检查包中的变量所对应的ttc_MAP中，是否有新鲜的记录，如果有则将此记录预备返回组态王，如果后续生成的指令返回了更加新鲜的数据
//   则用其替换此值，返回组态王。
//
////////////////////////////////////////////////////////////////////////////////
BOOL CDevTTCANopen7000ssssss::SendDataToKingView(PPACKET pPac, int nLen)
{
    CDebug::ShowFunMessage(_T("CDevTTCANopen7000ssssss::SendDataToKingView"));
    ASSERT(pPac != NULL);
    //
    int chn;    //设备编号
    int regn;   //寄存器地址
    int n_off = 0;  //包的字节长度尾部偏移量

    int is_xGH = 0;    //是否是设备主动上报变量，即G变量，1为是

    BYTE order = 0xEE;


    chn = static_cast<BYTE>(pPac->nUnitNo);

    m_pSerialCom->shebei_is_working[chn] = 1;  //标识设备存在，并在线


    POSITION pos =  pPac->varList.GetHeadPosition();

    while(pos)
    {
        ID_NO2 * pIdNo = static_cast<ID_NO2*>(pPac->varList.GetNext(pos));

        regn =  pIdNo->wNo;

        switch(pPac->nRegType)
        {
            case REG_TYPE_SI:
            case REG_TYPE_SO:
            case REG_TYPE_SW:
                {
                    n_off = 2;    //双字节
                    BYTE   bb[4]={0,0,0,0};

                        //设置ztw_MAP中的order，为生成指令做准备，
                        if(pPac->nPacketType == PT_READ)
                        {
                            order = 0x09;      //数据读取指令
                        }
                        if(pPac->nPacketType == PT_WRITE)
                        {
                            order = 0x08;      //数据写入指令
                            ORDER_DATA aaa = {0,order,0};

                            *(short*)bb = pIdNo->plcValue.intVal;

                            aaa.data = bb[0];
                            m_pSerialCom->ztw_Maps[chn][regn] = aaa; //将组态王中的数据写入ztw_Maps
                            aaa.data = bb[1];
                            m_pSerialCom->ztw_Maps[chn][regn+1] = aaa; //将组态王中的数据写入ztw_Maps

                        }
                }
                break;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

            case REG_TYPE_SG:   //数据是设备主动上报的类型，包括0x6和0x19指令
                {
                    is_xGH = 1;
                    BYTE   bb[4]={0,0,0,0};
                    //从ttc_MAP中提取数据，将其写入组态王
                    m_pSerialCom->theIterator = m_pSerialCom->ttc_Maps[chn].find(regn);
                    if(m_pSerialCom->theIterator != m_pSerialCom->ttc_Maps[chn].end() )
                    {
                        bb[0] = m_pSerialCom->ttc_Maps[chn][regn].data;
                        bb[1] = m_pSerialCom->ttc_Maps[chn][regn+1].data;
                        pIdNo->plcValue.intVal = *(short*)bb;

                        CHelper::KwCoFileTimeNow(&pIdNo->ftTimeStamps);
                        pIdNo->wQualities = 0xc0;  //数据质量 好
                    }
                }
                break;
```

```
////////////////////////////////////////////////////////////////////////////////////////////////
//////////处理数据是设备主动请求类型，0x07指令//////////////

            case REG_TYPE_ST:   //////////将组态王数据写入ztw_Maps中，备设备用0x07指令读取//////////////
                {
                    is_xGH = 1;
                    BYTE   bb[4]={0,0,0,0};
                    //将组态王的数据写入ztw_Maps中
                    ORDER_DATA aaa = {0,0x17,0};
                    *(short*)bb = pIdNo->plcValue.intVal;
                    aaa.data = bb[0];
                    m_pSerialCom->ztw_Maps[chn][regn] = aaa;
                    aaa.data = bb[1];
                    m_pSerialCom->ztw_Maps[chn][regn+1] = aaa;
                }
                break;


          default:
              {
                    ASSERT(FALSE);
                    m_pPro->m_nLastErrorCode = ERR_REGISTER_NOT_EXIST;
                    CDebug::ShowErroMessage(_T("The register not exist!"));

                    ZeroMemory( &pIdNo->plcValue, sizeof(PlcValue) );
                    pIdNo->wQualities = COM_QUALITY_GOOD;

                    return FALSE;
              }

        }//switch

    }  //while


    if( is_xGH == 1)  return TRUE;   //如果上面处理的是设备主动上报的变量，则不用生成发送指令，函数至此返回TRUE


   ////////////////////////生成发送指令/////////////////////////////////////////////////////

    BYTE  sd_CAN_data[16];       //定义临时发送缓冲区数组

    if(order == 0x09)           //如果该“包”为主从查询包，在这里生成查询指令//
    {
        sd_CAN_data[0] = 0xAA;
        sd_CAN_data[1] = 02;
        sd_CAN_data[2] = HIBYTE(pPac->nStartNo);
        sd_CAN_data[3] = LOBYTE(pPac->nStartNo);
        sd_CAN_data[4] = static_cast<BYTE>(pPac->nUnitNo);
        sd_CAN_data[5] =(BYTE)order;
        sd_CAN_data[6] = 0x01;
        sd_CAN_data[7] = LOBYTE(pPac->nEndNo - pPac->nStartNo + n_off);
        sd_CAN_data[8] = 0xDD;
//      memcpy( m_pSerialCom->s_CAN_datas.data[m_pSerialCom->s_CAN_datas.head], sd_CAN_data, (8+sd_CAN_data[6]));  //将查询指令写入发送循环缓
//      m_pSerialCom->s_CAN_datas.head++;
    }

    if((order == 0x08)||(order == 0x03)||(order == 0x04)||(order == 0x05))  //如果该“包”为主从写入包，或“位指令”，在这里生成写入指令//
    {
        sd_CAN_data[0] = 0xAA;
        sd_CAN_data[1] = 02;
        sd_CAN_data[2] = HIBYTE(pPac->nStartNo);
        sd_CAN_data[3] = LOBYTE(pPac->nStartNo);
        sd_CAN_data[4] = static_cast<BYTE>(pPac->nUnitNo);
        sd_CAN_data[5] = (BYTE)order;
        sd_CAN_data[6] = LOBYTE(pPac->nEndNo - pPac->nStartNo + n_off);
        for(int i=0; i<sd_CAN_data[6];i++)
        {
            sd_CAN_data[7+i] = m_pSerialCom->ztw_Maps[chn][0x0000FFFF&(pPac->nStartNo+i)].data;
        }
        sd_CAN_data[7+sd_CAN_data[6]] = 0xDD;
//      memcpy( m_pSerialCom->s_CAN_datas.data[m_pSerialCom->s_CAN_datas.head], sd_CAN_data, (8+sd_CAN_data[6]));  //将查询指令写入发送循环缓
//      m_pSerialCom->s_CAN_datas.head++;
    }

//  pPac->nPacketType = PT_READ;  //无论“包”原来是什么状态，一律设置为PT_READ，这样可将返回数据写入组态王


    m_pSerialCom->theIterator = m_pSerialCom->ttc_Maps[chn].find(0x0000FFFF&(pPac->nEndNo+ n_off-1));  //查询是否存在对应的ttc_Maps
    if(m_pSerialCom->theIterator != m_pSerialCom->ttc_Maps[chn].end() ) //如果有，将其置为无效，等待接收新的ttc_Maps
    {
        m_pSerialCom->ttc_Maps[chn][0x0000FFFF&pPac->nEndNo].order =  0xEE;
    }


    int j;

    for( j=0; j<3; j++)   ///////指令重复次数，
    {

    memcpy( m_pSerialCom->s_CAN_datas.data[m_pSerialCom->s_CAN_datas.head], sd_CAN_data, (8+sd_CAN_data[6]));  //将指令写入发送循环缓冲区
    m_pSerialCom->s_CAN_datas.head++;

    m_pSerialCom->sand_buf(); //将发送循环缓冲区中的数据，通过串口发送



    for(int i=0; i<300; i++)   ///////
    {
       //WAIT_INTERVAL=5
    Sleep(WAIT_INTERVAL);  //等待设备通过串口返回查询结果数据
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///  不知道为什么在pPac->nEndNo中前面多了4个FFFF,如：0x8000 变成了0xffff8000,  因此在使用这个参数时，要屏蔽前面4个ffff  //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    m_pSerialCom->theIterator = m_pSerialCom->ttc_Maps[chn].find(0x0000FFFF&(pPac->nEndNo+ n_off-1));  //查询是否获取了最后一个字节数据
    if(m_pSerialCom->theIterator != m_pSerialCom->ttc_Maps[chn].end() ) //如果收到最后一个字节，则将设备返回的数据写入组态王
    {
     if( m_pSerialCom->ttc_Maps[chn][0x0000FFFF&pPac->nEndNo].order == order+0x10) //是否是所发指令的应答指令
     {
        i = 900; //跳出等待循环；

        pos =  pPac->varList.GetHeadPosition();  //重新定位包列表
        while(pos)
        {
            ID_NO2 * pIdNo = static_cast<ID_NO2*>(pPac->varList.GetNext(pos));

            regn =  pIdNo->wNo;

//          pIdNo->wQualities = 0x0;   //先将数据质量设置为无效


            switch(pPac->nRegType)
            {


                case REG_TYPE_SI:
                case REG_TYPE_SO:
                case REG_TYPE_SW:
                    {
                        BYTE   bb[4]={0,0,0,0};
                        //将ttc_Maps中数据写入组态王    由于前面已经对ttc_Maps进行了find检查，这里不再进行find测试，而是认为对应ttc_Maps已经有
                        //并且已经做了是否是应答回复指令的测试，这里不再测试，
                        if(m_pSerialCom->ttc_Maps[chn][regn].order == 0x19) //如果是新鲜的读指令返回数据
                        {
                            m_pSerialCom->ttc_Maps[chn][regn].order = 0xEE;//标识数据已被使用
                            m_pSerialCom->ttc_Maps[chn][regn+1].order = 0xEE;//标识数据已被使用

                            bb[0] = m_pSerialCom->ttc_Maps[chn][regn].data;
                            bb[1] = m_pSerialCom->ttc_Maps[chn][regn+1].data;


                            pIdNo->plcValue.intVal = *(short*)bb;

                            CHelper::KvCoFileTimeNow(&pIdNo->ftTimeStamps);
                            pIdNo->wQualities = 0xc0;
                            j=10;  //跳出j循环
                        }
                        if( m_pSerialCom->ttc_Maps[chn][regn].order == 0x18) //如果是新鲜的写指令返回数据数据
                        {
                            m_pSerialCom->ttc_Maps[chn][regn].order = 0xEE;//标识数据已被使用
                            m_pSerialCom->ttc_Maps[chn][regn+1].order = 0xEE;//标识数据已被使用

                            if(    m_pSerialCom->ttc_Maps[chn][regn].data == m_pSerialCom->ztw_Maps[chn][regn].data
                                && m_pSerialCom->ttc_Maps[chn][regn+1].data == m_pSerialCom->ztw_Maps[chn][regn+1].data  )
                                j=10;  //跳出j循环
                        }
                    }
                    break;

                default:
                    {
                        ASSERT(FALSE);
                        m_pPro->m_nLastErrorCode = ERR_REGISTER_NOT_EXIST;
                        CDebug::ShowErroMessage(_T("The register not exist!"));

                        ZeroMemory( &pIdNo->plcValue, sizeof(PlcValue) );
                        pIdNo->wQualities = COM_QUALITY_GOOD;

                        return FALSE;
                    }

        }//switch


      }  //while

     }//if( (*m_pSerialCom->theIterator).second.order != 0xEE)
    }//if(m_pSerialCom->theIterator != m_pSerialCom->ttc_Maps[chn].end() )

   }//for(int i=0; i<20; i++)
  }//for(int j=0; j<2; j++)

    if(j<10)
        return FALSE;
    else
        return TRUE;
}
```

图 8－18 简化后的 SendDataToKingView( )函数

8.3.5 SerialCom.cpp 文件的“修改”

（1）在串口 SerialCom.hpp 中定义 ztw_maps［64］和 ttc_maps［64］

（2）在 DealReceivedData( )函数中完成 TTCANopen 的帧同步和缓存，并生成对应的应答指令。

由于上述的程序过程比较简单，且这部分的程序代码是中铁航的工程师从“CAN 模块”的单片机中移植修改而成，可读性不强，这里就不给用户展示了。

8.4 组态软件应用测试

我们开发了两种组态王的驱动程序，第一种是常规的驱动程序，安装在中心监控计算机上帮助系统完成对各 TTCANopen 设备的监控；另一种是“设备模拟驱动”，主要是用来模拟各种 TTCANopen 设备并与主机进行通讯。有了这个驱动我们就可以在不开发相关硬件的情况下，模拟系统的工作过程，这对系统的前期软件开发起着非常大的促进作用。

按照组态王驱动开发包提供的文档说明，安装好驱动程序，就可以进行简单的测试了。

8.4.1 搭建测试环境

我们使用了三台笔记本计算机搭建组态王测试环境，其中一台用于中心监控（组态王），一台用于仿真各种 TTCANopen 设备（组态王），它们之间通过“应用模块”进行连接，第三台使用 “监听模块”连接，通过串口调试助手对 CAN 网络进行监视见图 8－19。

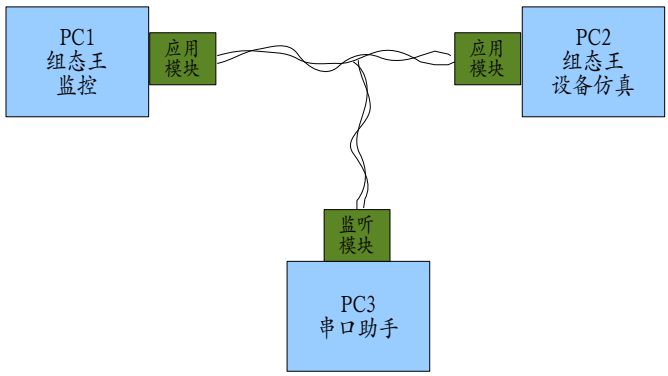

图 8－19 组态王 TTCANopen 测试环境

8.4.1 简单轮询系统测试

我们在 PC2 上用组态王建立一个模拟设备，并新建一个只读变量，使用寄存器设置为 SI_0x8000，其取值范围 0～100，且每 1 秒自动对该值＋10，当其大于 100 时清零，我们用表头图形去显示该值，该设备响应 TTCANopen 的读取指令，返回当前只读变量值给监控机。

在中心监控 PC1 上对应建立上述设备和变量，寄存器同样设置为 SI_0x8000，并用表头图形显示该值，设置组态王读取周期为 200 毫秒，当两个设备设置好并启动运行，我们可以看到 PC1 上的表头指针会跟 PC2 上的表头指针随动，指向相同的数值，没有失步现象，见图 8－20。

图 8－20 中心监控表头与设备表头随动

通过 PC3 串口调试助手的数据接收窗口观察 CAN 总线指令收发情况，我们发现有多条读取指令读到相同的数值，重采现象严重，见图 8－21。

图 8－21 总线上数据重采现象

当我们调整 PC1 的读取周期为 2 秒时，PC1 上的表头指针变得迟钝，其随动性变差，甚至发生了严重的失步现象，见图 8－22。

图 8－22 中心监控表头与设备表头随动失步

通过 PC3 观察 CAN 总线指令收发情况，我们发现有些 PC2 的数据没有被采集到，出现漏采现象，见图 8－23。

图 8－23 总线上数据漏采现象

重采和漏采都是我们所不愿意见到的，然而它却是轮询系统不可调和的一对矛盾，当采集频率高时，系统对数据变化的反应灵敏度就高，但重采会很严重，总线的利用率会很低，当采集频率过低时， 系统对数据变化的反应灵敏度会变差，甚至会丢失数据，这对告警信息的采集尤为不利。

幸好，CAN 总线给我们提供了总线竞争机制，使设备能够在总线上主动上报数据，从而摆脱了轮询系统的束缚，这也是 CAN 总线优于 485 总线的特征之一，下一个实验将展示一个简单的事件触发测试。

8.4.2 简单设备触发方式系统测试

我们在 PC2 上用组态王建立一个模拟设备，并新建一个事件触发变量，使用寄存器设置为 SG_0x8000，其取值范围 0～100，且每 1 秒自动对该值＋10，当其大于 100 时清零，我们用表头图形去显示该值，当该变量发生变化时设备会主动发送数据上报指令（0x19），将当前值发送给监控机。

在中心监控 PC1 上对应建立上述设备和变量，寄存器同样设置为 SG_0x8000，并用表头图形显示该值，当两个设备设置好并启动运行，我们可以看到 PC1 上的表头指针会跟 PC2 上的表头指针随动，指向相同的数值，没有失步现象，见图 8－24。

图 8－24 中心监控表头与设备表头随动

在总线通讯速率许可的范围内，我们调整模拟设备事件触发变量数值发生变化的运行周

期，甚至使用手动方式触发其变化，我们看到 PC1 上的表头指针始终跟 PC2 上的表头指针随动，指向相同的数值，没有失步现象，观察 PC3 中 CAN 总线指令收发情况，其即没有重采现象也没有漏采发生，见图 8－25。

图 8－25 总线上即无重采也无漏采

从以上实验，我们看到事件触发方式克服了轮询方式的弊端，其对数据变化反应的即时性和对总线的利用率上都大大的优于传统的轮询方式，我们在设计系统时应当优先选择事件触发方式，对于哪些连续时间变量则可以根据实际应用情况选择时间周期触发或数值增量触发。

在上面的实验中，当事件长时间没有触发，从机跟主机就没有通讯发生，此时的主机就没有办法辨别从机的状态，是否出现了通讯故障等，因此，在全触发方式的设备上都设计有“心跳信息”，即设备周期的向主机发送心跳信息，告知主机“我还活着，正在跟你说话。”。

8.4.3 简单控制系统测试

在 PC2 上用组态王建立一个模拟设备，并新建一个只写变量，寄存器设置 SW_0x8000，其取值范围 0～100，我们用表头图形去显示该值，设备接受监控机发送来的写指令，并按写指令中的内容设置刷新该值，并返回写应答指令。

在中心监控 PC1 上对应建立上述设备和变量，寄存器同样设置为 SW_0x8000，并建立一个输入框用于输入该值，当两个设备设置好并启动运行，我们向 PC1 中的框中输入 0～100 的数值，PC2 上的表头指针将会指向该值，见图 8－26。PC3 上的 CAN 总线指令收发情况见图 8－27。

图 8－26 中心控制设备表头指针

图 8－27 监控控制指令

8.4.4 简单位控系统测试

“位”控或称开关控制在工控系统中是最常见的也被认为是最简单的控制之一，但在我们编写驱动时，发现它却是整个驱动中最难把握和编写的内容之一，一个简单的原因就是在于 TTCANopen 协议中没有真正的“位”控指令，协议中最基本的操作单位是字节，当然我们可以用一个字节去作为一个开关量去处理，这样一条指令最多可控制 8 个开关，从资源利用上讲确实有些浪费，但对于一些简单的和资源并不紧张的系统确实是可以这样做的，但作为一个标准驱动则要为用户提供一个更为合理的解决方案，这完全仰仗于王剑宇先生编写的 TC0002A0 子协议，在该子协议中定义了对字节操作数的“位与指令”、“位或指令”和“位非指令”，使协议对字节流的控制深入到“位”。虽然如此，我们在编写驱动时还是感到“力不从心”，很难用比较简单的方法完成一个较为完整的“位”控功能过程。

在测试之前，我们先了解一下位变量是如何建立的，在组态王新建变量的对话框中使用寄存器设置应选 B 开头的选项，如：BI_0x8000.n 其中 n=0~7，这样我们就能够指定寄存器字节中的某一位代表该新建的“位”变量或开关变量。

1）“位”只读变量的测试

我们在 PC2 上用组态王建立一个模拟设备，这个设备由 8 个手动开关组成，每一个开关用一个只读“位”变量表示（注意：读写特性都是针对中心监控而言的），使用寄存器设置为 BI_0x8000.0~7，在设备上我们可以手动将设备开或合。

在中心监控 PC1 上对应建立上述设备和变量，寄存器同样设置为 SI_0x8000.0~7，并用 8 个指示灯显示被监控设备的开合，当设备开关合上时，指示灯显示为红色，断开时显示为灰色，见图 7－28。

图 8－28 只读“位”变量测试

在 PC3 上观察总线上的指令，见图 8－29，我们看到中心监控只用了一条普通的读取指令就将 8 个开关状态读回，如果我们继续增加开关的数量直至 64 个（注意：设置寄存器地址应当连续），中心还是会使用一条指令将 64 个开关状态读回，这是因为在驱动中 AddVarToPacket( )函数自动打包的结果，如果我们减少开关的数量，或者关闭多数开关量的访问，只剩下一个开关量，对于只读“位”变量来说，系统还是会使用读指令读回一个字节的状态，在该字节中包含了我们要读取的“位”变量。

图 8－29 只读“位”变量测试总线上的指令监视

2）“位”只写变量的测试

现在反过来，我们在 PC2 上用组态王建立一个模拟设备，这个设备由 8 个指示灯组成，每一个指示灯用一个只写“位”变量表示，使用寄存器设置为 BO_0x8000.0~7，当该“位”被写入 1 时显示红色，被写入 0 时为灰色。

在中心监控 PC1 上对应建立上述设备和变量，寄存器同样设置为 SO_0x8000.0~7，并用 8 个开关表示这些“位”变量，当开关合上时，向对应设备指示灯写入 1，使其显示为红色，断开时向对应设备指示灯写入 0，使其显示为灰色，见图 8－30。

图 8－30 只写“位”变量测试

在 PC3 上观察总线上的指令，见图 8－31，当我们改变某一个开关的状态，中心监控就会发送一条“位与指令”或“位或指令”用于改变设备指示灯的显示状态，即使我们使用编程手段同时改变两个或多个开关状态，中心监控还是会发送多条“位与指令”或“位或指令”，即：为每一个开关量的改变对应发送一条“位控”指令，这是因为组态王驱动开发不支持“写”数据的打包发送所至。

图 8－31 只写“位”变量测试总线上的指令监视

3）“组合位控”变量的测试

从上面的测试中，我们看到由于组态王驱动不支持写变量打包，使得“位”控变量的总线运行效率很低，这在小系统中矛盾并不突出，但在一些以开关量为主的大型应用中却是难以容忍的，因此我们特别设计了“组合位控”变量类型，如：CY_0x、CH_0x、CF_0x，这里 C 表示一个字节长度（含 8 个位），也就是将 8 个位变量当做一个字节进行处理，Y 表示“与”操作，H 表示“或”操作， F 表示“非”操作，当我们要对某寄存器内容进行“组合位控”操作时，需要在监控主机上建立四个指向该寄存器的变量，使用寄存器类型 CY_0x、CH_0x、CF_0x 和 CG_0x，分别表示“与操作数”、“或操作数”、“非操作数”和“返回结果”，当我们改变 CY_0x 对应的变量值时，该值会作为 TTCANopen“位与指令”的操作数发送给设备，设备返回的“位与”结果由 CG_0x 变量来显示。

我们在 PC2 上用组态王建立一个模拟设备，这个设备由 8 个指示灯组成，每一个指示灯用一个只写**内存离散**变量表示，当其被写入 1 时显示红色，被写入 0 时为灰色，我们另外新建一个只写字节变量，使用寄存器设置 CW_0x8000，并用程序语言使该字节的每一位与 8 个指示灯内存变量产生关联，从而使该字节的位值来影响各指示灯的颜色显示，设备接收中心监控设备发送来的写指令和位操作指令，并将结果返回监控主机。

在 PC1 上建立四个变量，对应使用寄存器 CY_0x8000、CH_0x8000、CF_0x8000 和 CG_0x8000，为前 3 各变量各建立一个变量输入框，为第四个变量建立一个变量显示框，设计好后启动程序，分别改变 3 各输入框中的数值，观察设备（PC2）端指示灯的显示状态和本机变量显示框中的返回值，见图 8－32。

图 8－32 “组合位控”变量的测试

在 PC3 上观察总线上的指令，见图 8－33，在这个实验中我们可以同时用一条指令控制设备上最多 8 个指示灯的状态，总线效率提高了，为此我们还设计了 USY_0x、CUSH_0x、USF_0x 可以同时控制 16 各开关的“组合位控”变量。

图 8－33 “组合位控”变量的测试总线上的指令监视

8.4.5 复杂系统模拟

任何复杂的系统都是由多个简单的控制过程相互作用演绎而来的，这其中主要包括用户应用逻辑和时序，并最终反映在系统 CAN 总线上 TTCANopen 协议指令逻辑与时序上，虽然我们使用 PC 机和组态软件可以模拟部分系统逻辑过程，但接近真实的时序过程却很难去模拟，如：总线竞争过程等，这必须要借助一定的硬件平台来完成，最简单最容易实现的硬件平台就是各种功能的亚当模块，由这些模块我们可以组建成一个较为完备的系统。

感兴趣的读者可以将我们上一章建立的测试环境中的主机上安装组态王软件和驱动，对空调控制系统进行测试。

注［1］：引自百度百科。

8.5 本章后记

由于本章成文相对较早，以至于我们后来对 TTCANopen 协议进行了非常重大的改动，而未能在本章中体现出来，本章的插图丢失，文字表述上也有许多不妥之处，包括部分［注释］也未能标识到位，在这里再次的向读者朋友表示歉意。尽管如此，我们还是将本章原样发表出来，希望对读者了解开发组态软件驱动有所帮助。

我们将在 TTCANopen 协议通过公测，并进入基本稳定期后，重新开发该组态软件驱动，有幸的话，希望能够邀请到北京亚控公司相关技术人员参予。