27 自编程序

27.1 使用 API 函数

■ 读取和写入控制器 /PLC

单句柄函数 (☞27.1.1 单 / 多句柄函数)

- 读取 API
  - 缓存型 (☞27.1.2 缓冲 / 直接型)
  - 直接型 (☞27.1.2 缓冲 / 直接型)
- 写入 API
  - 直接型 (☞27.1.2 缓冲 / 直接型)
  - 写入后刷新缓存 (☞27.1.2 缓冲 / 直接型)

■ PLC 与多台控制器通讯

多句柄函数 (☞27.1.1 单 / 多句柄函数)

- 读取 API
  - 缓存型 (☞27.1.2 缓冲 / 直接型)
  - 直接型 (☞27.1.2 缓冲 / 直接型)
- 写入 API
  - 直接型 (☞27.1.2 缓冲 / 直接型)
  - 写入后刷新缓存 (☞27.1.2 缓冲 / 直接型)

■ 高效通讯

- 组符号访问 (☞27.1.4 组访问)
- 排队访问 (☞27.1.5 排队访问)

■ 其他函数

- 系统 API(27.1.7 系统 API)
- SRAM 数据访问 API(27.1.8 SRAM 数据访问 API)
- CF 卡和 SD 卡 API(27.1.9 CF 卡和 SD 卡 API)
- 其他 API(27.9 其他 API)

27.1.1 单 / 多句柄函数

单句柄 API

此 API 用于与目标控制器的顺序通讯。调用一个 API 时，不能调用另一个 API。

但调用 API 可以免去一些麻烦的步骤，例如用 Pro-Server EX 访问句柄。

多句柄 API

此 API 可实现对多台控制器同时使用多个单句柄 API 功能。多句柄 API 与单句柄 API 的区别是：多句柄 API 名称的末尾有一个大写的字母 “M”。

例如，与单句柄 API “ReadDeviceVariant()”执行相同功能的多句柄 API，其名称为“ReadDeviceVariantM()”。

多句柄 API 可用于多线程应用程序，或用于同时访问多台控制器 /PLC。

27.1.2 缓冲 / 直接型

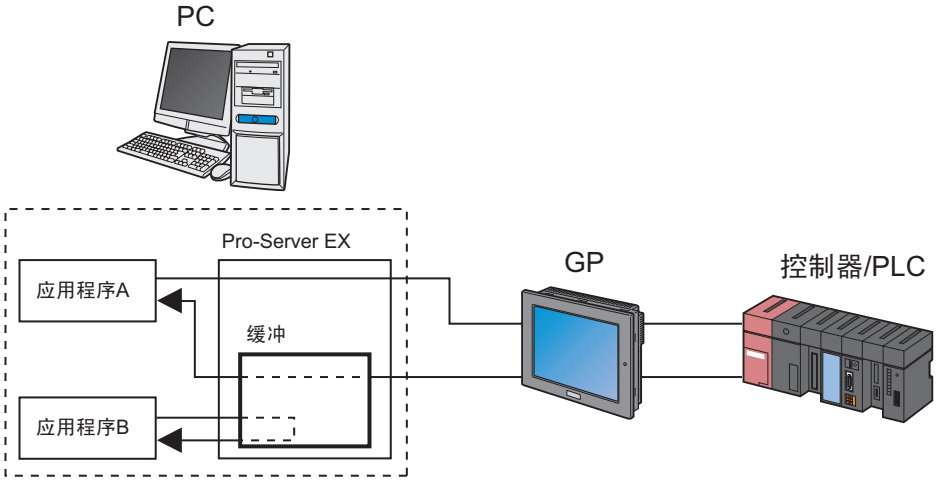
缓冲读取

当多个应用程序向同一台控制器 /PLC 发送读取请求时，如果 Pro-Server EX 访问控制器 /PLC 并逐个与各应用程序的读取请求会话，将花费较长的时间。

但是，使用缓冲读取功能，当两个应用程序 A 和 B 向同一台控制器 /PLC 发送读取请求时， Pro-Server EX 首先根据应用程序 A 的请求从控制器 /PLC 中读取数据，将其保存到内部缓冲区，然后响应读取请求将数据发送到应用程序 A。

随后，根据应用程序 B 的请求， Pro-Server EX 将保存在缓冲区中的数据发送到应用程序 B，因为响应数据已经和应用程序 A 的数据保存在了一起。

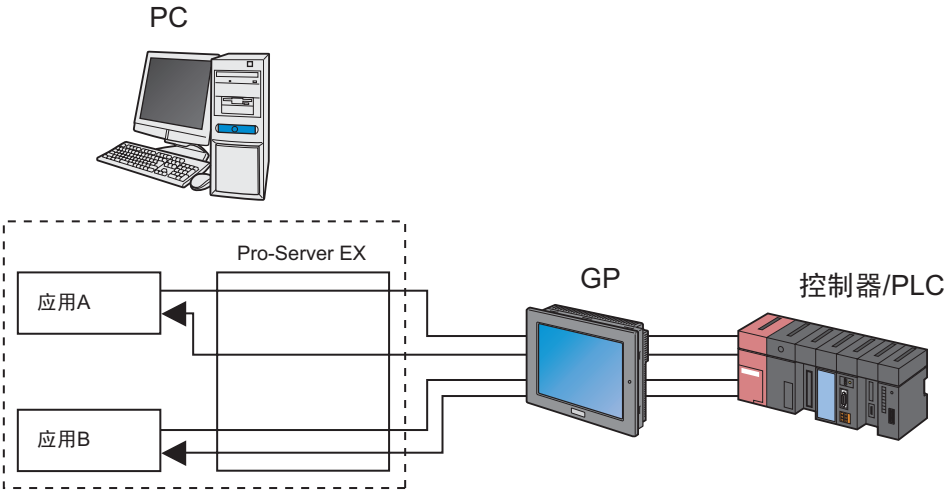
Pro-Server EX 还提供缓冲区控制 API。详情请参阅 “27.3 缓冲区控制 API”。



直接读取

此功能总是从控制器 /PLC 中读取最新的数据，无论缓存的状态如何。

直接读取 API 名称末尾有一个大写的 “D” 或 “DM”。



直接写入

此 API 写入数据。直接写入 API 名称末尾有一个大写的 “D” 或 “DM”。

写入并刷新缓冲区

从控制器缓冲存储数据时， Pro-Server EX 在写入数据后将重新读取相关寄存器数据，以便刷新缓冲区数据。

此 API 的处理速度要慢于直接写入 API。当 Pro-Server EX 含有缓冲读取的数据时，请使用 “写入并刷新缓冲区”。

27.1.3 缓冲区控制 API

使用缓冲区控制 API，可以确认目标寄存器的缓存数据是否得到了更新。

注 释
- 缓冲区控制 API 并非用于重写网络工程文件，而是用于在 Pro-Server EX 的内部存储器中添加或更改数据。

■ 缓冲区

缓冲存储寄存器数据时， Pro-Server EX 将多个寄存器作为一个整体来管理。管理的单位即称为 “缓冲区”。

(1) 一个缓冲区由多条记录组成。
(2) 一条记录可以是直接指定的多个连续寄存器地址、符号或组符号。
(3) 可以为每个缓冲区指定一个唯一的名称。

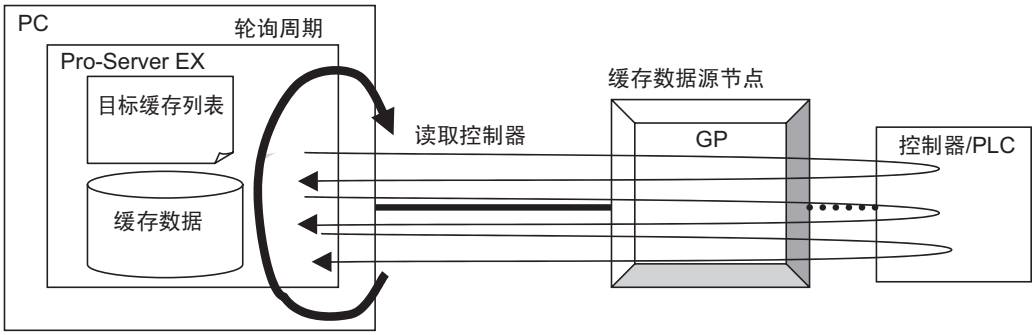
注 释
- 注册缓冲区可采用以下两种方法：
  (1) 用 Pro-Studio EX 注册 (在功能画面的 “Device Cache” 中创建缓冲区，然后将它注册到网络工程文件中。)
  (2) 用 API 注册

■ 缓冲区更新步骤

更新缓冲区可以用 “轮询” 和 “保持监视” 两种方法。

◆ 轮询方式的原理

达到缓冲区注册中指定的周期时， Pro-Server EX 根据缓冲区中的目标寄存器列表读取寄存器数据，从而更新缓冲区。

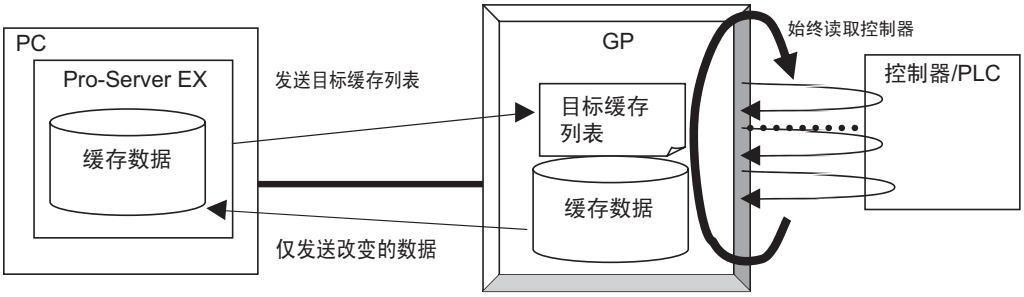


◆ 保持监视方式的原理

开始更新缓冲区时， Pro-Server EX 向数据源节点发送一份目标寄存器列表。

数据源节点根据此列表不断读取寄存器数据 (以最快速度)，并只将更改的数据发送到 Pro-Server EX。

Pro-Server EX 接收数据，并将其作为缓冲区数据进行处理。



注 释 • 如果缓存数据源节点为 GP 系列，则不能使用保持监视方式。

■ 选择保持监视方式或轮询方式

如果用保持监视方式来监视大量寄存器数据，则 Pro-Server EX 将耽于监视工作，从而使整个系统的性能下降。

为避免此类情况，建议仅对高度紧急的项目使用保持监视方式，而对其他项目使用轮询方式。

采用轮询方式时，基于 PC 或网络状况、控制器 /PLC 类型和系统性能，缓冲区可能未能按更新周期获得更新。此时，请使用直接读取 API。

至于各方式可接受的标准数据量，保持监视方式可处理几十到几百字节，轮询方式可处理几千字节。对于更大的数据量，请使用直接读取 API。

注意，允许的字节数因系统性能而不同。

■ 启动和停止缓冲存储

Pro-Server EX 缓冲存储启动 / 停止的时序描述如下。

(1) 缓冲存储由缓冲区启动或停止。

(2) 用 Pro-Studio EX 在网络工程文件中注册缓冲区，可选择以下三种注册方法。缓冲存储的启动时机如下。

1) 在启动 Pro-Server EX 时

启动 Pro-Server EX 并载入网络工程文件后， Pro-Server EX 启动缓冲存储。

重新载入网络工程文件时， Pro-Server EX 也会启动缓冲存储。

2) 读取预先注册的寄存器时自动启动缓冲存储

当寄存器读取 API 读取缓冲区中注册的寄存器时， Pro-Server EX 启动缓冲存储。

即使是对缓冲区中注册的部分寄存器执行读取， Pro-Server EX 也会对所有注册的寄存器启动缓冲存储。

除了寄存器读取 API 以外，所有读取方式均可启动缓冲存储。(例如，当一个寄存器被指定为数据传输功能的数据源或用于启动条件检查时，缓冲存储启动。)

但是，仅当用方式 2) 启动缓冲存储时，如果在一段时间内没有对缓冲区目标寄存器的访问， Pro-Server EX 才会停止缓冲存储。

3) 用使用缓冲区启动 API(PS_StartCache) 的程序启动缓冲存储

(3) 在下述情况下， Pro-Server EX 停止缓冲存储。

1) Pro-Server EX 关闭时，缓冲区停止，丢弃缓存数据。

2) 在重新载入网络工程文件之前，缓冲区停止，丢弃缓存数据。

3) 启用了 “Automatically start when a registered device is read”功能，启动缓冲存储后，在一段时间内没有访问缓冲区，缓冲区停止。(不丢弃缓存数据。)

4) 用使用缓存停止 API(PS_StopCache) 的程序停止了缓冲区。

27.1.4 组访问

有些 API 用组符号指定寄存器地址。

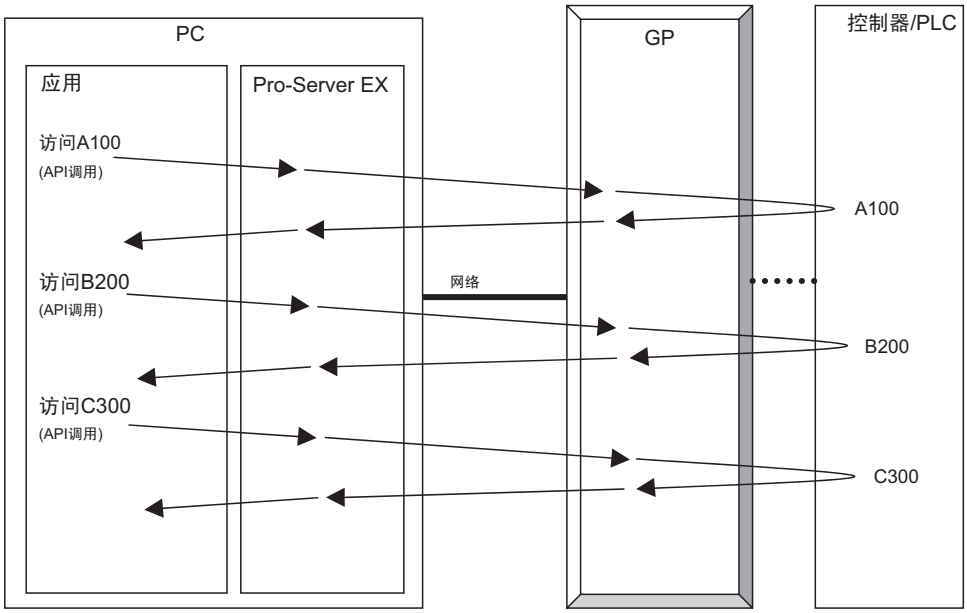
使用组符号， Pro-Server EX 能通过调用一个 API 高效地访问多个寄存器。

注 释

- Pro-Server EX 通过包含多个寄存器的组符号访问多个寄存器时，访问速度会变快， Pro-Server EX 和 GP 在内部优化处理过程。因此，不能指定寄存器的访问顺序。（“组符号注册”中符号的注册顺序并不代表访问顺序。）
  多个寄存器中的任意一个如发生访问错误，处理将停止。 Pro-Server EX 将其视为整个组的访问错误，不再访问其余的寄存器。
- 一个 API 调用允许的最大组符号数据量为 1M 字节。

◆ 为各寄存器单独调用 API 时：

每次调用 API， Pro-Server EX 均与寄存器通讯。

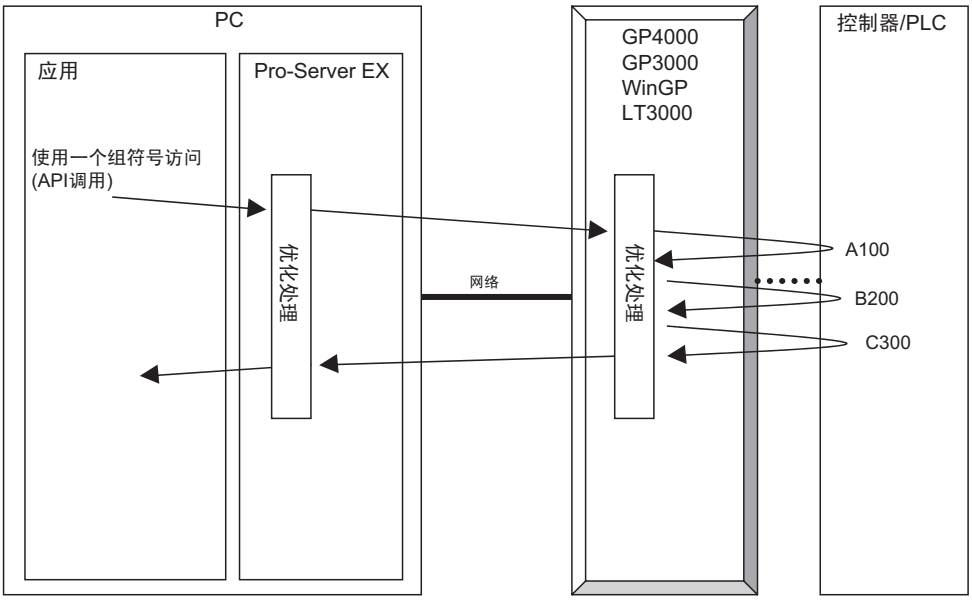


◆ 访问组符号时

处理过程因目标节点是 GP4000 系列、 GP3000 系列、 WinGP、 LT3000 还是 GP 系列而有所不同。

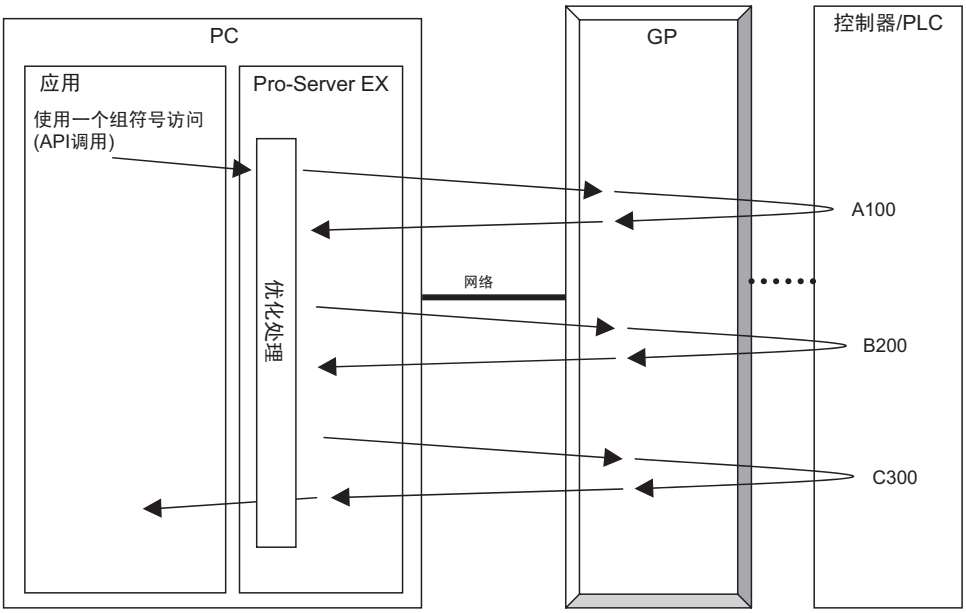
- GP4000 系列节点、 GP3000 系列节点、 WinGP 节点、 LT3000 节点

  'Pro-Server EX 仅向各节点发送一次请求。节点会在其内部划分请求，以分别访问各台控制器。这样， Pro-Server EX 就能高效地与网络中和各台控制器进行通讯。



- GP 系列节点

  仅调用一次 API， Pro-Server EX 内部划分请求，分别访问各 GP 系列节点。但是，如果组中含有几个连续符号， Pro-Server EX 会立刻访问这些符号。

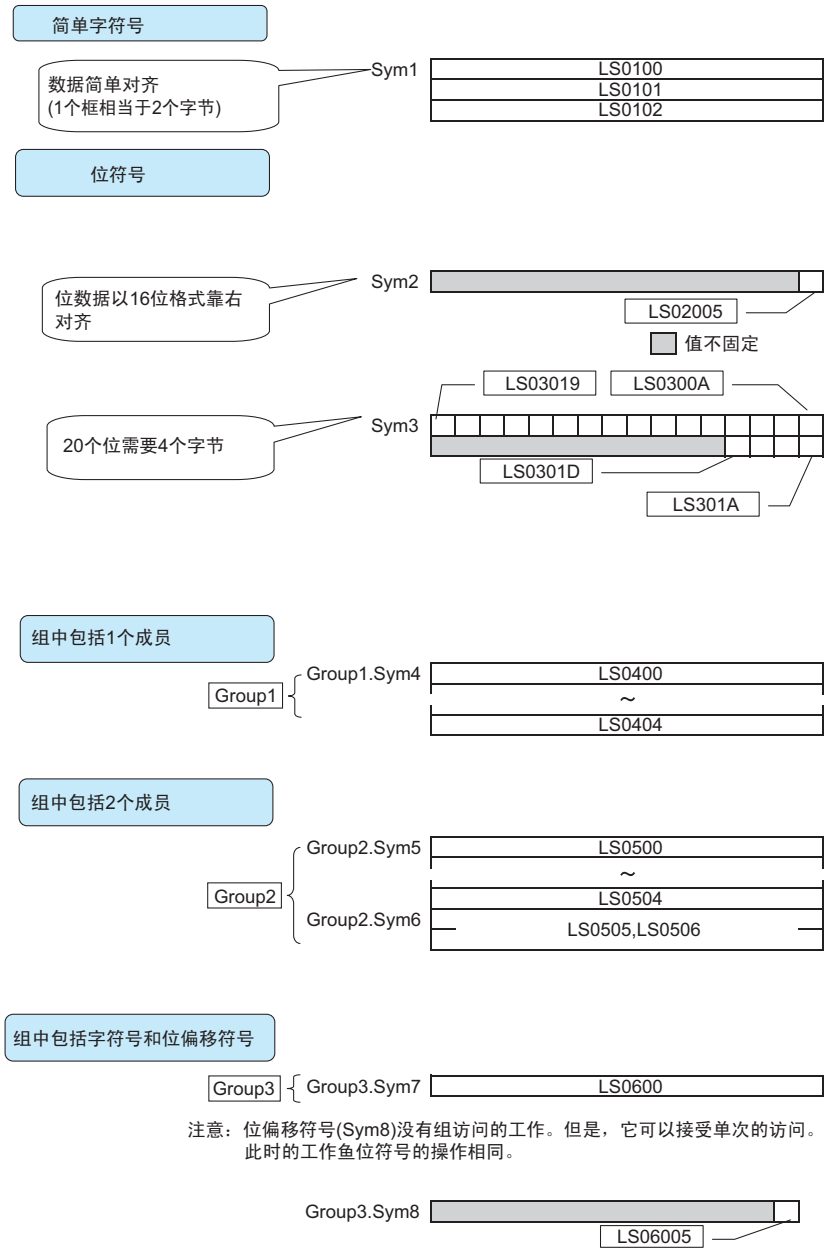


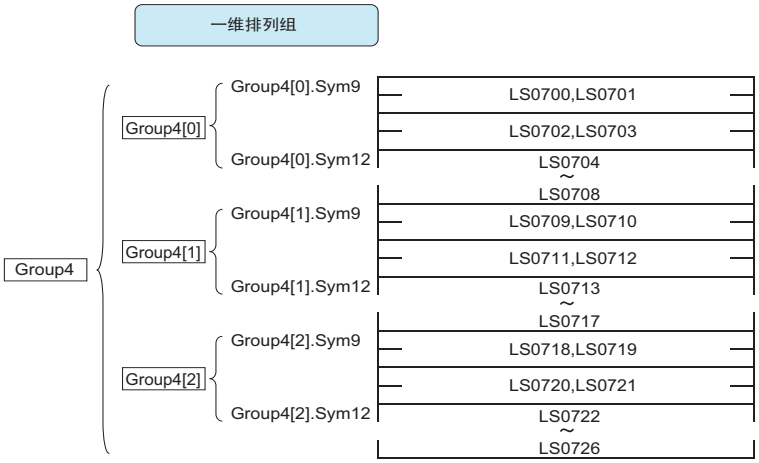
■ 组符号访问的数据结构

Pro-Server EX 通过组符号访问多个寄存器时，数据缓冲区的结构因符号类型或组大小而有所不同。以下是按组符号数据类型列出的数据缓冲区结构：

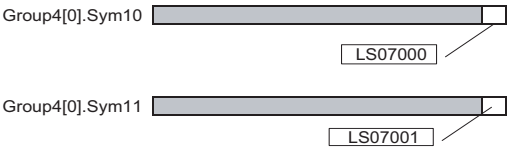
组符号数据类型	缓冲区数据大小
位数据	• 对于位符号 数据缓冲区大小为 16 位的整数倍。 • 对于位偏移符号 无数据缓冲区。
8 位 (有符号) 数据	数据缓冲区占 1 字节。使用二进制值。
8 位 (无符号) 数据	
8 位 (HEX) 数据	
8 位 (BCD) 数据	数据缓冲区占 1 字节。访问寄存器时， Pro-Server EX 执行 BCD 至二进制的转换。
16 位 (有符号) 数据	数据缓冲区占 2 字节 / 寄存器。使用二进制值。
16 位 (无符号) 数据	
16 位 (HEX) 数据	
16 位 (BCD) 数据	数据缓冲区占 2 字节 / 寄存器。访问寄存器时， Pro-Server EX 执行 BCD 至二进制的转换。
32 位 (有符号) 数据	数据缓冲区占 4 字节 / 寄存器。使用二进制值。
32 位 (无符号) 数据	
32 位 (HEX) 数据	
32 位 (BCD) 数据	数据缓冲区占 4 字节 / 寄存器。访问寄存器时， Pro-Server EX 执行 BCD 至二进制的转换。
单精度浮点	数据缓冲区占 4 字节 / 寄存器。将数值作为单精度浮点值处理。
双精度浮点	数据缓冲区占 8 字节 / 寄存器。将数值作为单精度浮点值处理。
字符串数据	数据缓冲区占 1 字节 / 字符。将数据作为以 NULL 终止的字符串。
TIME 数据	数据缓冲区占 1 个寄存器 /4 字节。访问实际的寄存器时，内部格式的二进制值会被转换成有外接控制器格式的值。
TIME_OF_DAY 数据	
DATE 数据	
DATE_AND_TIME 数据	数据缓冲区占 1 个寄存器 /8 字节。访问实际的寄存器时，内部格式的二进制值会被转换成有外接控制器格式的值。

数据缓冲区结构示例如下。

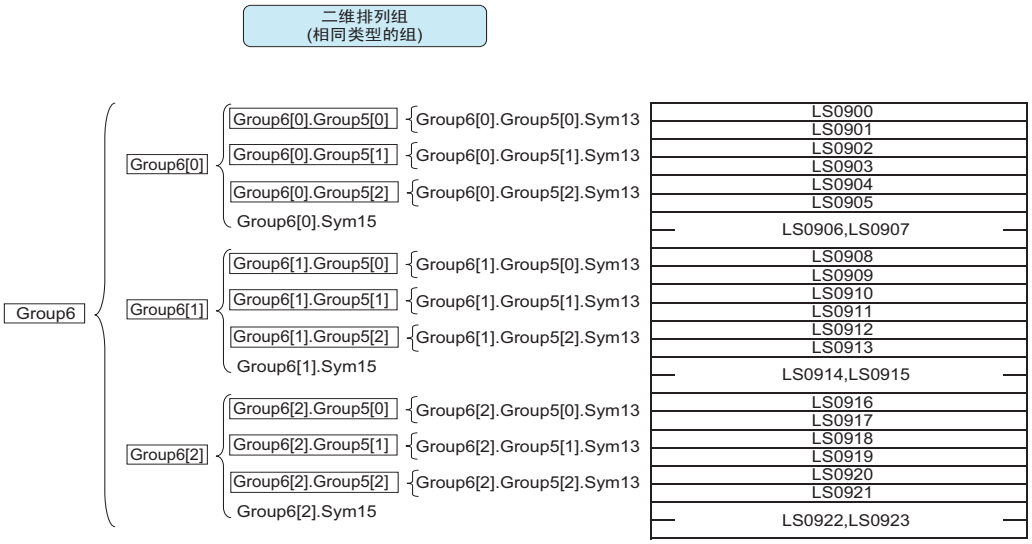

简单字符号
数据简单对齐
(1个框相当于2个字节)
Sym1
LS0100
LS0101
LS0102
位符号
位数据以16位格式靠右
对齐
Sym2
LS02005
值不固定
20个位需要4个字节
Sym3
LS03019
LS0300A
LS0301D
LS301A
组中包括1个成员
Group1
Group1.Sym4
LS0400
~
LS0404
组中包括2个成员
Group2
Group2.Sym5
LS0500
~
LS0504
Group2.Sym6
LS0505,LS0506
组中包括字符号和位偏移符号
Group3
Group3.Sym7
LS0600
注意：位偏移符号(Sym8)没有组访问的工作。但是，它可以接受单次的访问。
此时的工作鱼位符号的操作相同。
Group3.Sym8
LS06005



注意：位偏移符号(Sym10, Sym11)没有组访问的工作。但是，它们可以接受单次的访问。此时的工作鱼位符号的操作相同。

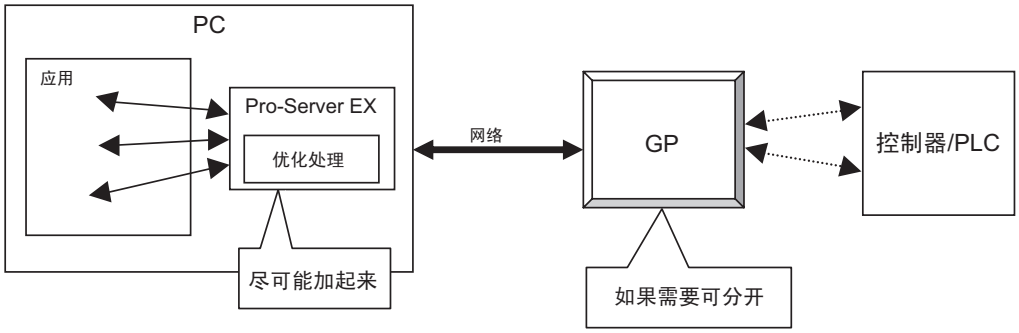


Group4[1].Sym10和Group4[1].Sym11的寄存器地址分别为LS07090和LS07091。
Group4[2].Sym10和Group4[2].Sym11的寄存器地址分别为LS07180和LS07181。



27.1.5 排队访问

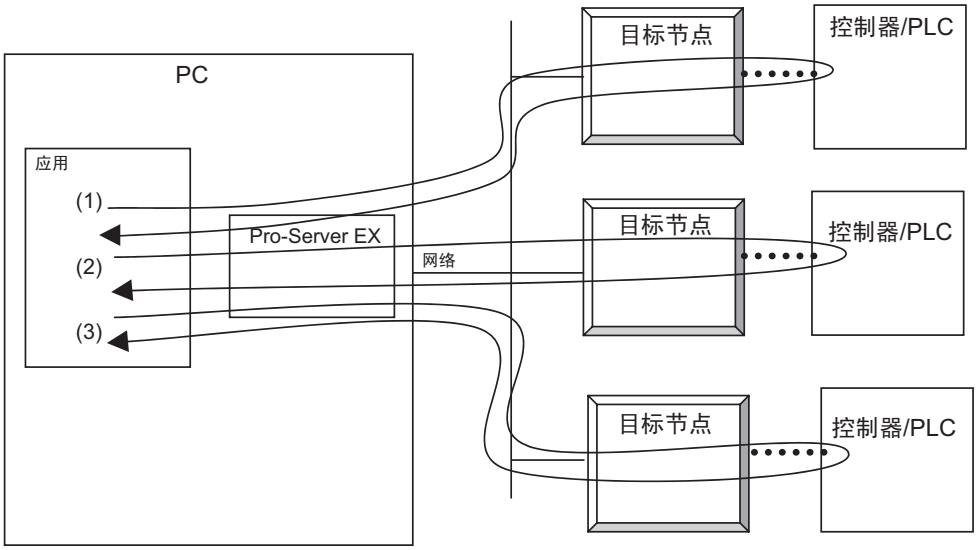
每次调用 API， Pro-Server EX 都会保存一个寄存器访问请求，然后优化保存的多个请求，立刻访问各寄存器。



排队访问的原理

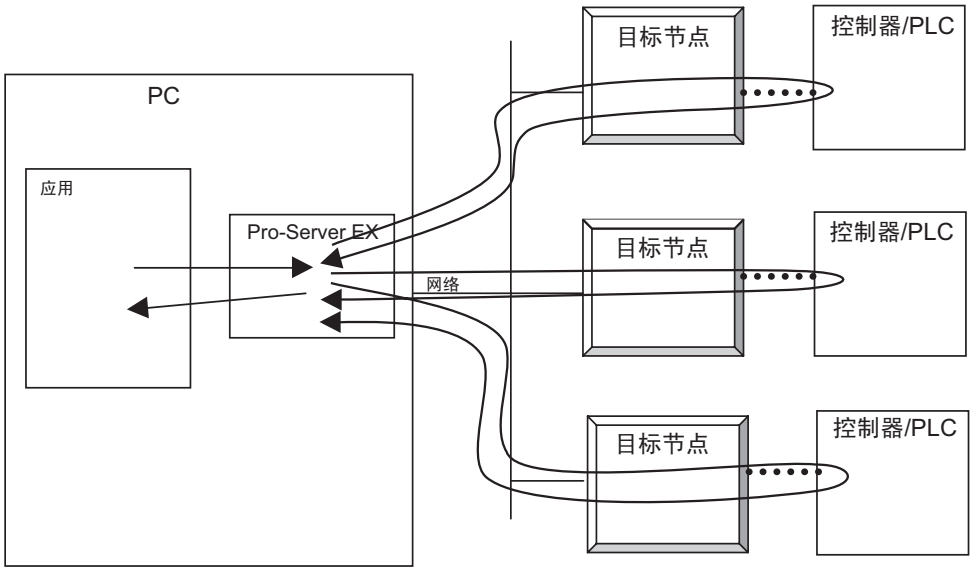
◆ 简单的 API 访问

Pro-Server EX 执行顺序处理。



◆ 排队访问

Pro-Server EX 对各节点执行并行处理。



使用步骤

(1) 声明排队访问开始。 (调用 BeginQueuingRead() 或 BeginQueungWrite()。)

(2) 调用寄存器读取或寄存器写入 API。

(例如：调用 ReadDevice16() 或 WriteDevice16()。)

如果参数正常，不久 API 即会返回， Pro-Server EX 仅保存寄存器访问请求。此步称为 “访问请求注册”。

(3) 实际执行保存的寄存器访问请求，需调用 ExecuteQueuingAccess()。在此步中， Pro-Server EX 优化寄存器访问请求，尝试与寄存器高效率通讯。

如果 Pro-Server EX 成功访问了所有指定的寄存器， ExecuteQueuingAccess() 将返回一个成功代码。反之，如果 Pro-Server EX 未能访问到任何寄存器， ExecuteQueuingAccess() 将返回一个访问错误代码。

如需获知是否成功执行了各寄存器访问请求，可调用 sQueuingAcceessSucceeded() 来检查结果。

重 要

- 在 “访问请求注册”过程中， Pro-Server EX 保存访问数据缓冲区地址 (仅地址，不包含数据)。
  因此运行 “访问请求注册”时，传递给各 API 的数据缓冲区地址必须继续存在，直到调用 ExecuteQueuingAccess() 后返回一个值。
  否则， Pro-Server EX 将访问无效地址，并强制退出。
  另外，再次使用排队访问时，数据缓冲区也必须在 “访问请求注册”指定的地址中。

注 释

- 注册访问请求时， Pro-Server EX 记忆用于访问的数据缓冲区地址。(仅记忆地址，而非数据。)
  因此，
- 使用排队访问时，不能同时注册读取访问和写入访问。例如，声明读取访问排队开始后，不能注册写入访问。同样，声明写入访问排队开始后，不能注册读取访问。
  但是，因为排队访问是为各个 Pro-Server 句柄注册的，所以可以为不同的 Pro-Server 句柄分别注册写入访问和读取访问。
- 访问请求一经注册，再次用相同的方式访问同一寄存器时，无需重新注册。
  由于 Pro-Server EX 按 Pro-Server 句柄保存访问请求，因此每次调用 ExecuteQueuingAccess() 时，将根据保存的数据反复执行访问请求。
  在下述情况下将清除访问请求注册：
  (1) 保存的 Pro-Server 句柄被丢弃时。
  (2) 新的排队访问注册开始时。
  (3) 现有的排队访问注册被取消 (调用了 CancelQueuingAccess())。执行 ExecuteQueuingAccess() 后，再执行的功能如果不是 “将错误代码转换为字符串 (EasyLoadErrorMessage 等)”， Pro-Server EX 将取消现有的排队数据，然后开始新的排队访问注册。

27.1.6 位数据访问

访问位寄存器时， Pro-Server EX 提供以下三种位数据处理方式：

(1) 按 16 位的整数倍处理位数据：将位寄存器作为 16 位整数倍的位串进行处理。
保存指定数量的位数据并从位 D0(最右端) 开始使用。
即使仅指定了一个寄存器，也需要一个 16 位的数据缓冲区。根据指定的寄存器数量，数据缓冲区的大小为 16 位的整数倍。

(示例) 20 位寄存器的数据缓冲区保存顺序

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
16	15	14	13	12	10	11	10	9	8	7	6	5	3	2	1
*	*	*	*	*	*	*	*	*	*	*	*	20	19	18	17

< 可用 API >
为 ReadDeviceBit/WriteDeviceBit()、 ReadDevice/WriteDevice() 或 ReadDeviceVariant/WriteDeviceVariant() 指定数据类型 1(EASY_AppKind_Bit) 时；

为 ReadSymbol/WriteSymbol() 指定位符号或包含位符号的组时

(2) 将位数据作为 Variant BOOL 数据进行处理：将一个位作为 Variant BOOL 数据处理。
数据缓冲区对应每个位处理一个 Variant BOOL 数据。指定了多少寄存器，就有多少 BOOL 数据。

< 可用 API >
为 ReadDeviceVariant/WriteDeviceVariant() 指定数据类型 “0x201” (EASY_AppKind_BOOL) 时；
为 ReadSymbolVariant/WriteSymbolVariant() 指定位符号或包含位符号的组时

(3) 为组符号访问处理位偏移符号
如果通过直接指定位偏移符号访问一个寄存器，数据缓冲区将如上所述处理 “16 位倍数的串”或“Variant BOOL 数据”。
但是，如果用包含一个位偏移符号的组符号访问寄存器，将不在数据缓冲区中为位偏移符号开辟一个数据区。
位偏移符号不能在没有字符号或父符号的情况下单独存在。父符号有数据区，位偏移符号可使用其中的一部分。
详情请参阅 “27.1.4 组访问”。

27.1.7 系统 API

系统 API 用于系统控制，如启动或关闭 Pro-Server EX、载入网络工程文件等。
系统 API 分类如下：

单句柄 API
不指定 Pro-Server 句柄，也能使用 Pro-Server 的功能。
采用此方法，可以同时使用多个 API。（如果尝试同时使用多个 API，会发生两次调用错误。）

多句柄 API
可通过指定 Pro-Server 句柄来使用 Pro-Server 的功能。
指定不同的 Pro-Server 句柄，即可同时使用多个 API。

27.1.8 SRAM 数据访问 API

根据 GP 设置和运行条件， GP 系列内部的 SRAM 保存各种数据。
以下 API 用于访问保存在 SRAM 中的数据。
所有的 SRAM 访问 API 均支持单句柄和多句柄函数。
本节介绍单句柄 API。多句柄 API 在其名称末尾有一个 M，且在第一个参数上会添加一个 Pro-Server 句柄。

27.1.9 CF 卡和 SD 卡 API

是用于访问 CF 卡和 SD 卡数据的 API。
与 SRAM 类似，根据 GP 设置和运行状态保存各种数据。

27.2 寄存器访问 API

■ 单句柄缓存读取 API

函数	位数据
INT WINAPI ReadDeviceBit(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	8 位数据
INT WINAPI ReadDevice8(LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* obData,WORD wCount);	
函数	16 位数据
INT WINAPI ReadDevice16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	32 位数据
INT WINAPI ReadDevice32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
函数	8 位 BCD 数据
INT WINAPI ReadDeviceBCD8(LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* obData,WORD wCount);	
函数	16 位 BCD 数据
INT WINAPI ReadDeviceBCD16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	32 位 BCD 数据
INT WINAPI ReadDeviceBCD32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
函数	单精度浮点数据
INT WINAPI ReadDeviceFloat(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
函数	双精度浮点数据
INT WINAPI ReadDeviceDouble(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
函数	字符串数据
INT WINAPI ReadDeviceStr(LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
函数	通用数据
INT WINAPI ReadDevice(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
函数	通用数据 (Variant 型)
INT WINAPI ReadDeviceVariant(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
函数	组符号
INT WINAPI ReadSymbol(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
函数	组符号 (Variant 型)
INT WINAPI ReadSymbolVariant(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

函数	TIME 数据
INT WINAPI ReadDeviceTIME(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	

函数	DATE 数据
INT WINAPI ReadDeviceDATE(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	TIME_OF_DAY 数据
INT WINAPI ReadDeviceTIME_OF_DAY(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	DATE_AND_TIME 数据
INT WINAPI ReadDeviceDATE_AND_TIME(LPCSTR sNodeName, LPCSTR sDeviceName, QWORD* oqwData, WORD wCount);	

* 有关各参数的详情，请参阅 “■ 读取 / 写入函数的参数”。

* 可将从 TIME、 DATE、 TIME_OF_DAY 和 DATE_AND_TIME 数据中读取的二进制值转换为文本格式。有关文本转换的更多信息，请参阅 "27.8 二进制日期和时间 / 文本显示转换 "。

■ 单句柄直接读取 API

函数	位数据
INT WINAPI ReadDeviceBitD(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	8 位数据
INT WINAPI ReadDevice8D(LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* obData,WORD wCount);	
函数	16 位数据
INT WINAPI ReadDevice16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	32 位数据
INT WINAPI ReadDevice32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
函数	8 位 BCD 数据
IINT WINAPI ReadDeviceBCD8D(LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* obData,WORD wCount);	
函数	16 位 BCD 数据
INT WINAPI ReadDeviceBCD16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	32 位 BCD 数据
INT WINAPI ReadDeviceBCD32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
函数	单精度浮点数据
INT WINAPI ReadDeviceFloatD(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
函数	双精度浮点数据
INT WINAPI ReadDeviceDoubleD(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
函数	字符串数据

INT WINAPI ReadDeviceStrD(LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
函数	通用数据
INT WINAPI ReadDeviceD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
函数	通用数据 (Variant 型)
INT WINAPI ReadDeviceVariantD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
函数	组符号
INT WINAPI ReadSymbolD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
函数	组符号 (Variant 型)
INT WINAPI ReadSymbolVariantD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	
函数	TIME 数据
INT WINAPI ReadDeviceTIMED(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	DATE 数据
INT WINAPI ReadDeviceDATED(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	TIME_OF_DAY 数据
INT WINAPI ReadDeviceTIME_OF_DAYD(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	DATE_AND_TIME 数据
INT WINAPI ReadDeviceDATE_AND_TIMED(LPCSTR sNodeName, LPCSTR sDeviceName, QWORD* oqwData, WORD wCount);	

* 有关各参数的详情，请参阅 “■ 读取 / 写入函数的参数”。

* 可将从 TIME、 DATE、 TIME_OF_DAY 和 DATE_AND_TIME 数据中读取的二进制值转换为文本格式。有关文本转换的更多信息，请参阅 "27.8 二进制日期和时间 / 文本显示转换 "。

■ 单句柄直接写入 API

函数	位数据
INT WINAPI WriteDeviceBitD(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	8 位数据
INT WINAPI WriteDevice8D(LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* pbData,WORD wCount);	
函数	16 位数据
INT WINAPI WriteDevice16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	32 位数据
INT WINAPI WriteDevice32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
函数	8 位 BCD 数据

INT WINAPI WriteDeviceBCD8D(LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* pbData,WORD wCount);	
函数	16 位 BCD 数据
INT WINAPI WriteDeviceBCD16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	32 位 BCD 数据
INT WINAPI WriteDeviceBCD32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
函数	单精度浮点数据
INT WINAPI WriteDeviceFloatD(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
函数	双精度浮点数据
INT WINAPI WriteDeviceDoubleD(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
函数	字符串数据
INT WINAPI WriteDeviceStrD(LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
函数	通用数据
INT WINAPI WriteDeviceD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
函数	通用数据 (Variant 型)
INT WINAPI WriteDeviceVariantD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
函数	组符号
INT WINAPI WriteSymbolD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
函数	组符号 (Variant 型)
INT WINAPI WriteSymbolVariantD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	
函数	TIME 数据
INT WINAPI WriteDeviceTIMED(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	DATE 数据
INT WINAPI WriteDeviceDATED(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	TIME_OF_DAY 数据
INT WINAPI WriteDeviceTIME_OF_DAYD(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	DATE_AND_TIME 数据
INT WINAPI WriteDeviceDATE_AND_TIMED(LPCSTR sNodeName, LPCSTR sDeviceName, QWORD* pqwData, WORD wCount);	

* 有关各参数的详情，请参阅 “■ 读取 / 写入函数的参数”。

* 可将写入 TIME、 DATE、 TIME_OF_DAY 和 DATE_AND_TIME 数据的二进制值转换为文本格式。有关文本转换的更多信息，请参阅 "27.8 二进制日期和时间 / 文本显示转换 "。

■ 写入后刷新缓存的单句柄写入 API

函数	位数据
INT WINAPI WriteDeviceBit(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	8 位数据
INT WINAPI WriteDevice8(LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* pbData,WORD wCount);	
函数	16 位数据
INT WINAPI WriteDevice16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	32 位数据
INT WINAPI WriteDevice32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
函数	8 位 BCD 数据
INT WINAPI WriteDeviceBCD8(LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* pbData,WORD wCount);	
函数	16 位 BCD 数据
INT WINAPI WriteDeviceBCD16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	32 位 BCD 数据
INT WINAPI WriteDeviceBCD32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
函数	单精度浮点数据
INT WINAPI WriteDeviceFloat(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
函数	双精度浮点数据
INT WINAPI WriteDeviceDouble(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
函数	字符串数据
INT WINAPI WriteDeviceStr(LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
函数	通用数据
INT WINAPI WriteDevice(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
函数	通用数据 (Variant 型)
INT WINAPI WriteDeviceVariant(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
函数	组符号
INT WINAPI WriteSymbol(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
函数	组符号 (Variant 型)
INT WINAPI WriteSymbolVariant(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

函数	TIME 数据
INT WINAPI WriteDeviceTIME(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	DATE 数据
INT WINAPI WriteDeviceDATE(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	TIME_OF_DAY 数据
INT WINAPI WriteDeviceTIME_OF_DAY(LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	DATE_AND_TIME 数据
INT WINAPI WriteDeviceDATE_AND_TIME(LPCSTR sNodeName, LPCSTR sDeviceName, QWORD* pqwData, WORD wCount);	

* 有关各参数的详情，请参阅 “■ 读取 / 写入函数的参数”。

* 可将写入 TIME、 DATE、 TIME_OF_DAY 和 DATE_AND_TIME 数据的二进制值转换为文本格式。有关文本转换的更多信息，请参阅 "27.8 二进制日期和时间 / 文本显示转换 "。

■ 多句柄缓存读取 API

函数	位数据
INT WINAPI ReadDeviceBitM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	8 位数据
INT WINAPI ReadDevice8M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* obData,WORD wCount);	
函数	16 位数据
INT WINAPI ReadDevice16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	32 位数据
INT WINAPI ReadDevice32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
函数	8 位 BCD 数据
INT WINAPI ReadDeviceBCD8M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* obData,WORD wCount);	
函数	16 位 BCD 数据
INT WINAPI ReadDeviceBCD16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	32 位 BCD 数据
INT WINAPI ReadDeviceBCD32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
函数	单精度浮点数据
INT WINAPI ReadDeviceFloatM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
函数	双精度浮点数据

INT WINAPI ReadDeviceDoubleM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
函数	字符串数据
INT WINAPI ReadDeviceStrM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
函数	通用数据
INT WINAPI ReadDeviceM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
函数	通用数据 (Variant 型)
INT WINAPI ReadDeviceVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
函数	组符号
INT WINAPI ReadSymbolM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
函数	组符号 (Variant 型)
INT WINAPI ReadSymbolVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	
函数	TIME 数据
INT WINAPI ReadDeviceTIMEM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	DATE 数据
INT WINAPI ReadDeviceDATEM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	TIME_OF_DAY 数据
INT WINAPI ReadDeviceTIME_OF_DAYM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	DATE_AND_TIME 数据
INT WINAPI ReadDeviceDATE_AND_TIMEM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, QWORD* oqwData, WORD wCount);	

* 有关各参数的详情，请参阅 “■ 读取 / 写入函数的参数”。

* 可将从 TIME、 DATE、 TIME_OF_DAY 和 DATE_AND_TIME 数据中读取的二进制值转换为文本格式。有关文本转换的更多信息，请参阅 "27.8 二进制日期和时间 / 文本显示转换 "。

■ 多句柄直接读取 API

函数	位数据
INT WINAPI ReadDeviceBitDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	8 位数据
INT WINAPI ReadDevice8DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* obData,WORD wCount);	
函数	16 位数据

INT WINAPI ReadDevice16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	32 位数据
INT WINAPI ReadDevice32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
函数	8 位 BCD 数据
INT WINAPI ReadDeviceBCD8DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* obData,WORD wCount);	
函数	16 位 BCD 数据
INT WINAPI ReadDeviceBCD16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
函数	32 位 BCD 数据
INT WINAPI ReadDeviceBCD32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
函数	单精度浮点数据
INT WINAPI ReadDeviceFloatDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
函数	双精度浮点数据
INT WINAPI ReadDeviceDoubleDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
函数	字符串数据
INT WINAPI ReadDeviceStrDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
函数	通用数据
INT WINAPI ReadDeviceDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
函数	通用数据 (Variant 型)
INT WINAPI ReadDeviceVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
函数	组符号
INT WINAPI ReadSymbolDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
函数	组符号 (Variant 型)
INT WINAPI ReadSymbolVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	
函数	TIME 数据
INT WINAPI ReadDeviceTIMEDM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	DATE 数据
INT WINAPI ReadDeviceDATEDM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	TIME_OF_DAY

INT WINAPI ReadDeviceTIME_OF_DAYDM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* odwData, WORD wCount);	
函数	DATE_AND_TIME 数据
INT WINAPI ReadDeviceDATE_AND_TIMEDM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, QWORD* oqwData, WORD wCount);	

* 有关各参数的详情，请参阅 “■ 读取 / 写入函数的参数”。

* 可将从 TIME、 DATE、 TIME_OF_DAY 和 DATE_AND_TIME 数据中读取的二进制值转换为文本格式。有关文本转换的更多信息，请参阅 "27.8 二进制日期和时间 / 文本显示转换 "。

■ 多句柄直接写入 API

函数	位数据
INT WINAPI WriteDeviceBitDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	8 位数据
INT WINAPI WriteDevice8DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* pbData,WORD wCount);	
函数	16 位数据
INT WINAPI WriteDevice16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	32 位数据
INT WINAPI WriteDevice32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
函数	8 位 BCD 数据
INT WINAPI WriteDeviceBCD8DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* pbData,WORD wCount);	
函数	16 位 BCD 数据
INT WINAPI WriteDeviceBCD16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	32 位 BCD 数据
INT WINAPI WriteDeviceBCD32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
函数	单精度浮点数据
INT WINAPI WriteDeviceFloatDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
函数	双精度浮点数据
INT WINAPI WriteDeviceDoubleDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
函数	字符串数据
INT WINAPI WriteDeviceStrDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
函数	通用数据
INT WINAPI WriteDeviceDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	

函数	通用数据 (Variant 型)
INT WINAPI WriteDeviceVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
函数	组符号
INT WINAPI WriteSymbolDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
函数	组符号 (Variant 型)
INT WINAPI WriteSymbolVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	
函数	TIME 数据
INT WINAPI WriteDeviceTIMEDM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	DATE 数据
INT WINAPI WriteDeviceDATEDM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	TIME_OF_DAY 数据
INT WINAPI WriteDeviceTIME_OF_DAYDM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	DATE_AND_TIME 数据
INT WINAPI WriteDeviceDATE_AND_TIMEDM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, QWORD* pqwData, WORD wCount);	

* 有关各参数的详情，请参阅 “■ 读取 / 写入函数的参数”。

* 可将写入 TIME、 DATE、 TIME_OF_DAY 和 DATE_AND_TIME 数据的二进制值转换为文本格式。有关文本转换的更多信息，请参阅 "27.8 二进制日期和时间 / 文本显示转换 "。

■ 写入后刷新缓存的多句柄写入 API

函数	位数据
INT WINAPI WriteDeviceBitM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	8 位数据
INT WINAPI WriteDevice8M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* pbData,WORD wCount);	
函数	16 位数据
INT WINAPI WriteDevice16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	32 位数据
INT WINAPI WriteDevice32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
函数	8 位 BCD 数据
INT WINAPI WriteDeviceBCD8M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,BYTE* pbData,WORD wCount);	
函数	16 位 BCD 数据

INT WINAPI WriteDeviceBCD16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
函数	32 位 BCD 数据
INT WINAPI WriteDeviceBCD32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
函数	单精度浮点数据
INT WINAPI WriteDeviceFloatM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
函数	双精度浮点数据
INT WINAPI WriteDeviceDoubleM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
函数	字符串数据
INT WINAPI WriteDeviceStrM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
函数	通用数据
INT WINAPI WriteDeviceM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
函数	通用数据 (Variant 型)
INT WINAPI WriteDeviceVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
函数	组符号
INT WINAPI WriteSymbolM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
函数	组符号 (Variant 型)
INT WINAPI WriteSymbolVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	
函数	TIME 数据
INT WINAPI WriteDeviceTIMEM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	DATE 数据
INT WINAPI WriteDeviceDATEM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	TIME_OF_DAY 数据
INT WINAPI WriteDeviceTIME_OF_DAYM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, DWORD* pdwData, WORD wCount);	
函数	DATE_AND_TIME 数据
INT WINAPI WriteDeviceDATE_AND_TIMEM(HANDLE hProServer, LPCSTR sNodeName, LPCSTR sDeviceName, QWORD* pqwData, WORD wCount);	

* 有关各参数的详情，请参阅 “■ 读取 / 写入函数的参数”。

* 可将写入 TIME、 DATE、 TIME_OF_DAY 和 DATE_AND_TIME 数据的二进制值转换为文本格式。有关文本转换的更多信息，请参阅 "27.8 二进制日期和时间 / 文本显示转换 "。

■ 读取 / 写入函数的参数

< 参数 >

bsNodeName　　：节点名称 (字符串)

直接指定 Pro-Studio EX 中注册的入口节点名称或 IP 地址。
例：1) 指定节点名称：“AGP”
例：2) 直接指定 IP 地址：“192.9.201.1”

bsDeviceName　　：读取 / 写入函数中的符号 (字符串)

直接指定 Pro-Studio EX 中注册的符号名称或寄存器地址。
例：1) 指定符号名称：　　“SWITCH1”
例：2) 直接指定寄存器地址：　　“M100”

函数	符号数据类型													
	位	8 位		16 位		32 位		浮点	双精度	字符串	TIME	DATE	TIME_OF_DAY	DATE_AND_TIME
		S/U/HEX	BCD	S/U/HEX	BCD	S/U/HEX	BCD							
XXXDeviceBit	0	-	-	-	-	-	-	-	-	-	-	-	-	-
XXXDevice8	-	0	-	-	-	-	-	-	-	-	-	-	-	-
XXXDevice16	-	-	-	0	-	-	-	-	-	-	-	-	-	-
XXXDevice32	-	-	-	-	-	0	-	-	-	-	-	-	-	-
XXXDeviceBCD8	-	-	0	-	-	-	-	-	-	-	-	-	-	-
XXXDeviceBCD16	-	-	-	-	0	-	-	-	-	-	-	-	-	-
XXXDeviceBCD32	-	-	-	-	-	-	0	-	-	-	-	-	-	-
XXXDeviceFloat	-	-	-	-	-	-	-	0	-	-	-	-	-	-
XXXDeviceDouble	-	-	-	-	-	-	-	-	0	-	-	-	-	-
XXXDeviceStr	-	-	-	-	-	-	-	-	-	0	-	-	-	-
XXXDevice	0	0	0	0	0	0	0	0	0	0	0	0	0	0
XXXDeviceTIME	-	-	-	-	-	-	-	-	-	-	0	-	-	-
XXXDeviceDATE	-	-	-	-	-	-	-	-	-	-	-	0	-	-
XXXDeviceTIME_OF_DAY	-	-	-	-	-	-	-	-	-	-	-	-	0	-
XXXDeviceDATE_AND_TIME	-	-	-	-	-	-	-	-	-	-	-	-	-	0

pxxData　　：读取 / 写入目标数据

可访问的数据类型和对应的参数类型列表如下。

可访问的数据类型	参数类型
位数据	WORD * pwData
8 位数据	BYTE * pbData
16 位数据	WORD * pwData
32 位数据	DWORD * pdwData
8 位 BCD 数据	BYTE * pbData
16 位 BCD 数据	WORD * pwData
32 位 BCD 数据	DWORD * pdwData
单精度浮点数据	FLOAT * pflData
双精度浮点数据	DOUBLE * pdbData
字符串数据	LPTSTR psData
通用数据	LPVOID pData
通用数据 (用于 VB)	LPVARIANT pData
TIME 数据	DWORD * pdwData
DATE 数据	DWORD * pdwData
TIME_OF_DAY 数据	DWORD * pdwData
DATE_AND_TIME 数据	QWORD * pdwData

wCount　　：读取 / 写入目标数据的数量

使用 Read/WriteDeviceStr 函数，以字节数计算字符串数据。对于 16 位的寄存器符号，指定两个字符的倍数；对于 32 位的寄存器符号，指定 4 个字符的倍数。
读取 / 写入函数可处理的最大数据量如下所示：

可访问的数据类型	读取	写入
位数据	255	255
8 位数据	1020	1020
16 位数据	1020	1020
32 位数据	510	510
8 位 BCD 数据	1020	1020
16 位 BCD 数据	1020	1020
32 位 BCD 数据	510	510
单精度浮点数据	510	510
双精度浮点数据	255	255

可访问的数据类型	读取	写入
字符串数据	2040 个字符 (单字节)	2040 个字符 (单字节)
TIME 数据	510	510
DATE 数据	510	510
TIME_OF_DAY 数据	510	510
DATE_AND_TIME 数据	255	255

wAppKind ：数据类型指定

值	数据类型	值	数据类型
1	位	11	双精度
2	有符号 16 位	12	字符串
3	无符号 16 位	13	有符号 8 位
4	HEX16 位	14	无符号 8 位
5	BCD16 位	15	HEX8 位
6	有符号 32 位	16	BCD 8 位
7	无符号 32 位	17	TIME
8	HEX32 位	18	DATE
9	BCD32 位	19	TIME_OF_DAY
10	浮点	20	DATE_AND_TIME (*)

* 不能与 VB 函数一起使用。

在读取 / 写入寄存器函数中，数据类型是用参数指定的。因此，数据类型可动态改变。

< 返回值 >
正常结束：0
异常结束：错误代码

< 特别注意事项 >
当使用 Read/WriteDeviceBit 函数时：
pwData 保存 wCount 所指定数量的数据，从 D0 位开始连续进行。
示例：wCount 为 “20”

	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
PwData	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
PwData+1	*	*	*	*	*	*	*	*	*	*	*	*	20	19	18	17

读取 / 写入多个连续位数据时，用 Read/Write/Device 8、 16 和 32 函数比用 Read/WriteDeviceBit 函数更为有效。
用 “*” (星号) 表示的位保存的是未定义的值。请在应用程序中屏蔽这些区域。

使用 Read/WriteDeviceBCD8、 Read/WriteDeviceBCD16 或 Read/WriteDeviceBCD32 函数时：
如果目标控制器 /PLC 处理 BCD 数据，则可以使用这些函数。但是，用这些函数传递的数据 (pxxData 的内容) 会被作为二进制处理，而不是 BCD 数据。 (Pro-Server EX 在内部执行 BCD 转换。) 不能处理负值。

函数	十进制形式	十六进制形式
Read/WriteDeviceBCD8	0~99	00 至 63
Read/WriteDeviceBCD16	0~9999	0000~270F
Read/WriteDeviceBCD32	0~99999999	00000000~05F5E0FF

使用字符串数据函数时：
为接收变量的字符串数据，需要留出足够的数据存储区。

27.3 缓冲区控制 API

函数	创建缓冲区

为提高寄存器读取的处理速度 Pro-Server EX 内嵌了寄存器数据缓冲存储函数 (带有复制功能)。此 API 用于创建缓冲区。
此 API 仅定义缓冲区。如需定义要缓冲存储哪个寄存器，请使用 PS_EntryCacheRecord()。

单个
INT WINAPI PS_CreateCache(LPCSTR sCacheName, DWORD dwPollingTime);
多个
INT WINAPI PS_CreateCacheM(HANDLE hProServer, LPCSTR sCacheName, DWORD dwPollingTime);

参数	**返回值**
sCacheName：(入) 缓冲区名称 dwPollingTime：(入) 若要选择保持监视方式，请指定 “0”。 缓冲区会尽快得到更新。 如果指定非 “0” 值，选择的即为轮询方式。请以毫秒为单位指定轮询周期 (缓存更新周期)。	正常结束：0 异常结束：错误代码

特别注意事项
- 一个 Pro-Server EX 程序最多可以创建 1000 个缓冲区。
- 用 Pro-Studio EX 创建网络工程文件时，可以直接使用已注册的缓冲区。不需要再用此 API 重新创建。

函数	将记录注册到缓冲区

将缓冲寄存器 (缓冲源寄存器) 注册到用 PS_CreateCache() 创建的缓冲区。

对于 GP 系列节点或 Pro-Server EX 节点， Pro-Server EX 不支持用保持监视方式更新缓冲区。
因此，对于采用保持监视方式 (用 PS_CreateCache() 创建缓冲区时将 dwPollingTime 设置为 “0”) 的缓冲区，如果用此 API 指定了 GP 系列节点或 Pro-Server EX 节点，将发生错误。

单个
INT WINAPI PS_EntryCacheRecord(LPCSTR sCacheName, LPCSTR sNodeName, LPCSTR sDevice, WORD wAppKind, WORD wCount);
多个
INT WINAPI PS_EntryCacheRecordM(HANDLE hProServer, LPCSTR sCacheName, LPCSTR sNodeName, LPCSTR sDevice, WORD wAppKind, WORD wCount);

参数

sCacheName：(入) 缓冲区名称

将缓冲源寄存器注册到用此名称指定的缓冲区。

sNodeName：(入) 带有缓冲源控制器 /PLC 名称的入口节点名称

sDevice：(入) 缓冲源寄存器

指定缓冲源寄存器时，可以直接指定寄存器地址，或指定用 Pro-Studio EX 注册的符号或组。指定组可以立刻注册多个符号。

wAppKind：(入) 源寄存器数据类型

可用数据类型因缓冲源寄存器的指定方式而不同。

a) 直接指定缓冲源寄存器的寄存器地址时：

指定 Pro-Server EX 支持的数据类型 (1 ~ 20)。不能指定 “0”。

值	数据类型	值	数据类型
1	位	11	双精度浮点
2	16 位有符号十进制	12	字符串
3	16 位无符号十进制	13	8 位有符号数据
4	16 位十六进制	14	8 位无符号数据
5	16 位 BCD	15	8 位 (HEX) 数据
6	32 位有符号十进制	16	8 位 (BCD) 数据
7	32 位无符号十进制	17	TIME 数据
8	32 位十六进制	18	TIME_OF_DAY 数据
9	32 位 BCD	19	DATE 数据
10	单精度浮点	20	DATE_AND_TIME 数据

b) 为缓冲源寄存器指定符号时：

指定 Pro-Server EX 支持的数据类型 (0 ~ 20)。如果指定 “0”，则使用符号定义中指定的符号类型。

c) 为缓冲源寄存器指定组时：

固定为 “0”。

符号类型为指定组中的所有符号注册。

wCount ：(入) 缓冲存储的寄存器数据量

可用值因缓冲源寄存器的指定方式而不同。

a) 直接指定缓冲源寄存器的寄存器地址时：

数据量为 1 ~ 2040，取决于可使用的寄存器类型。(最大值取决于寄存器类型。)

b) 为缓冲源寄存器指定符号时：

如果指定 “0”，则使用符号定义中指定的数量。

如果指定非 0 值，则数据量 (1 ~ 2040) 取决于可使用的寄存器类型。(最大值取决于寄存器类型。)

c) 为缓冲源寄存器指定组时：

固定为 “0”。

指定组中的所有符号均用于缓冲存储。

返回值

正常结束：0

异常结束：错误代码

特别注意事项

函数	启动缓冲存储

启动缓冲存储。

单个
INT WINAPI PS_StartCache(LPCSTR sCacheName);
多个
INT WINAPI PS_StartCacheM(HANDLE hProServer, LPCSTR sCacheName);

参数	**返回值**
sCacheName：(入) 启动的缓冲区名称 也可以指定用 Pro-Studio EX 注册的缓冲区名称。	正常结束：0 异常结束：错误代码

特别注意事项

函数	停止缓冲存储

暂时停止缓冲存储。
缓冲存储停止，但仍保留缓冲区定义。
如需重启缓冲存储，请调用 PS_StartCache()。

单个
INT WINAPI PS_StopCache(LPCSTR sCacheName);
多个
INT WINAPI PS_StopCacheM(HANDLE hProServer, LPCSTR sCacheName);

参数	**返回值**
sCacheName：(入) 停止的缓冲区名称 也可以指定用 Pro-Studio EX 注册的缓冲区名称。	正常结束：0 异常结束：错误代码

特别注意事项

函数	检查缓冲存储状态

检查缓冲存储状态。

单个
INT WINAPI PS_GetCacheStatus(LPCSTR sCacheName);
多个
INT WINAPI PS_GetCacheStatusM(HANDLE hProServer, LPCSTR sCacheName);

参数	**返回值**
sCacheName：(入) 要检查的缓冲区名称 也可以指定用 Pro-Studio EX 注册的缓冲区名称。	0：已创建缓冲区，但尚未启动。 1: 缓冲存储过程中 2: 缓冲存储暂停 XX: 错误代码

特别注意事项

函数	丢弃缓冲区

停止缓冲存储，并丢弃缓冲区。

单个
INT WINAPI PS_DestroyCache(LPCSTR sCacheName);
多个
INT WINAPI PS_DestroyCacheM(HANDLE hProServer, LPCSTR sCacheName);

参数	**返回值**
sCacheName：(入) 要丢弃的缓冲区名称 也可以指定用 Pro-Studio EX 注册的缓冲区名称。	正常结束：0 异常结束：错误代码

函数	设置缓冲区更新通知函数

设置函数，向指定窗口通知缓冲区更新状态。

从应用程序缓冲读取一个寄存器时，即使频繁执行读取，如果不更新缓存数据的话，数据也不会有变化。
缓存数据得到更新时 (使用保持监视方式时，至少一个目标寄存器发生变化；使用轮询方式时，一个轮询周期完成)， Pro-Server EX 能向指定窗口发送消息。
如果系统设计意图即是要在收到此消息后执行对寄存器的缓冲读取，则可以提高系统效率。
使用此 API，可以在 Pro-Server EX 中设置 “目标缓冲区名称”、“接收消息的窗口” 和 “消息内容”。
这些设置正常完成后， API 返回一个 ID，用于识别当前设置的通知函数。

单个
INT WINAPI PS_SetNotifyFromCache(LPCSTR sCacheName, HWND hWnd, UINT message, WPARAM WParam, LPARAM LParam, HANDLE* ohCacheNotifyID);
多个
INT WINAPI PS_SetNotifyFromCacheM(HANDLE hProServer, LPCSTR sCacheName, HWND hWnd, UINT message, WPARAM WParam, LPARAM LParam, HANDLE* ohCacheNotifyID);

参数	**返回值**
sCacheName：(入) 缓冲区名称 也可以指定用 Pro-Studio EX 注册的缓冲区名称。 hWnd：(入) 接收消息窗口的句柄 message：(入) 准备发送到窗口的消息 ID wParam：(入) 准备和消息 ID 一同发送到窗口的 WPARAM 值 LParam：(入) 准备和消息 ID 一同发送到窗口的 LPARAM 值 ohCacheNotifyID：(出) 返回当前设置通知函数的识别 ID。	正常结束：0 异常结束：错误代码

特别注意事项
如果不再需要返回的句柄，请用 PS_KillNotifyFromCache() 将其丢弃。
缓冲区得到更新后，调用 PostMessage()，将消息 (由第二个参数指定)、 wParam 值 (由第三个参数指定) 和 LParam 值 (由第四个参数指定) 发送到目标窗口 (hWnd)。
有关 PostMessage() 的详情，请参阅 Windows API 手册。

函数	接受下一缓冲区更新通知

接受下一缓冲区更新通知

Pro-Server EX 提供在缓冲区得到更新后向指定窗口发送消息的函数。但是，执行该通知函数后，即使缓冲区随后即得到了更新，也只有在再次调用该 API 时， Pro-Server EX 才会发送消息。这是因为，如果处理一个通知程序花费了很长时间，当 Pro-Server EX 发送下一缓冲区更新消息时，将发生多次调用相关程序的错误。(如果通知程序在完成处理前收到了下一条消息，则发生多次调用程序错误。)
为避免发生这一错误，此 API 明确通知 Pro-Server EX 可以发送一下消息。
在通知程序处理结束时调用此 API，则可以实现在每次缓冲区得到更新时进行连续处理。

单个
INT WINAPI PS_AcceptNextNotifyFromCache(HANDLE hCacheNotifyID);
多个
INT WINAPI PS_AcceptNextNotifyFromCacheM(HANDLE hProServer, HANDLE hCacheNotifyID);

参数	**返回值**
hCacheNotifyID：(入) 下一消息接受通知函数的 ID ID 由 PS_SetNotifyFromCache() 获得	正常结束：0 异常结束：错误代码

特别注意事项

函数	取消缓冲区更新通知

取消向指定窗口发送缓冲区更新消息的函数。

取消后，即使与 hCacheNotifyID 相关的缓冲区得到更新， Pro-Server EX 也不会向相关窗口发送缓冲区更新消息。

单个
INT WINAPI PS_KillNotifyFromCache(HANDLE hCacheNotifyID);
多个
INT WINAPI PS_KillNotifyFromCacheM(HANDLE hProServer, HANDLE hCacheNotifyID);

参数	**返回值**
hCacheNotifyID：(入) 要取消的通知函数的 ID ID 由 PS_SetNotifyFromCache() 获得	正常结束：0 异常结束：错误代码

特别注意事项
此 API 不会收取或丢弃 Pro-Server EX 发送的消息，即使消息保留在窗口中。因此，如果 Pro-Server EX 已向某窗口发送了一条消息，且在调用此 API 前应用程序尚未收取这条消息，应用程序甚至可以在调用此 API 后再从窗口中收取消息。(根据执行时机，可能会在调用此 API 后才调用通知程序。)

函数	获取缓冲区更新次数

返回缓冲区更新次数。

通过监视程序中的更新次数，可以查看缓冲区是否得到更新。
使用此函数，可以省去对寄存器缓冲读取 API 不必要的调用。（对于没有变化的寄存器，即使调用了寄存器缓冲读取 API，值也不会发生变化。）

单个
INT WINAPI PS_GetUpdateCounter(LPCSTR sCacheName, DWORD* odwCount);
多个
INT WINAPI PS_GetUpdateCounterM(HANDLE hProServer, LPCSTR sCacheName, DWORD* odwCount);

参数	**返回值**
sCacheName：（入）要监视的缓冲区名称 也可以指定用 Pro-Studio EX 注册的缓冲区名称。 odwCount：（外）缓冲区更新次数 对更新次数进行无休止的计数，范围从 0~4294967295。 （计数达到 4294967295 后，返回 “0”。）	正常结束：0 异常结束：错误代码

特别注意事项

27.4 排队访问控制 API

函数	开始对寄存器读取请求进行排队
调用此 API 后， Pro-Server EX 开始对寄存器读取请求进行排队，直到调用 ExecuteQueuingAccess()。 将对应于各个 Pro-Server 句柄执行排队。 单个 INT WINAPI BeginQueuingRead() ; 多个 INT WINAPI BeginQueuingReadM(HANDLE hProServer);	
参数	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项 • 在 BeginQueuingRead() 后面调用 ExecuteQueuingAccess() 之前，请勿调用寄存器写入 API。调用 BeginQueuingRead() 后， Pro-Server EX 对缓冲读取或直接读取请求进行排队。但是，不能将缓冲读取和直接读取请求排在一起。 • 如需丢弃请求队列，请调用 CancelQueuingAccess()。 • 队列中最多可包含 1500 个请求，数据大小最多为 1M 字节。	

函数	开始对寄存器写入请求进行排队
调用此 API 后， Pro-Server EX 开始对寄存器写入请求进行排队，直到调用 ExecuteQueuingAccess()。 将对应于各个 Pro-Server 句柄执行排队。 单个 INT WINAPI BeginQueuingWrite() ; 多个 INT WINAPI BeginQueuingWriteM(HANDLE hProServer);	
参数	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项 • 在 BeginQueuingWrite() 后面调用 ExecuteQueuingAccess() 之前，请勿调用寄存器读取 API。调用 BeginQueuingWrite() 后， Pro-Server EX 对缓冲写入或直接写入请求进行排队。但是，不能将缓冲写入和直接写入请求排在一起。 • 如需丢弃请求队列，请调用 CancelQueuingAccess()。 • 队列中最多可包含 1500 个请求，数据大小最多为 1M 字节。	
特别注意事项	

函数	执行寄存器读取 / 写入请求队列
根据寄存器读取 / 写入请求队列访问寄存器数据。 单个 INT WINAPI ExecuteQueuingAccess() ; 多个 INT WINAPI ExecuteQueuingAccessM(HANDLE hProServer);	
参数	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项 • 如果 Pro-Server EX 成功访问了所有指定的寄存器， ExecuteQueuingAccess() 将返回一个成功代码。反之，如果 Pro-Server EX 未能访问到任何寄存器， ExecuteQueuingAccess() 将返回一个访问错误代码。如需获知是否成功执行了各寄存器访问请求，可调用 sQueuingAcceessSucceeded() 来检查结果。 • 不能在排队访问中注册 ACTION。	

函数	丢弃寄存器读取 / 写入请求队列
丢弃寄存器读取 / 写入请求队列。 单个 INT WINAPI CancelQueuingAccess() ; 多个 INT WINAPI CancelQueuingAccessM(HANDLE hProServer);	
参数	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项 调用 BeginQueuingWrite() 或 BeginQueuingRead() 后， Pro-Server EX 开始对寄存器访问请求进行排队，直到调用 ExecuteQueuingAccess()。 如果因某种原因不再需要请求队列，请调用此 API。 Pro-Server EX 丢弃请求队列，退出排队。	

函数	检查寄存器读取 / 写入请求队列的运行结果

检查调用 ExecuteQueuingAccess() 后，是否成功执行了各寄存器访问请求。

单个
INT WINAPI IsQueuingAccessSucceeded(INT iIndex)；
多个
INT WINAPI IsQueuingAccessSucceededM(HANDLE hProServer,INT iIndex);

参数
iIndex（入）要检查的请求编号

调用 BeginQueuingWrite() 或 BeginQueuingRead() 后，会调用数次寄存器访问 API 对寄存器访问请求进行排队，直到调用 ExecuteQueuingAccess()。注意，在执行 ExecuteQueuingAccess() 之前，无法得知实际的寄存器访问结果。
如需获知各个寄存器访问请求的结果，请首先执行 ExecuteQueuingAccess()，然后指定对应于目标寄存器的请求编号 (从 0 开始)。

返回值
XX：错误代码
0：表示指定编号的寄存器访问请求已成功执行。

特别注意事项
(例如)

```
BeginQueuingWrite()；
        WriteDevice16(" 节点 1","LS100", 数据 ,10);
        WriteDevice16(" 节点 1","LS200", 数据 ,10);
        WriteDevice16(" 节点 1","LS300", 数据 ,10);
ExecuteQueuingAccess()
```

如需检查 “节点 1”访问 “LS200”是否成功执行，可使用 IsQueuingAccessSucceeded(1)。
如果返回值为 “0”，则表示成功执行了该访问。

27.5 系统 API

函数	创建 Pro-Server 句柄
使用多句柄函数时获取 Pro-Server 句柄。 HANDLE WINAPI CreateProServerHandle();	
参数	**返回值** 正常结束：非 0(句柄代码) 异常结束：0
特别注意事项	

函数	释放 Pro-Server 句柄
释放获取的 Pro-Server 句柄。 INT WINAPI DeleteProServerHandle(HANDLE hProServer);	
参数 hProServer：(入) 要释放的 Pro-Server 句柄	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

函数	载入网络工程文件
载入由参数指定的网络工程文件。 单个 INT WINAPI EasyLoadNetworkProject(LPCSTR sDBName,DWORD dwSetOrAdd = TRUE); 多个 INT WINAPI EasyLoadNetworkProjectM(HANDLE hProServer,LPCSTR sDBName,DWORD dwSetOrAdd = TRUE);	
参数 sDBName：指定要载入网络工程文件的完整路径。 dwSetOrAdd：保留 (固定为 “1”) hProServer：Pro-Server 句柄	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

函数	将错误代码转换为字符串

将 Pro-Server EX 各 API 返回的错误代码转换为错误消息。
EasyLoadErrorMessage() 返回多字节字符串 (ASCII) 消息。 EasyLoadErrorMessageW() 返回宽字符串 (UNICODE) 消息。

BOOL WINAPI EasyLoadErrorMessage(INT iErrorCode,LPSTR osErrorMessage)；
BOOL WINAPI EasyLoadErrorMessageW(INT iErrorCode,LPWSTR owsErrorMessage);

参数
iErrorCode（入）Pro-Server EX 函数返回的错误代码
osErrorMessage（出）指向转换的字符串（多字节字符串）的保存区。（调用此 API，需要至少 512 字节的保存区。）
osErrorMessage（出）指向转换的字符串（多字节字符串）的保存区。（调用此 API，需要至少 1024 字节的保存区。）

返回值
正常结束：非 0
字符串转换失败（例如：未定义代码）：0

特别注意事项
- 此 API 用于确保与旧版本 Pro-Server 的兼容性。
- 使用 EasyLoadErrorMessageEx()，可将错误代码转换为更为详细的错误消息。建议使用 EasyLoadErrorMessageEx()。

函数	将错误代码转换为字符串（带状态信息）

将 Pro-Server EX 各 API 返回的错误代码转换为错误消息。
Pro-Server EX 在返回错误消息的同时，如有可能，还返回错误发生状况和其他信息。
EasyLoadErrorMessage() 总是返回对应于指定错误代码的同一错误消息。而 EasyLoadErrorMessageEx() 则能返回更为详细的错误信息，包括通讯目标设备的名称、错误发生位置等，具体取决于错误发生时的状况。因此，对应于同一错误代码， EasyLoadErrorMessageEx() 可以根据实际情况返回不同的错误消息。
EasyLoadErrorMessageEx() 和 EasyLoadErrorMessageExM() 返回多字节字符串 (ASCII) 消息。
EasyLoadErrorMessageExW() 和 EasyLoadErrorMessageExWM() 返回宽字符串 (UNICODE) 消息。

单个
BOOL WINAPI EasyLoadErrorMessageEx(INT iErrorCode,LPSTR osErrorMessage)；
BOOL WINAPI EasyLoadErrorMessageExW(INT iErrorCode,LPWSTR owsErrorMessage)；
多个
BOOL WINAPI EasyLoadErrorMessageExM(HANDLE hProServer,INT iErrorCode,LPSTR osErrorMessage);
BOOL WINAPI EasyLoadErrorMessageExWM(HANDLE hProServer,INT iErrorCode,LPWSTR owsErrorMessage);

参数
iErrorCode（入）Pro-Server EX 函数返回的错误代码
osErrorMessage（出）指向转换的字符串（多字节字符串）的保存区。（调用此 API 需要至少 1024 字节的保存区。）
owsErrorMessage（出）指向转换的字符串（宽字符串）的保存区。（调用此 API，需要至少 2048 字节的保存区。）

返回值
正常结束：非 0
字符串转换失败（例如：未定义代码）：0

特别注意事项
- EasyLoadErrorMessage() 将错误代码转换为消息，用于调用 Pro-Server EX 的 API 后返回错误的情况。
- Pro-Server EX 每句柄只能保存一条错误状态消息。因此，如果在造成错误的 API 与 EasyLoadErrorMessage() 之间调用另一个 API， EasyLoadErrorMessage() 将不会返回错误状态信息，因为保存的错误状态消息被重写。所以，使用 EasyLoadErrorMessageM() 时，指定的 Pro-Server 句柄必须与调用相关 API 时使用的句柄相同。

函数	初始化 Pro-Server API
初始化一个 Pro-Server EX API，在内部声明 API 的使用。 如果在未启动 Pro-Server EX 的情况下执行 EasyInit()， Pro-Server EX 将自动启动。 INT WINAPI EasyInit();	
参数	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

函数	结束 Pro-Server API
INT WINAPI EasyTerm();	
参数	**返回值**
特别注意事项 此 API 用于确保与旧版本 Pro-Server 的兼容性。 使用 Pro-Server EX 时，不需要调用此 API。（即使调用，也不会执行。）	

函数	关闭 Pro-Server EX
关闭 Pro-Server EX。 调用此 API 后，请勿调用任何 Pro-Server EX 的 API。 调用此 API 前，请务必丢弃 Pro-Server 句柄。 INT WINAPI EasyTermServer();	
参数	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

函数	Pro-Server EX 关闭通知

此 API 可使用户获知 Pro-Server EX 的关闭状态。
Pro-Server EX 开始关闭处理时，通过使用 Windows API 的 PostMessage()，向用此 API 注册的窗口发送一条指定的消息。
有关 PostMessage() 的详情，请参阅 Windows API 手册。
应用程序收到来自该窗口的消息时，即得知 Pro-Server EX 将立刻关闭。

单个
INT WINAPI EasyNotifyFromServerEnd(HWND hReceivedWnd,UINT uMessage,WPARAM WParam = 0, LPARAM LParam = 0);
多个
INT WINAPI EasyNotifyFromServerEndM(HANDLE hProServer,HWND hReceivedWnd,UINT uMessage,WPARAM WParam = 0, LPARAM LParam = 0);

参数
hReceivedWnd：(入) 接收关闭消息的窗口。
uMessage：(入) 要发送的关闭消息的 ID。
准备关闭 Pro-Server EX 时，此 ID 将被发送到用 hReceivedWnd 指定的窗口。
WParam：(入) 准备与消息一起发送的 WPARAM(PostMessage() 中 WPARAM 的值)
Lparam：(入) 准备与消息一起发送的 LPARAM(PostMessage() 中 LPARAM 的值)

返回值
正常结束：0
异常结束：错误代码

特别注意事项
此 API 用于使应用程序与 Pro-Server EX 同时关闭。
例如，为此 API 的 hReceivedWnd 指定应用程序主窗口， uMessage 指定 WM_QUIT，调用此 API 后，当 Pro-Server EX 关闭时，将向应用程序主窗口发送 WM_QUIT。
通常，应用程序用 WM_QUIT 作为程序关闭信号。这样就能使应用程序与 Pro-Server EX 同时关闭。

函数	阻止消息处理

如果函数处理过程较长，大多数 Pro-Server EX API(函数) 会在函数处理过程中处理 Windows 消息。此 API 可指定执行或阻止 Windows 消息的处理。
如果阻止了 Windows 消息的处理，则将在消息队列中保存相关的 Windows 消息，在函数执行过程中不会处理它们。
这样，在函数执行过程中，就不需要再重复点击图标来调用函数。
不过，这样将阻止所有 Windows 消息和图标点击消息的处理，同时也无法处理定时器和窗口重绘等重要消息。
对每个 Pro-Server EX 句柄，都可指定是执行还是阻止 Windows 消息的处理。默认设置为 “执行”消息处理。

单个
INT EasySetWaitType(DWORD dwMode);
多个
INT EasySetWaitTypeM(HANDLE hProServer,DWORD dwMode);

参数
hProServerHandle：(入) 用于处理模式更改的 Pro-Server 句柄
dwMode：(入) 执行消息处理，请指定 “1”。
阻止消息处理，请指定 “2”。

返回值
正常结束：0
异常结束：错误代码

特别注意事项

函数	获取消息处理模式
获取调用 Pro-Server EX API 过程中当前消息的处理模式。 多句柄 API 为每个句柄返回当前消息处理模式。 单个 INT EasyGetWaitType(); 多个 INT EasyGetWaitTypeM(HANDLE hProServerHandle);	
参数 HANDLE hProServerHandle：（入）状态获取句柄	**返回值** 1: 执行消息处理。 2: 阻止消息处理。
特别注意事项	

函数	向日志查看器中添加日志

如果内部处理发生了特定事件 (Pro-Server EX 启动 / 关闭、错误等)， Pro-Server EX 可以记录此事件。
通过日志查看器可以查看记录下来的信息。 (请参阅 “28.5 监视系统事件日志”)
使用此 API， Pro-Server EX 可以记录特定消息。此 API 可用于应用程序调试。

INT WINAPI EasyOutputLog(BYTE bLevel,LPCSTR sPrompt,LPCSTR sMessage);

参数

bLevel：(入) 事件类型

记录所有消息会导致系统性能下降。为避免此类情况， Pro-Server EX 提供了过滤功能，可按事件类型记录消息。
请指定当前记录消息所属的事件类型。
事件类型列表如下。

定义	十六进制值	事件类型
EASY_LogLevel_SysMessage	0x01	系统消息
EASY_LogLevel_SysError	0x02	系统错误消息
EASY_LogLevel_AppError	0x04	用户程序错误消息
EASY_LogLevel_AppStart	0x08	用户程序启动消息
EASY_LogLevel_AppEnd	0x10	用户程序关闭消息
EASY_LogLevel_AppWarning	0x20	用户程序警告消息
EASY_LogLevel_AppMessage1	0x40	用户程序详细消息 1
EASY_LogLevel_AppMessage2	0x80	用户程序详细消息 2

sPrompt：(入) 指示事件发生位置的字符串 (以 NULL 终止)
sMessage：(入) 要记录的字符串消息 (以 NULL 终止)

实际记录的消息是两个字符串 (sPrompt 和 sMessage) 的简单组合。

返回值

正常结束：0
异常结束：错误代码

特别注意事项

函数	清除日志查看器中的日志
清除 EasyOutputLog() 记录的信息。 此 API 可用于应用程序调试。 INT WINAPI EasyOutputLogClear();	
参数 HANDLE hProServerHandle：(入) 状态获取句柄	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

27.6 SRAM 数据访问 API

函数	读取 SRAM 备份数据

读取保存在 GP 系列节点 SRAM 中的数据，并将数据以文件形式保存在 PC 上。
配方数据存为二进制格式，其他类型的数据存为 CSV 格式。

INT WINAPI EasyBackupDataRead(LPCSTR sSaveFileName,LPCSTR sNodeName,INT iBackupDataType,INT iSaveMode);

参数

sSaveFileName：(入) 保存读取数据的文件路径。 (字符串指针)
sNodeName：(入) 读取数据源节点的名称 (字符串指针)
不能指定 Pro-Server EX 节点。
iSaveMode：(入) 保存模式
0：新建 (如果已经存在一个具有相同文件名的文件， Pro-Server EX 删除并重写该文件。)
1: 添加 (将读取到的数据添加到现有文件的末尾。如果没有保存数据的文件， Pro-Server EX 将新建一个文件。)
其他：保留
iBackupDataType：(入) 要读取数据的数据类型

值	数据源节点 (GP 系列)	数据源节点 (非 GP 系列)
0x0001	配方数据	配方数据
0x0002	日志数据	采样组 1 的采样数据
0x0003	曲线图数据	除采样组 1 以外的所有采样组数据
0x0004	采样数据	
0x0005	报警块 1	报警块 1
0x0006	历史报警或报警块 2	报警块 2
0x0007	报警日志或报警块 3	报警块 3
0x0008	报警块 4	报警块 4
0x0009	报警块 5	报警块 5
0x000A	报警块 6	报警块 6
0x000B	报警块 7	报警块 7
0x000C	报警块 8	报警块 8
其他	(保留)	(保留)

如果数据源节点为 GP4000 系列 /GP3000 系列 /WinGP/LT3000，数据类型为报警块 1~8，根据 GP-Pro EX 的设置，一个报警块最多可以保存三种类型的数据 (活动数据、历史数据和日志数据)。但是，此 API 根据以下顺序检查报警块是否包含有效数据，如果有，则读取这些数据。
(1) 历史报警
(2) 报警日志
(3) 活动报警
如果没有有效数据，将发生错误。

返回值
正常结束：0
异常结束：错误代码

特别注意事项

函数	读取扩展 SRAM 备份数据

读取保存在 GP 系列节点 SRAM 中的数据，并将数据以文件形式保存在 PC 上。
配方数据存为二进制格式，其他类型的数据存为 CSV 格式。
与 EasyBackupDataRead() 不同，使用此 API 可访问 GP4000 系列、 GP3000 系列、 WinGP 和 LT3000 的扩展数据。

```
INT WINAPI EasyBackupDataReadEx(LPCSTR sSaveFileName, LPCSTR sNodeName, INT
iBackupDataType, INT iSaveMode, INT iNumber = 0 , INT iStringTable = 0x0000);
```

参数

sSaveFileName：(入) 保存读取数据的文件路径。 (字符串指针)

sNodeName：(入) 读取数据源节点的名称 (字符串指针)

不能指定 Pro-Server EX 节点。

iSaveMode：(入) 保存模式

0: 新建 (如果已经存在一个具有相同文件名的文件， Pro-Server EX 删除并重写该文件。)

1: 添加 (将读取到的数据添加到现有文件的末尾。如果没有保存数据的文件，Pro-Server EX 将新建一个文件。)

其他：保留

iBackupDataType：(入) 要读取数据的数据类型

值	数据源节点 (GP 系列)	数据源节点 (非 GP 系列)
0x0001	配方数据	配方数据
0x0002	日志数据	采样组 1 的采样数据
0x0003	曲线图数据	除采样组 1 以外的所有采样组数据
0x0004	采样数据	
0x0005	报警块 1	报警块 1 指定报警类型的 iNumber。
0x0006	历史报警或报警块 2	报警块 2 指定报警类型的 iNumber。
0x0007	报警日志或报警块 3	报警块 3 指定报警类型的 iNumber。
0x0008	报警块 4	报警块 4 指定报警类型的 iNumber。
0x0009	报警块 5	报警块 5 指定报警类型的 iNumber。
0x000A	报警块 6	报警块 6 指定报警类型的 iNumber。
0x000B	报警块 7	报警块 7 指定报警类型的 iNumber。
0x000C	报警块 8	报警块 8 指定报警类型的 iNumber。
0x8002	(保留)	特定组编号的采样组 指定组编号的 iNumber。

返回值

正常结束：0

异常结束：错误代码

iNumber：（入）当 sSaveFileName 指定 GP 系列文件时，此参数被忽略。
另外，此参数的含义因 iBackupDataType 的值而不同。

iBackupDataType 的值	描述
0x0005 至 0x000C	有三种类型的报警数据 (活动、历史和日志)。指定一个目标报警类型。 iNumber 的值 / 描述 0：Pro-Server EX 根据以下顺序检查报警块是否包含有效数据，如果有，则读取这些数据。 (1) 历史报警 (2) 报警日志 (3) 活动报警 如果没有有效数据，将发生错误。 1：读取活动报警数据。 2：读取历史报警数据。 3：读取报警日志数据。 如果目标数据类型不在 iBackupDataType 指定的报警块中，就会发生错误。
0x8002	要读取的采样组的组编号 1 ~ 64 中的任意值
其他	(保留)

iStringTable：（入）保留
总是指定 “0”。

函数	写入 SRAM 备份数据
以二进制格式将指定的配方数据写入 GP 系列节点的 SRAM。 INT WINAPI EasyBackupDataWrite(LPCSTR sSourceFileName,LPCSTR sNodeName,INT iBackupDataType);	
参数 sSourceFileName：(入) 要写入的二进制配方数据文件的路径 (字符串指针) sNodeName：(入) 数据写入目标入口节点的名称 (字符串指针) 不能指定 Pro-Server EX 节点、 GP3000 系列节点、 GP4000 系列节点、WinGP 节点或 LT3000 节点。 BackupDataType：(入) 固定为 “1”。 (“1” 表示配方数据。)	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

27.7 CF 卡 /SD 卡 API

注 释

- 是用于访问 CF 卡和 SD 卡数据的 API。不能在没有 CF 卡或 SD 卡插槽的机型上使用此 API。
- 使用带 SD 卡插槽的机型时，请用 “SD” 和 “SD 卡” 替代 “CF” 和 “CF 卡”。
- 可使用 CF 卡 API 函数读取和写入 SD 卡。
  同样，也可用 SD 卡 API 函数读取和写入 CF 卡。

函数	读取 CF 卡状态

获取所连接 GP 中 CF 卡的连接状态。

单个
CF 卡：INT WINAPI EasyIsCFCard(LPCSTR sNodeName) ;
SD 卡：INT WINAPI EasyIsSDCard(LPCSTR sNodeName);
多个
CF 卡：INT WINAPI EasyIsCFCardM(HANDLE hProServer,LPCSTR sNodeName) ;
SD 卡：INT WINAPI EasyIsSDCardM(HANDLE hProServer,LPCSTR sNodeName);

参数
hProServer：Pro-Server 句柄
sNodeName：状态读取目标 GP 节点的名称 (必须在网络工程中预先注册该节点名称。)

返回值

函数返回值	GP 系列节点	非 GP 系列节点
0x00000000	正常	正常
0x10000001	无 CF 卡	无 CF 卡，或 CF 卡插槽盖处于打开状态 (无论是否插入了 CF 卡)
0x10000002	检测到与 CF 卡驱动程序不兼容	
0x10000004	检测到 CF 卡错误	检测到 CF 卡错误
0x10000008	未初始化 CF 卡	
其他	与 CF 卡无关的错误	

特别注意事项

函数	读取 CF 卡文件列表 (任意文件夹名称)

从 GP 节点内插入的 CF 卡中输出文件列表到一个用参数指定的文件。可指定任意文件保存该文件列表。

CF 卡：INT WINAPI EasyGetListInCfCard(LPCSTR sNodeName,LPCSTR sDirectory, INT* oiCount,LPCSTR sSaveFileName) ;
SD 卡：INT WINAPI EasyGetListInSdCard(LPCSTR sNodeName, LPCSTR sDirectory, INT* oiCount, LPCSTR sSaveFileName);

参数

sNodeName：输出文件列表的 GP 节点名称
sDirectory：接收文件列表的文件夹名称 (全部大写)
oiCount：输出文件数
sSaveFileName：保存输出目录信息的文件名 指定的文件保存 stEasyDirInfo 指定的二进制对齐类型数据，数量由 pioCount 的返回值指定。所有文件名和扩展名均为大写。

```
struct stEasyDirInfo {
BYTE bFileName[8+1];// 文件名 ( 以 “0” 结尾 )
BYTE bExt[3+1];// 文件扩展名 ( 以 “0” 结尾 )
BYTE bDummy[3];// 虚拟
DWORD dwFileSize;// 文件大小
BYTE bFileTimeStamp[8+1];// 文件时间戳 ( 以 “0” 结尾 )
BYTE bDummy2[3];// 虚拟 2
} ;
```

返回值

正常结束：0
异常结束：错误代码

特别注意事项

作为 bFileTimeStamp(8 字节) 的补充，高位 4 字节以 MS-DOS 格式表示时间，低位 4 字节以 MS-DOS 格式表示日期 (十六进制字符串)。
MS-DOS 时间 / 日期格式如下：
(例如：20C42C22 表示 2002/1/2 4:6:8。“2C22”是日期的十六进制符号，“20C4”是时间的十六进制符号。)

位	描述
0~4	日 (1~31)
5~8	月 (1 = 1 月， 2 = 2 月， 12 = 12 月)
9~15	年：表示从 1980 年开始经过的年数，实际年份是 1980 与这些位的值加起来的和。

指定 MS-DOS 格式的时间。时间占用 16 个位，格式如下：

位	描述
0~4	秒除以 2 的值 (0~29)
5~10	分 (0~59)
11~15	时 (0~23， 24 小时制)

从 GP4000 系列、 GP3000 系列、 WinGP 或 GP 系列节点读取文件列表时，短于 8 个字符的文件名和短于 3 个字符的文件扩展名分别显示为 bFileName[8+1] 和 bExt[3+1]，如下所示。

读取源节点	非 GP 系列节点	GP 系列节点
bFileName[8+1]	文件名短于 8 个字符时，在原始文件名末尾保存 Null(0x00)，在 Null 之后保存未定义的值。	文件名短于 8 个字符时，在原始文件名后保存单字节空格 (0x20)，以 Null(0x00) 作为最后一个字符。
bExt[3+1]	文件扩展名短于 3 个字符时，在原始文件扩展名末尾保存 Null(0x00)，在 Null 之后保存未定义的值。	文件扩展名短于 3 个字符时，在原始文件扩展名后保存单字节空格 (0x20)，以 Null(0x00) 作为最后一个字符。

(例如) 文件名和文件扩展名为 ABC.D

非 GP 系列节点

bFileName[8+1]	0x410x420x430x00********** (**** 表示一个未定义的值)
bExt[3+1]	0x440x00******** (**** 表示一个未定义的值)

GP 系列节点

bFileName[8+1]	0x410x420x430x200x200x200x200x200x00
bExt[3+1]	0x440x200x200x00

函数	读取 CF 卡文件列表 (类型指定)

从 GP 节点内插入的 CF 卡中输出文件列表到一个用参数指定的文件。只能输出 sDirectory 所指定目录中的文件列表。

INT WINAPI EasyGetListInCard(LPCSTR sNodeName,LPCSTR sDirectory, INT* oiCount,LPCSTR sSaveFileName) ;

参数

sNodeName：输出文件列表的 GP 节点名称

sDirector：输出列表的目录名称 (全部大写)。此 API 仅支持以下目录：

LOG(日志数据)
TREND(趋势图数据)
ALARM(报警数据)
CAPTURE(画面捕捉数据)
FILE(配方数据)

oiCount：输出文件数

sSaveFileName：保存输出目录信息的文件名 指定的文件保存 stEasyDirInfo 指定的二进制对齐类型数据，数量由 pioCount 的返回值指定。所有文件名和扩展名均为大写。

```
struct stEasyDirInfo {
BYTE bFileName[8+1];// 文件名 ( 以 “0” 结尾 )
BYTE bExt[3+1];// 文件扩展名 ( 以 “0” 结尾 )
BYTE bDummy[3];// 虚拟
DWORD dwFileSize;// 文件大小
BYTE bFileTimeStamp[8+1];// 文件时间戳 ( 以 “0” 结尾 )
BYTE bDummy2[3];// 虚拟 2
} ;
```

返回值

正常结束：0
异常结束：错误代码

特别注意事项

从 GP4000 系列、 GP3000 系列、 WinGP 或 GP 系列节点读取文件列表时，短于 8 个字符的文件名和短于 3 个字符的文件扩展名分别显示为 bFileName[8+1] 和 bExt[3+1]，如下所示。

读取源节点	非 GP 系列节点	GP 系列节点
bFileName[8+1]	文件名短于 8 个字符时，在原始文件名末尾保存 Null(0x00)，在 Null 之后保存未定义的值。	文件名短于 8 个字符时，在原始文件名后保存单字节空格 (0x20)，以 Null(0x00) 作为最后一个字符。
bExt[3+1]	文件扩展名短于 3 个字符时，在原始文件扩展名末尾保存 Null(0x00)，在 Null 之后保存未定义的值。	文件扩展名短于 3 个字符时，在原始文件扩展名后保存单字节空格 (0x20)，以 Null(0x00) 作为最后一个字符。

(例如) 文件名和文件扩展名为 ABC.D

非 GP 系列节点

bFileName[8+1]	0x410x420x430x00********** (**** 表示一个未定义的值)
bExt[3+1]	0x440x00******** (**** 表示一个未定义的值)

GP 系列节点

bFileName[8+1]	0x410x420x430x200x200x200x200x200x00
bExt[3+1]	0x440x200x200x00

函数	读取 CF 卡文件 (任意文件名指定)

从 CF 卡中读取指定文件。可指定任意文件进行读取 。 CF 卡：INT WINAPI EasyFileReadInCfCard(LPCSTR sNodeName,LPCSTR sFolderName,LPCSTR sFileName, LPCSTR pWriteFileName,DWORD* odwFileSize) ; SD 卡：INT WINAPI EasyFileReadInSdCard(LPCSTR sNodeName, LPCSTR sFolderName, LPCSTR sFileName, LPCSTR pWriteFileName, DWORD* odwFileSize);	
参数 sNodeName：输出文件列表的 GP 节点名称 sFolderName：包含从 CF 卡中读取的源文件的文件夹名称 (最多 32 个单字节字符) sFileName：从 CF 卡中读取的源文件的名称 (最多为 8.3 格式的字符串) pWriteFileName odwFileSize：要读取的 CF 卡文件的大小	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

函数	读取 CF 卡文件 (类型指定)

从 CF 卡中读取指定文件。只能读取由 pReadFileType 指定的文件类型。

INT WINAPI EasyFileReadCard(LPCSTR sNodeName,LPCSTR pReadFileType,WORD wReadFileNo,LPCSTR sWriteFileName,DWORD* odwFileSize) ;

参数
sNodeName：输出文件列表的 GP 节点名称
pReadFileType：要从 CF 卡中读取的源文件的类型 (请参阅 < 特别注意事项 >)
wReadFileNo：要从 CF 卡中读取的源文件的编号
sWriteFileName

odwFileSize：要读取的 CF 卡文件的大小

返回值
正常结束：0
异常结束：错误代码

特别注意事项
此 API 支持以下文件类型。只能读取保存在指定 CF 卡文件夹中的文件。

■GP 系列节点支持的文件类型

数据类型	文件类型	目标文件夹
配方数据	ZF	FILE
CSV 数据	ZR	FILE
图像画面	ZI	DATA
音频数据	ZO	DATA
曲线图数据	ZT	TREND
采样	ZS	TREND
报警 4~8	Z4 至 Z8	ARAM
日志数据	ZL	LOG
报警日志	ZG	ALARM
历史报警	ZH	ALARM
活动报警	ZA	ALARM
画面数据备份	ZC	MRM
画面捕捉	CP	CAPTURE

■GP4000 系列节点、 GP3000 系列节点和 WinGP 节点支持的文件类型

数据类型	文件类型	目标文件夹
配方数据	ZF 或 F	FILE
CSV 数据	ZR	FILE
图像画面	ZI 或 I	DATA
音频数据	ZO 或 O	DATA
专用于 GP-Pro EX 的曲线图数据 (为兼容性)	ZT	TREND
专用于 GP-Pro EX 的采样数据 (为兼容性)	ZS	TREND
报警 1	Z1 或 ZA	ALARM
报警 2	Z2 或 ZH	ALARM
报警 3	Z3 或 ZG	ALARM
报警 4~8	Z4 至 Z8	ALARM
专用于 GP-Pro EX 的日志数据 (为兼容性)	ZL	LOG
捕捉数据	CP	CAPTURE
采样 1~64	ZS1 至 ZS64	SAMP01 至 SAMP64

函数	将文件写入 CF 卡 (任意文件名指定)

将指定文件写入 CF 卡。可指定任意文件进行写入 。

CF 卡：INT WINAPI EasyFileWriteInCfCard(LPCSTR sNodeName, LPCSTR pReadFileName, LPCSTR sFolderName, LPCSTR sFileName);
SD 卡：INT WINAPI EasyFileWriteInSdCard(LPCSTR sNodeName, LPCSTR pReadFileName, LPCSTR sFolderName, LPCSTR sFileName);

参数
sNodeName：写入文件目标 GP 节点的名称
pReadFileName：要写入 CF 卡的源文件的名称 (完整路径)
sFolderName：CF 卡中包含目标文件的文件夹名称 (最多 32 个单字节字符)
sFileName：CF 卡中目标文件的名称 (最多为 8.3 格式的字符串)

返回值
正常结束：0
异常结束：错误代码

特别注意事项

函数	将文件写入 CF 卡 (类型指定)
将指定文件写入 CF 卡。只能写入由 pWriteFileType 指定的文件类型。 INT WINAPI EasyFileWriteCard(LPCSTR sNodeName,LPCSTR pReadFileName,LPCSTR sWriteFileType,WORD wWriteFileNo) ;	
参数 sNodeName：写入文件目标 GP 节点的名称 pReadFileName：要写入 CF 卡的源文件的名称 (完整路径) sWriteFileType：CF 卡中目标文件的类型 (请参阅 “读取 CF 卡文件 (类型指定)” 功能的 < 特别注意事项 >) wWriteFileNo：CF 卡中目标文件的文件编号	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

函数	删除 CF 卡文件 (任意文件)
从 CF 卡中删除指定文件。可指定任意文件进行删除 。 CF 卡：INT WINAPI EasyFileDeleteInCfCard(LPCSTR sNodeName, LPCSTR sFolderName, LPCSTR sFileName) ; SD 卡：INT WINAPI EasyFileDeleteInSdCard(LPCSTR sNodeName, LPCSTR sFolderName, LPCSTR sFileName);	
参数 sNodeName：包含拟删除文件的 GP 节点名称 sFolderName：包含拟删除 CF 卡文件的文件夹名称 (最多 32 个单字节字符) sFileName：拟删除 CF 卡文件的名称 (最多为 8.3 格式的字符串)	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

函数	删除 CF 卡文件 (类型指定)

从 CF 卡中删除指定文件。只能删除由 pDeleteFileType 指定的文件类型。

INT WINAPI EasyFileDeleteCard(LPCSTR sNodeName, LPCSTR pDeleteFileType, WORD wDeleteFileNo);

参数
sNodeName：包含拟删除文件的 GP 节点名称
pDeleteFileType：拟删除 CF 卡文件的类型 (请参阅 < 特别注意事项 >)
wDeleteFileNo：拟删除 CF 卡文件的编号

返回值
正常结束：0
异常结束：错误代码

特别注意事项
若对 CF 卡中不存在的文件执行此函数，不会视为错误，处理会正常结束。
此 API 支持以下文件类型。只能读取保存在指定 CF 卡文件夹中的文件。

■GP 系列节点支持的文件类型

数据类型	文件类型	目标文件夹
配方数据	ZF	FILE
CSV 数据	ZR	FILE
图像画面	ZI	DATA
音频数据	ZO	DATA
曲线图数据	ZT	TREND
采样	ZS	TREND
报警 4~8	Z4 至 Z8	ARAM
日志数据	ZL	LOG
报警日志	ZG	ALARM
历史报警	ZH	ALARM
活动报警	ZA	ALARM
画面数据备份	ZC	MRM
画面捕捉	CP	CAPTURE

■GP4000 系列节点、 GP3000 系列节点和 WinGP 节点支持的文件类型

数据类型	文件类型	目标文件夹
配方数据	ZF 或 F	FILE
CSV 数据	ZR	FILE
图像画面	ZI 或 I	DATA
音频数据	ZO 或 O	DATA
专用于 GP-Pro EX 的曲线图数据 (为兼容性)	ZT	TREND
专用于 GP-Pro EX 的采样数据 (为兼容性)	ZS	TREND
报警 1	Z1 或 ZA	ALARM
报警 2	Z2 或 ZH	ALARM
报警 3	Z3 或 ZG	ALARM
报警 4~8	Z4 至 Z8	ALARM
专用于 GP-Pro EX 的日志数据 (为兼容性)	ZL	LOG
捕捉数据	CP	CAPTURE
采样 1~64	ZS1 至 ZS64	SAMP01 至 SAMP64

函数	重命名 CF 卡文件

重命名 CF 卡中的指定文件。

CF 卡：INT WINAPI EasyFileRenameInCfCard(LPCSTR sNodeName, LPCSTR sFolderName, LPCSTR sFileName,LPCSTR sFileRename);
SD 卡：INT WINAPI EasyFileRenameInSdCard(LPCSTR sNodeName, LPCSTR sFolderName, LPCSTR sFileName,LPCSTR sFileRename);

参数
sNodeName：写入文件目标 GP 节点的名称
sFolderName：包含拟重命名 CF 卡文件的文件夹名称 (最多 32 个单字节字符)
sFileName：拟重命名 CF 卡文件的名称 (最多为 8.3 格式的字符串)
sFileRename：新文件名 (最多为 8.3 格式的字符串)

返回值
正常结束：0
异常结束：错误代码

特别注意事项

函数	获取 CF 卡可用容量信息

获取连接到指定入口节点的 CF 卡的可用容量信息。

CF 卡：INT WINAPI EasyGetCfFreeSpace(LPCSTR sNodeName,INT* oiUnallocated);
CF 卡：INT WINAPI EasyGetCfFreeSpaceEx(LPCTSTR sNodeName,INT* pioUnallocatedL,INT* pioUnallocatedH);
SD 卡：INT WINAPI EasyGetSdFreeSpace(LPCSTR sNodeName,INT* oiUnallocated);
SD 卡：INT WINAPI EasyGetSdFreeSpaceEx(LPCTSTR sNodeName,INT* pioUnallocatedL,INT* pioUnallocatedH);

参数	返回值
sNodeName：输出文件列表的 GP 节点名称 oiUnallocated (*1)：CF 卡可用容量 (字节数) pioUnallocatedL：(出) 低 4 字节中的可用容量 pioUnallocatedH：(出) 高 4 字节中的可用容量	正常结束：0 异常结束：错误代码

特别注意事项
* 如果可用容量超过 INT 的允许范围，请使用 CF 卡 (扩展) 或 SD 卡 (扩展) 函数。

函数	FTP 被动模式设置

Pro-Server EX 使用一种特殊的协议访问 GP 系列节点中的 CF 卡。但是，访问 GP4000 系列节点、 GP3000 系列节点和 WinGP 节点时，使用的是 FTP 协议。
对于 FTP 协议， Pro-Server EX 支持两种模式：普通模式和被动模式。
此 API 指定 FTP 协议的模式。

INT WINAPI EasyFileSetPassiveMode(INT iPassive) ;

参数	返回值
iPassive：(入) 0：普通模式 非 0：被动模式 初始化 ProEasy 时， FTP 协议被设置为 “普通模式”。	正常结束：0 异常结束：错误代码

特别注意事项

27.8 二进制日期和时间 / 文本显示转换

■ 二进制值到文本转换 API

函数	二进制值到文本转换 (时间类型)

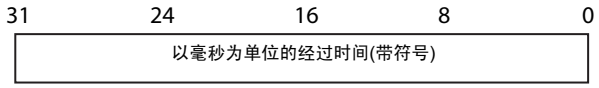
二进制值到 TIME 型字符串转换函数。

INT WINAPI EasyTIMEToString(DWORD dwData, LPSTR osTime);

参数	**返回值**
dwData：(入) 转换前的二进制值 osTime：(出) 转换后的文本字符串 *1	正常结束：0 异常结束：错误代码

特别注意事项

输入格式



输出格式

%s%02ud%02uh%02um%02us%03ums (符号 , 日 , 时 , 分 , 秒 , 毫秒)

输出示例

(1) 01d02h03m04s005ms

(2) -02d03h04m05s006ms

函数	二进制值到文本转换 (TIME_OF_DAY 型)

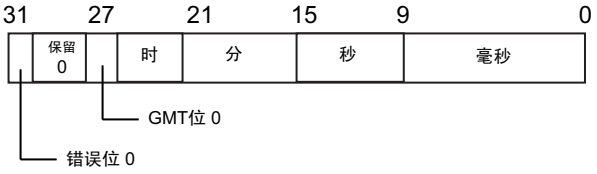
二进制值到 TIME_OF_DAY 型字符串转换函数。

INT WINAPI EasyTIME_OF_DAYToString(DWORD dwData, LPSTR osTod);

参数	**返回值**
dwData：(入) 转换前的二进制值 osTod：(出) 转换后的文本字符串 *1	正常结束：0 异常结束：错误代码

特别注意事项

输入格式



输出格式

%02u:%02u:%02u.%03u (时 , 分 , 秒 , 毫秒)

输出示例

23:59:59.999

函数	二进制值到文本转换 (DATE 型)

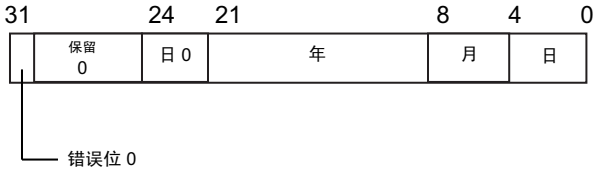
二进制值到 DATE 型字符串转换函数。

INT WINAPI EasyDATEToString(DWORD dwData, LPSTR osDate);

参数	**返回值**
dwData：（入）转换前的二进制值 osDate：（出）转换后的文本字符串 *1	正常结束：0 异常结束：错误代码

特别注意事项

输入格式



输出格式

%04u-%02d-%02u (年 , 月 , 日)

输出示例

2012-01-01

函数	二进制值到文本转换 (DATE_AND_TIME 型)

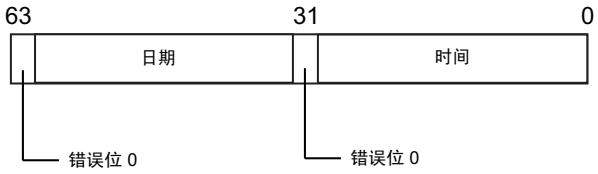
二进制值到 DATE_AND_TIME 型字符串转换函数。

INT WINAPI EasyDATE_AND_TIMEToString(QWORD qwData, LPSTR osDt);

参数	**返回值**
dwData：（入）转换前的二进制值 osDt：（出）转换后的文本字符串 *1	正常结束：0 异常结束：错误代码

特别注意事项

输入格式



输出格式

%04u-%02u-%02u-%02u:%02u:%02u.%03u (年 , 月 , 日 , 时 , 分 , 秒 , 毫秒)

输出示例

2012-01-02-03:04:05.006

*1 请确保此区大小在 32 字节以上。

*2 关于各寄存器访问 API 的更多详情，请参阅 27.2 寄存器访问 API。

■ 文本到二进制值转换 API

函数	INT WINAPI EasyStringToTIME()

TIME 型字符串到二进制值转换函数。

INT WINAPI EasyStringToTIME(LPCSTR sTime, DWORD *pdwData);

参数
sTime（入）转换前的文本字符串
pdwData（出）转换后的二进制值

返回值
正常结束：0
异常结束：错误代码

特别注意事项
输入格式
%s%02ud%02uh%02um%02us%03ums（符号，日，时，分，秒，毫秒）

	日	时	分	秒	毫秒
设置范围	-24...24	0...23	0...59	0...59	0...999
单位（分隔符）	d	h	m	s	ms

- 根据输入格式在设置范围内输入所有项目。
- 设置各项目时，须使各项转换为毫秒的值在 -2,147,483,648~2,147,483,647 范围之内。

输入示例
01d02h03m04s005ms

函数	INT WINAPI EasyStringToTIME_OF_DAY()

TIME_OF_DAY 型字符串到二进制值转换函数。

INT WINAPI EasyStringToTIME_OF_DAY(LPCSTR sTod, DWORD *pdwData);

参数
sTod（入）转换前的文本字符串
pdwData（出）转换后的二进制值

返回值
正常结束：0
异常结束：错误代码

特别注意事项
输入格式

%02u:%02u:%02u.%03u（时，分，秒，毫秒）

	时	分	秒	毫秒
设置范围	0...23	0...59	0...59	0...999
单位（分隔符）	:	:	.	

- 根据输入格式在设置范围内输入所有项目。

输入示例
23:59:59.999

函数	INT WINAPI EasyStringToDATE()

DATE 型字符串到二进制值转换函数。

INT WINAPI EasyStringToDATE(LPCSTR sDate, DWORD *pdwData);

参数
sDate （入）转换前的文本字符串
pdwData （出）转换后的二进制值

返回值
正常结束：0
异常结束：错误代码

特别注意事项
输入格式
%04u-%02d-%02u（年，月，日）

	年	月	日
设置范围	1970...8191	1...12	1...31
单位（分隔符）	-	-	

- 根据输入格式在设置范围内输入所有项目。

输入示例
2012-01-01

函数	INT WINAPI EasyStringToDATE_AND_TIME()

DATE_AND_TIME 型字符串到二进制值转换函数。

INT WINAPI EasyStringToDATE_AND_TIME(LPCSTR sDt, QWORD *pqwData);

参数
sDt （入）转换前的文本字符串
pdwData （出）转换后的二进制值

返回值
正常结束：0
异常结束：错误代码

特别注意事项
输入格式
%04u-%02u-%02u-%02u:%02u:%02u.%03u（年，月，日，时，分，秒，毫秒）

	年	月	日	时	分	秒	毫秒
设置范围	1970...8191	1...12	-24...24	0...23	0...59	0...59	0...999
单位（分隔符）	-	-	-	:	:	.	

- 根据输入格式在设置范围内输入所有项目。

输入示例
2012-03-21-01:02:03.004

*1 关于各寄存器访问 API 的更多详情，请参阅 27.2 寄存器访问 API。

27.9 其他 API

函数	读取 GP 时间 (DWORD 型)
获取指定节点的当前时间，用 DWORD 值表示。此函数仅对保存在自 LS2048 始的 6 个字中的时间有效。 DWORD WINAPI EasyGetGPTime(LPCSTR sNodeName,DWORD* odwTime) ;	
参数 sNodeName：目标节点的名称 (不能指定 Pro-Server EX 节点。) odwTime：获取的时间 (为 DWORD 型数值 (实质上是 ANSI 定义的 time_t 类型)。)	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

函数	读取 GP 时间 (VARIANT 型)
获取指定节点的当前时间，用 Variant 值表示。此函数仅对保存在自 LS2048 始的 6 个字中的时间有效。 DWORD WINAPI EasyGetGPTimeVariant(LPCSTR sNodeName,LPVARIANT ovTime) ;	
参数 sNodeName：目标节点的名称 (不能指定 Pro-Server EX 节点。) ovTime：获取的时间 (为 VARIANT 型数值。内部处理格式为 “ 日期 ”。)	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项	

函数	读取 GP 时间 (STRING 型)
获取指定节点的当前时间，用 LPTSTR 字符串表示。此函数仅对保存在自 LS2048 始的 6 个字中的时间有效。 DWORD WINAPI EasyGetGPTimeString (LPCSTR sNodeName,LPCSTR sFormat,LPSTR osTime) ;	
参数 sNodeName：目标节点的名称 (不能指定 Pro-Server EX 节点。) pFormat：指定拟获取时间的格式。百分号 (%) 后面格式指定代码的具体变化情况如 < 特别注意事项 > 中所示。 其他字符没有变化。 osTime：获取的字符串时间 (如果存储区小于获取的字符串长度的 + 1(NULL)，将发生未知的内存数据损坏。为此，请留出预计字符串长度 + 1(NULL) 的存储区。否则，将无法保证正常运行。)	**返回值** 正常结束：0 异常结束：错误代码

特别注意事项
百分号 (%) 后面格式指定代码的具体变化情况如下表所列。其他字符没有变化。例如，指定 “%Y_%M %S”，实际时间 “2006/1/2 12:34:56”将表示为字符串 “2006_34 56”。

格式指定代码	文件夹
%a	星期几的缩写 (*2)
%A	星期几的全名 (*2)
%b	月的缩写 (*2)
%B	月的全名 (*2)
%c	当地日期和时间
%#c	当地长日期和时间
%d	十进制日 (01~31)(*1)
%H	24 小时制时 (00~23)(*1)
%I	12 小时制时 (01~12)(*1)
%j	十进制日 (001~366)(*1)
%m	十进制月 (01~12)(*1)
%M	十进制分 (00~59)(*1)
%p	当地时区的 AM/PM(*2)
%S	十进制秒 (00~59)(*1)
%U	十进制周序号。周日作为每周的第一天。 (00~53)(*1)
%w	十进制星期。周日为 “0”。 (0~6)(*1)
%W	十进制周序号。周一作为每周的第一天。 (00~53)(*1)
%x	当地日期
%#x	当地长日期
%X	当地时间 (*2)
%y	十进制公历年的后两位 (00~99) (*1)
%Y	十进制 4 位公历年 (*1)
%z、 %Z	时区的全称或缩写。如果不知道时区，请空白。 (*2)
%%	百分号 (*2)

* 1: 若在 d、H、I、j、m、M、S、U、w、W、y 或 Y 前加 “#” 号 (如：%#d)，前面的 “0” 将被删除。
(例如：“05” 表示为 “5”。)

* 2: 若在 a、 A、 b、 B、 p、 X、 z、 Z 或 % 前加 “#” 号 (如：%#a)，“#” 将被忽略。

函数	读取 GP 时间 (STRING VARIANT 型)

获取指定节点的当前时间，用 Variant 型字符串表示。此函数仅对保存在自 LS2048 始的 6 个字中的时间有效。

DWORD WINAPI EasyGetGPTimeStringVariant(LPCSTR sNodeName,LPCSTR sFormat,LPVARIANT ovTime) ;

参数	返回值
sNodeName：目标节点的名称 (不能指定 Pro-Server EX 节点。) pFormat：指定拟获取时间的格式。百分号 (%) 后面格式指定代码的具体变化情况如下表所列。其他字符没有变化。 (详情请参阅 “读取 GP 时间 (STRING 型)”的 < 特别注意事项 >。) ovTime：获取的字符串时间 (为 VARIANT 型。内部处理格式为 “BSTR”。)	正常结束：0 异常结束：错误代码

特别注意事项

函数	读取入口节点状态

获取所连接 GP 节点的状态。由于可更改响应超时值，可使用此功能检查连接状态。

单个
INT WINAPI GetNodeProperty(LPCSTR sNodeName,DWORD dwTimeLimit,LPSTR osGPType,LPSTR osSystemVersion,LPSTR osComVersion,LPSTR osECOMVersion);
多个
INT WINAPI GetNodePropertyM(HANDLE hProServer,LPCSTR sNodeName,DWORD dwTimeLimit,LPSTR osGPType,LPSTR osSystemVersion,LPSTR osComVersion,LPSTR osECOMVersion);

参数	返回值
hProServer：(入)Pro-Server 句柄 sNodeName：(入) 拟读取其状态的 GP 节点名称 dwTimeLimit：(入) 响应超时设定值 (指定 “0”表示使用默认值 3000 毫秒。) 设置范围在 1~2,147,483,647 之间 (单位：毫秒) API 将目标节点的状态信息返回到以下区域。 请为各项留出至少 32 字节的空间。 osGPType：(出)GP 型号代码 osSystemVersion：(出)GP 系统版本 osComVersion：(出)PLC 协议驱动程序版本 对于 Pro-Server EX 节点或 GP4000 系列节点 /GP3000 系列节点 /WinGP 节点 /LT3000 节点，此项空白。 osECOMVersion：(出)2way 驱动程序版本 对于 Pro-Server EX 节点或 GP4000 系列节点 /GP3000 系列节点 /WinGP 节点 /LT3000 节点，此项空白。	正常结束：0 异常结束：错误代码

特别注意事项

函数	获取符号 / 组字节大小
获取访问寄存器符号或组符号所需的字节总数。 INT WINAPI SizeOfSymbol(LPCSTR sNodeName,LPCSTR sSymbolName,INT* oiByteSize);	
参数 sNodeName：(入) 带控制器 /PLC 名称的入口节点名称 sSymbolName：(入) 目标寄存器或符号的名称 oiByteSize：(出) 获取的字节大小	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项 “sSymbolName”可指定为寄存器符号、非队列组、整个队列组或队列组中的一个元素。	

函数	获取组成员数量
获取组或符号表的成员数量 (符号和组成员的总数)。 INT WINAPI GetCountOfSymbolMember(LPCSTR sNodeName,LPCSTR sSymbolName,INT* oiCountOfMember) ;	
参数 sNodeName：(入) 带控制器 /PLC 名称的入口节点名称 sSymbolName：(入) 目标组符号或符号表的名称 oiCountOfMember：(出) 获取的成员数量	**返回值** 正常结束：0 异常结束：错误代码
特别注意事项 如果某个组符号存在于指定的组符号中，则成员数计为一个，即使在内部组符号中存在多个寄存器符号。	

函数	获取符号 / 组 / 符号表定义信息
获取定义信息 (数据类型、数据数量等。) INT WINAPI GetSymbolInformation(LPCSTR sNodeName,LPCSTR sSymbolName,INT iMaxCountOfSymbolMember,LPSTR osSymbolSheetName,SymbolInformation* oSymbolInformation,INT* oiGotCountOfSymbolMember);	
参数 sNodeName：(入) 带控制器 /PLC 名称的入口节点名称 sSymbolName：(入) 符号 / 组 / 符号表的名称 iMaxCountOfSymbolMember：(入) 指定值为预计信息的最大数量 + 1。 指定准备的 “oSymbolInformation”的数量。 osSymbolSheetName：(出)API 返回符号表的名称，此符号表中包含用 sSymbolName 指定的符号。请留出 66 字节以上的空间。 oSymbolInformation：(出)API 用队列结构返回获取的详细信息。 为 iMaxCountOfSymbolMember 所指定数量做准备。 oiGotCountOfSymbolMember：(出)API 返回已返回到 oSymbolInformation 的信 息的数量。	**返回值** 正常结束：0 异常结束：错误代码

特别注意事项

• SymbolInformation 的结构

```
struct SymbolInformation
{
   WORDm_wAppKind;// 数据类型，符号：1 ~ 20，组：0x8000
   WORDm_wDataCount; // 数据数量
   DWORDm_dwSizeOf; // 访问所需的缓冲区字节数
   char m_sSymbolName[64+1];// 符号或组的名称
   charm_bDummy1[3];// 保留
   charm_sDeviceAddress[256+1]; // 寄存器地址 ( 如果是组的话，请空白。 )
   charm_bDummy2[3];// 保留
};
```

获取的信息被返回到 SymbolInformation 所指定队列结构的 oSymbolInformation 中。首元素包含 sSymbolName 所指定符号、组或表的信息。
sSymbolName 表示组时，第二及之后的元素包含组成员信息。
sSymbolName 表示表时，这些元素包含整个表的信息。
sSymbolName 表示符号时，第二及之后的元素不包含任何信息。

如果目标符号是一个位偏移符号，请注意以下几点：

(1) 若将一个位偏移符号直接指定为信息源符号 (直接为 sSymbolName 指定一个位偏移符号)，则会将 “2” 作为访问位符号所需的字节数设置给 SymbolInformation 的 m_dwSizeOf，或 oSymbolInformation 的首元素。
这时，由于信息源是一个符号， oSymbolInformation 没有第二或后续元素。

(2) 若将一个组符号指定为信息源符号，且指定的组包含一个位偏移符号，则会将 “0” 设置给 m_dwSizeOf 或 oSymbolInformation 的第二或后续元素，因为它表示组访问成员所需的访问大小。

• 如果成员数量未知，请调用 GetCountOfSymbolMember()。调用此函数前，请准备 “指定数量 + 1” 的 “SymbolInformation”。

27.10 API 使用注意事项

■ 关于 Pro-Server EX 支持的数据类型

(1)API 可指定、或响应 API 可接收的主要数据类型

定义名称	十进制值	十六进制值	数据含义
EASY_AppKind_Bit	1	0x0001	位数据
EASY_AppKind_SignedWord	2	0x0002	16 位有符号数据
EASY_AppKind_UnsignedWord	3	0x0003	16 位无符号数据
EASY_AppKind_HexWord	4	0x0004	16 位 (HEX) 数据
EASY_AppKind_BCDWord	5	0x0005	16 位 (BCD) 数据
EASY_AppKind_SignedDWord	6	0x0006	32 位有符号数据
EASY_AppKind_UnsignedDWord	7	0x0007	32 位无符号数据
EASY_AppKind_HexDWord	8	0x0008	32 位 (HEX) 数据
EASY_AppKind_BCDDWord	9	0x0009	32 位 (BCD) 数据
EASY_AppKind_Float	10	0xA	单精度浮点数据
EASY_AppKind_Real	11	0xB	双精度浮点数据
EASY_AppKind_Str	12	0xC	字符串数据
EASY_AppKind_SignedByte	13	0x0013	8 位有符号数据
EASY_AppKind_UnsignedByte	14	0x0014	8 位无符号数据
EASY_AppKind_HexByte	15	0x0015	8 位 (HEX) 数据
EASY_AppKind_BCDByte	16	0x0016	8 位 (BCD) 数据
EASY_AppKind_TIME	17	0x0017	TIME 数据
EASY_AppKind_TIME_OF_DAY	18	0x0018	TIME_OF_DAY 数据
EASY_AppKind_DATE	19	0x0019	DATE 数据
EASY_AppKind_DATE_AND_TIME	20	0x0020	DATE_AND_TIME 数据

(2) 特殊情况下的可用数据类型

定义名称	十进制值	十六进制值	数据含义
EASY_AppKind_NULL	0	0x0000	表示可将符号作为寄存器地址使用的 API 使用了为符号定义的数据类型。
EASY_AppKind_BOOL	513	0x0201	作为 Variant BOOL 数据按位处理位数据。
EASY_AppKind_Group	-32768	0x8000	组符号
EASY_AppKind_SymbolSheet	-28672	0x9000	符号表

■ 关于带控制器 /PLC 名称的入口节点名称

(1) GP4000 系列节点、 GP3000 系列节点、 WinGP 节点和 LT3000 节点可连接多台控制器 /PLC。要访问这些控制器 /PLC，必须指定入口节点和控制器 /PLC 的名称。

(2) 对于有些 Pro-Server EX API 的参数，只需指定入口节点的名称。有些则必须连同入口节点名称一起指定控制器 /PLC 的名称。
< 如何指定控制器 /PLC 名称 >
指定控制器 /PLC 名称时，在入口节点 名称后加 “.” 号。
例如)
AGPNode.PLC1

(3) 如果访问的寄存器在 GP4000 系列节点 /GP3000 系列节点 /WinGP 节点 /LT3000 节点或 Pro-Server EX 节点内，请将控制器 /PLC 名称指定为 “#INTERNAL”。 (可省略。)

(4) 如果访问的是 GP4000 系列节点 /GP3000 系列节点 /WinGP 节点 /LT3000 节点内 Memory Link 驱动程序的存储器，请将控制器 /PLC 名称指定为 “#MEMLINK”。 (不可省略。)

(5) 如需访问 GP 系列节点或 Pro-Server EX 节点，不需要指定控制器 /PLC 名称。
("." (点号) 不需要。)

(6) 访问分配了 GP4000 系列节点 /GP3000 系列节点 /WinGP 节点 /LT3000 节点的内部寄存器或系统区的控制器 /PLC 时，通过指定带控制器 /PLC 名称的参与节点名称，可省略控制器 /PLC 名称的指定。
但是在这种情况下， Pro-Server EX 搜索目标寄存器的顺序是：先搜索内部寄存器，再搜索分配到系统区的控制器 /PLC。

■ 关于符号搜索顺序

对于 Pro-Server EX 的寄存器访问 API，必须指定带控制器 /PLC 名称的入口节点名称，以及作为字符串的寄存器地址或寄存器符号。 Pro-Server EX 根据以下顺序判断指定的字符串直接指定的是寄存器地址还是寄存器符号。

(1) Pro-Server EX 搜索符号表，查看是否有匹配的名称。如果指定的字符串存在于符号表中，则将其视为表。

(2) Pro-Server EX 将指定字符串视为组名称或符号，并搜索本地符号表。如果指定的字符串存在于本地符号表中，则将其视为本地符号。

(3) 如果指定的字符串不存在于本地符号表中，则 Pro-Server EX 搜索全局符号表。 (这里的目标全局符号表属于已用 “带控制器 /PLC 名称的入口节点名称” 指定的控制器 /PLC。不搜索其他控制器 /PLC 的全局符号表。)

(4) 如果指定的字符串不存在于全局符号表中，则将其视为寄存器地址。

■ 名称重复

Pro-Server EX 提供以下名称类别：

(1) 节点名称

(2) 控制器 /PLC 名称

(3) 触发条件名称

(4) 符号表名称

(5) 组 / 符号名称

(6) ACTION 名称

原则上， Pro-Server EX 不允许名称重复，但以下两种情况除外：

(1) 控制器 /PLC 名称重复不会产生问题，例如它们分属于不同的入口节点。

(2) 组 / 符号名称重复不会产生问题，例如它们分属于不同的入口节点或不同的控制器 /PLC。

■ 全局符号名称和本地符号名称重复

如果 Pro-Server EX API 使用符号来指定寄存器地址，且本地符号和全局符号均使用此相同的符号名称，则会将其视为本地符号。

■ 将 Pro-Server EX API 用于多线程应用程序

所有 Pro-Server EX API 的函数均为同步型。 (函数一经调用，直到处理完成才会返回。)

因此，当 Pro-Server EX 用单线程应用程序访问多个入口节点时，处理是逐个节点依次执行的。

而对于多线程应用程序，即使已使用一个线程访问一个入口节点， Pro-Server EX 也能通过另一个线程访问另一个入口节点。

Pro-Server EX API 可用于多线程应用程序。

创建多线程应用程序时，请注意以下几点：

(1) 原则上，请用多句柄函数执行多线程应用程序。

(2) 使用多句柄函数需要创建 Pro-Server EX 句柄。各线程请分别使用不同的 Pro-Server EX 句柄。

即使为一个线程创建了多个 Pro-Server EX 句柄，也不会产生问题。但是，不能使用为其他线程创建的 Pro-Server EX 句柄。

释放 Pro-Server EX 句柄时，请使用为其创建了句柄的同一线程。

(3) 使用 Pro-Server EX API 前，必须先调用 EasyInit()。

不过，在 EasyInit() 之前调用各 API 时，大多数 Pro-Server EX API 会自动调用 EasyInit()。

因此，在使用单线程应用程序时，不必在程序中考虑 EasyInit()。

(4) 调用 EasyInit() 的线程必须一直存在，直到应用程序结束。如果调用 EasyInit() 的线程在应用程序运行过程中关闭，则不能保证运行正常。

(5) 对一般程序而言，用于启动应用程序的线程会一直存在，直到应用程序结束。 (通常这适用于由 VB 或 VC 创建的应用程序。) 因此，创建多线程应用程序时，建议在启动应用程序时调用 EasyInit()。

■ 提高缓冲区更新的效率

(1) 使用缓冲存储函数时，须先在缓冲区中注册寄存器。（可在 Pro-Studio EX 缓冲存储注册画面注册寄存器，也可使用缓冲区控制 API。）
整个系统的性能因注册方式而不同。

(2) 选择准备注册的寄存器时，请使用寄存器访问日志函数来识别 Pro-Server EX 访问的寄存器。

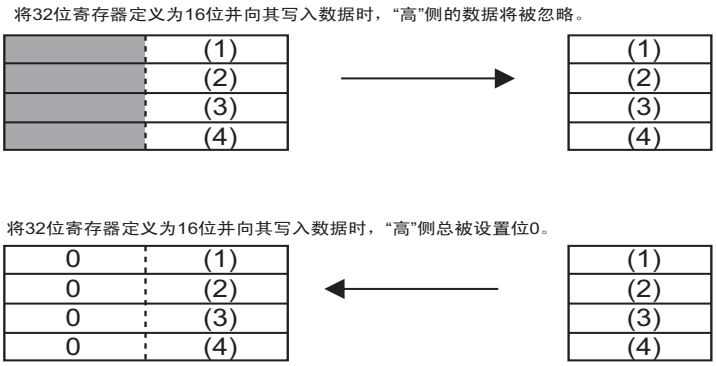
(3) 原则上应缓冲注册频繁读取的寄存器。

(4) 注册多个寄存器时，如果能按序列进行注册，处理速度会加快。
（例 1) 在缓冲区中注册 LS100 和 LS101，如果从 LS100 开始按序列注册，处理速度会比分别注册要快。另外，如果两个寄存器之间仅相隔几个字，则按序列注册要比分别注册处理速度快。
（例 2) 在缓冲区中注册 LS100 和 LS103，如果从 LS100 开始按序列注册 4 个寄存器，处理速度会比分别注册要快。

(5) 注册多个位寄存器时，如果能按字寄存器进行注册，处理速度会加快。
（例）从 LS123401 顺序注册 20 个位寄存器，如果从 LS1234 开始注册两个字，则处理速度会加快。

■ 32 位寄存器的 16 位访问操作

(1) 若将 16 位符号分配给了实际长度为 32 位的寄存器，且用 16 位符号访问该寄存器，或访问 32 位寄存器时直接指定了 16 位数据类型， Pro-Server EX 可以将 32 位寄存器当作 16 位寄存器进行处理。
此时， Pro-Server EX 对读取和写入 API 执行以下转换。



(2) 上述转换在使用数据传输函数或 API 进行访问的过程中执行。

(3) 在 GP 系列节点之间进行数据传输时，将发生错误。

(4) 在低版本的 Pro-Server 中，如果对实际长度为 32 位的寄存器执行 16 位访问，将发生错误。

■ 32 位寄存器的 16 位访问操作

(1) 若将 32 位符号分配给了实际长度为 16 位的寄存器，且用 32 位符号访问该寄存器，或访问 16 位寄存器时直接指定了 32 位数据类型， Pro-Server EX 可以将 16 位寄存器当作 32 位寄存器进行处理。
此时， Pro-Server EX 将两个连续 16 位寄存器当作一个寄存器进行处理。

■ 关于 Pro-Server 自动启动、强制关闭和重启

(1) 若 Pro-Server EX 尚未启动，调用 Pro-Server EX API 将自动启动 Pro-Server EX(部分 API 除外)。
如果 Pro-Server EX 不能启动， API 会返回一个错误代码。

(2) Pro-Server EX 正常启动后，调用第二个或后续 API 不会再次启动 Pro-Server EX，因为 Pro-Server EX 已经启动。

(3) 如果在程序运行过程中关闭 Pro-Server EX，然后调用一个 API(调用第二个或后续 API 时 Pro-Server EX 已关闭)， API 不会启动 Pro-Server EX。将返回一个错误代码。

(4) 在程序运行过程中请勿关闭 Pro-Server EX。
关闭 Pro-Server EX 前，请务必先关闭应用程序。 (关闭 Pro-Server EX 后，请勿调用 API。)
但是，如果从 Windows 启动菜单中人工启动 Pro-Server EX， API 将执行 Pro-Server EX 恢复处理，并尝试继续处理。如果能够恢复 Pro-Server EX，它将继续处理。但是，根据先前的关闭方式， Pro-Server EX 的恢复处理也可能失败。
例如，恢复处理可能在以下情况下失败：
- 从任务管理器中强制关闭了 Pro-Server EX
- 在调用 API 的过程中关闭了 Pro-Server EX

■ 关于指定符号索引

只有用于 API 的寄存器名称可指定符号索引。指定符号索引是指在符号名称后的 [] 中指定一个值，如下所示。符号索引指定的寄存器位于符号名称所指定寄存器之后，推后的寄存器数量由 [] 中的值与符号的数据类型决定。

(符号名称)[数值]

例如)Valve[2]
如果将 Valve 符号 “D100”指定为 “16-bit signed”， Valve[2] 表示 D102。如果将 “D100”指定为 “32-bit unsigned”，则表示 D104。

■ 关于排队缓冲读取和符号缓冲读取

使用排队缓冲读取 (在 BeginQueuingRead 后用 ReadDevice 函数 (不带 “D”) 进行排队注册) 或符号缓冲读取 (ReadSymbol(不带 “D”)) 时，操作因缓冲注册了哪部分目标寄存器而不同。

- 所有目标寄存器均缓冲注册：执行缓冲读取。
- 所有目标寄存器均未缓冲注册：执行直接读取。
- 仅部分目标寄存器进行了缓冲注册：对部分目标寄存器执行缓冲读取，其余则执行直接读取。但是，并非对所有进行了缓冲注册的寄存器均会执行缓冲读取，而是可能会对其中一部分执行直接读取。如果无法识别对哪些寄存器执行缓冲读取，则应该缓冲注册所有的目标寄存器，或使用直接读取 API 而不是缓冲读取 API。

■ 关于不能用于 .NET 的 API

以下 API 不能用于 .NET。如果使用了这些 API，则无法保存运行正常。

- 符号访问 (字节访问)

ReadDevice()， ReadDeviceD()， WriteDevice()， WriteDeviceD()
ReadDeviceM()， ReadDeviceDM()， WriteDeviceM()， WriteDeviceDM()
ReadSymbol()， ReadSymbolD()， WriteSymbol()， WriteSymbolD()
ReadSymbolM()， ReadSymbolDM()， WriteSymbolM()， WriteSymbolDM()

- 符号大小获取函数

SizeOfSymbol()

■ 关于不能在 VB 函数中使用的 API

以下 API 不能在 Visual Basic 函数中使用。如果使用了这些 API，其运行情况不能得到保证。
ReadDeviceDATE_AND_TIME(), ReadDeviceDATE_AND_TIMEM(),
ReadDeviceDATE_AND_TIMED(), ReadDeviceDATE_AND_TIMEDM(),
WriteDeviceDATE_AND_TIME(), WriteDeviceDATE_AND_TIMEM(),
WriteDeviceDATE_AND_TIMED(), WriteDeviceDATE_AND_TIMEDM(),
EasyStringToDATE_AND_TIME(), EasyDATE_AND_TIMEToString()

■ 在多线程应用程序中使用简单 DLL

所有 Pro-Easy API 的函数均为同步型。 (函数一经调用，直到处理完成才会返回。) 因此，当用单线程应用程序访问多个入口节点时，处理是逐个节点依次执行的。而对于多线程应用程序，即使已使用一个线程访问一个入口节点，也能通过另一个线程访问另一个入口节点。
Pro-Easy API 可用于多线程应用程序。
创建多线程应用程序时，请注意以下几点：

1. 原则上，请用多句柄函数执行多线程应用程序。
2. 使用多句柄函数需要创建 Pro-Server EX 句柄。各线程请分别使用不同的 Pro-Server EX 句柄。即使为一个线程创建了多个 Pro-Server EX 句柄，也不会产生问题。但是，不能使用为其他线程创建的 Pro-Server EX 句柄。释放 Pro-Server EX 句柄时，请使用为其创建了句柄的同一线程。
3. 使用 Pro-Server EX API 前，必须先调用 EasyInit()。由于在 EasyInit() 之前调用各 API 时，大多数 Pro-Server EX API 会自动调用 EasyInit()，因此在设计程序时不必考虑 EasyInit() 的调用。
4. 在多线程程序中，程序必须先从首先启动的线程 (主线程) 中调用 EasyInit()。从主线程之外调用 Pro-Server EX API 时，请先从主线程调用 EasyInit()。

■ Windows 中的消息处理

大多数 Windows 程序是事件驱动的，即根据各种事件显示对话框或播放音频等，事件包括 “点击图标”、“移动鼠标”、“按下按键” 等。

事件发生时， Windows 将向应用程序发送消息，说明事件的类型。应用程序通过接收消息来确认事件的发生，然后执行各处理。

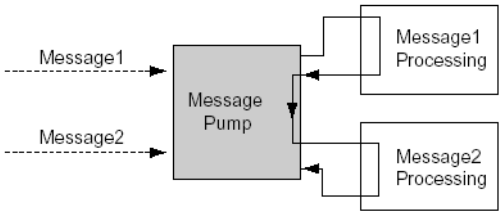
在本手册中，按顺序接收消息并将它们分配到各处理中的部分 (对应于 VB 的 DoEvents 或 VC 中执行 GetMessage() 和 DispatchMessage() 的部分) 称为消息泵。消息泵很少被提及，是因为通常用 VC 或 VB 编程时，它隐藏在 VC 或 VB 框架内。但是，如果此消息泵工作不正常， Windows 应用程序将发生意外操作。

例如，如果一段程序处理一条消息及恢复需花费较长的时间，则应用程序会因为无法从 Windows 接收同时发生的事件而无法处理此事件。

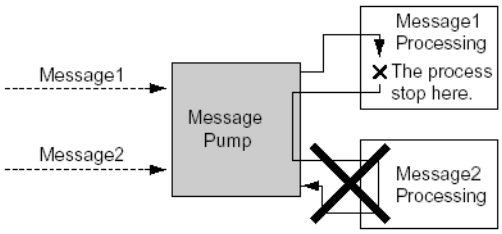
例如) 假设按消息 1、消息 2 的顺序从 Windows 发送消息。

消息泵取出消息 1，然后调用消息 1 的子程序。

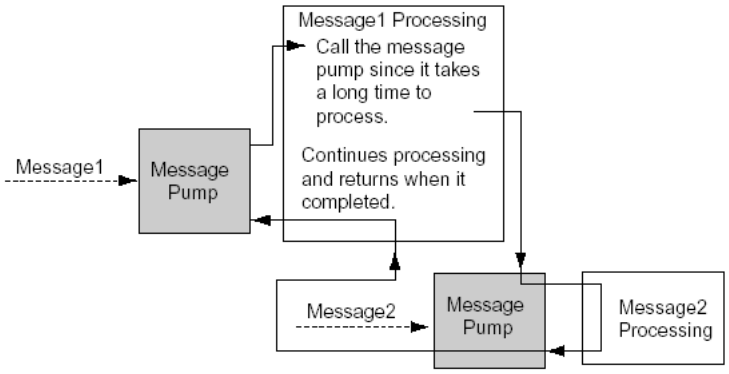
从上述处理恢复后，消息泵取出之后的消息 (消息 2)，然后调用消息 2 的子程序。



在上例中，假设处理消息 1 需要较长时间，则消息泵在未恢复的情况下将无法处理消息 2。



此时，请强制消息泵运行。（使用 VB 时调用 DoEvents，使用 VC 时调用 GetMessage() 和 DispatchMessage()）



Windows 应用程序是在假定一个程序能正确运行消息泵的基础上创建的。 Pro-Server EX API 用函数运行消息泵来执行比较耗时的处理，以避免发生上例中的情况。

■ 禁止 API 两次调用

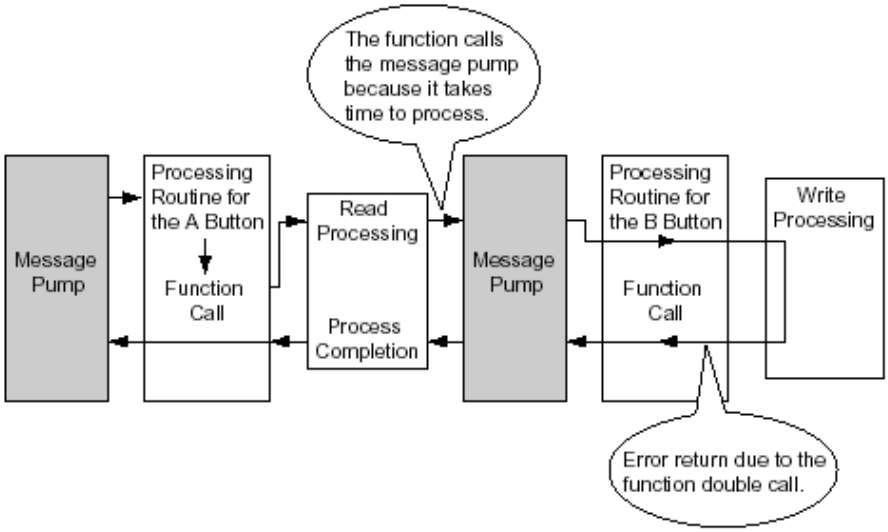
Pro-Server EX API 禁止在与一方通讯的同时 (在调用 Pro-Server EX 函数时) 与另一方进行通讯 (两次调用)。(使用多句柄时允许两次调用。详情请参阅关于多句柄的部分。) 但是，由于 Pro-Server EX API 在 API 内部运行消息泵，当事件发生时，用户程序将启动。

在消息处理程序中调用 API 时，可能会发生两次调用。

以下为两次调用的示例。

1. 按下两个按钮导致的两次调用

   假设有两个按钮， A 和 B。按下 A 会调用寄存器读取 API ；按下 B 会调用寄存器写入 API。在按下 A 调用寄存器读取 API 的同时按下 B 调用寄存器写入 API，将导致 API 两次调用并发生错误。



2. 定时器导致的两次调用

在 Windows 程序中执行周期性处理时，会经常用到定时器事件。但是，如果编程不当，在这些使用定时器事件的程序中就可能发生 API 两次调用。

(1) 每秒定时调用寄存器读取 API，读取寄存器并显示。

(2) 类似于 “按下按钮时调用寄存器写入 API 并将数值写入寄存器” 的程序在以下情况下会导致错误。

在读取定时器事件 (1) 的过程中按下按钮 (2)，处理 (2) 开始运行

在写入 (2) 并读取 (1) 的过程中发生定时器事件

■ 避免 API 两次调用的方法

避免 API 两次调用的方法如下。

(1) 改进运算法则，避免在用户程序中执行 API 两次调用。例如，

   1. 在定时器处理程序和按钮处理程序头部应将定时器取消。
   2. 按下按钮运行一个处理的过程中，再次按下此按钮或另一个按钮应被忽略。

(2) 在使用多句柄的情况下，如果 Pro-Server EX 句柄不同，则不会发生 API 两次调用。

   使用多句柄型 API，将可能发生两次调用区域中的程序句柄设置为不同的句柄。

(3) 不应在 API 内处理消息

   用参数 2 调用 EasySetWaitType()。但是，此时可能发生应用程序执行意外操作等其他问题，因为将不处理除造成两次调用的消息以外的其他消息。

■ 如何在 VB 中读取字符串

(1) 用 ReadDeviceStr 在 VB 中读取字符串

此时，必须事先指定 (固定) 所读取字符串保存目标的大小。

```
'
Public Sub Sample1 ()
   Dim strData As String * 10  ' Correct designation method because it designates the size to read.
   'Dim strData As String      ' Incorrect designation method because it does not designate the character
                               ' string size.
   Dim lErr As Long
   lErr = ReadDeviceStr ("GP1", "LS100", strData, 10)
   If lErr <> 0 Then
       MsgBox "Read Error = " & lErr
   Else
       MsgBox "Read String = " & strData
   End If
End Sub
```

(2) 用 ReadDeviceVariant 在 VB 中读取字符串时使用 Variant 数据类型，而不事先指定所读取字符串保存目标的大小。

```
'
Public Sub Smaple2 ()
    Dim lErr As Long
    Dim vrData As Variant        ' Designate the Variant type to the area to save data read.
    lErr = ReadDeviceVariant ("GP1", "LS100", vrData, 10, EASY_AppKind_Str)
    If lErr <> 0 Then
        MsgBox "Read Error = " & lErr
    Else
        MsgBox "Read String = " & vrData
    End If
End Sub
```

注意， GP 用 NULL 作为字符串的结束标志。为此，如果用上述方法获取的字符串包含 NULL 作为字符串结束标志，则需要缩短字符串。
将字符串缩短至 NULL 的示例函数如下。

```
    Dim i As Integer
    i = InStr (1, strData, Chr$(0), vbBinaryCompare)
    If 0 < i Then
        TrimNull = Left (strData, i - 1)
    Else
        TrimNull = strData
    End If
End Function
```

27.11 使用 API(示例)

通过使用 Pro-Server EX 提供的读取 / 写入函数，可以从 / 向 VB 或 VC 程序读取 / 写入数据。
本节介绍用 API 读取 / 写入指定符号的步骤。

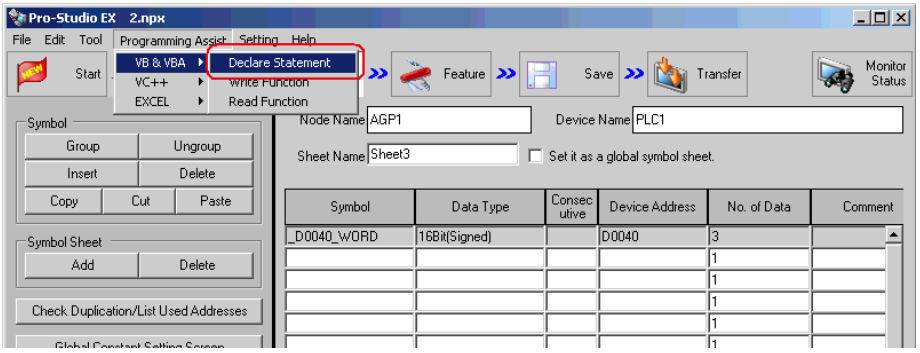
☞ “27.11.1 VB 支持的函数”
☞ “27.11.2 VC 支持的函数”
☞ “27.11.3 VB .NET 支持的函数”
☞ “27.11.4 C# .NET 支持的函数”

27.11.1 VB 支持的函数

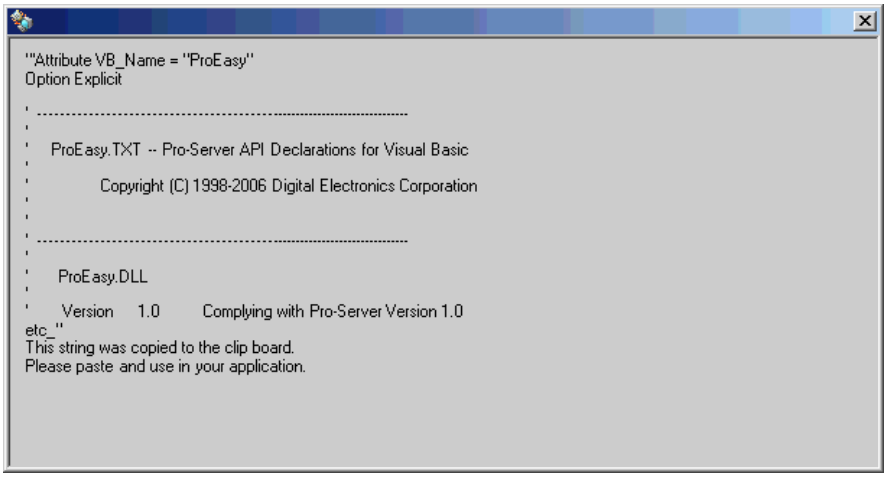
注 释 • 不能在 VB 函数中使用 DATE_AND_TIME 数据类型或 API 函数。

VB：声明语句

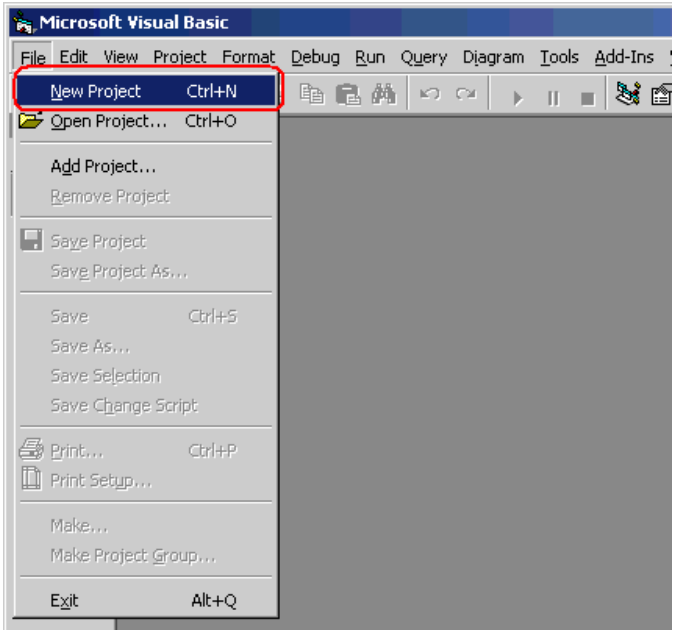
1 选择 [Programming Assist] - [VB & VBA] - [Declare Statement]。



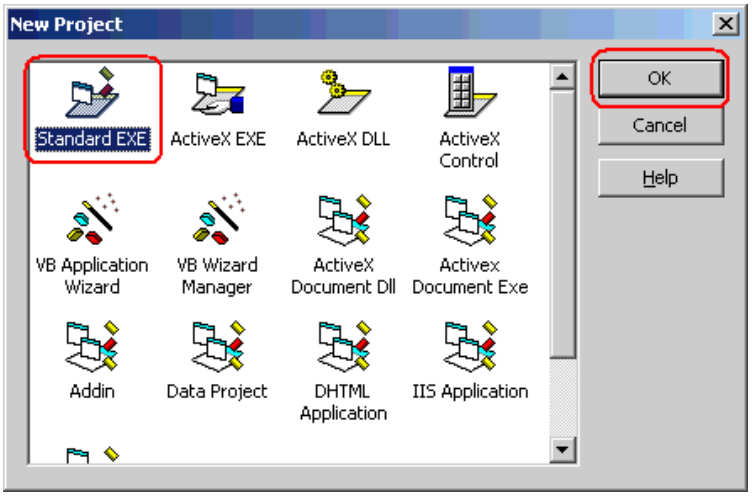
VB 声明语句被复制到剪贴板。



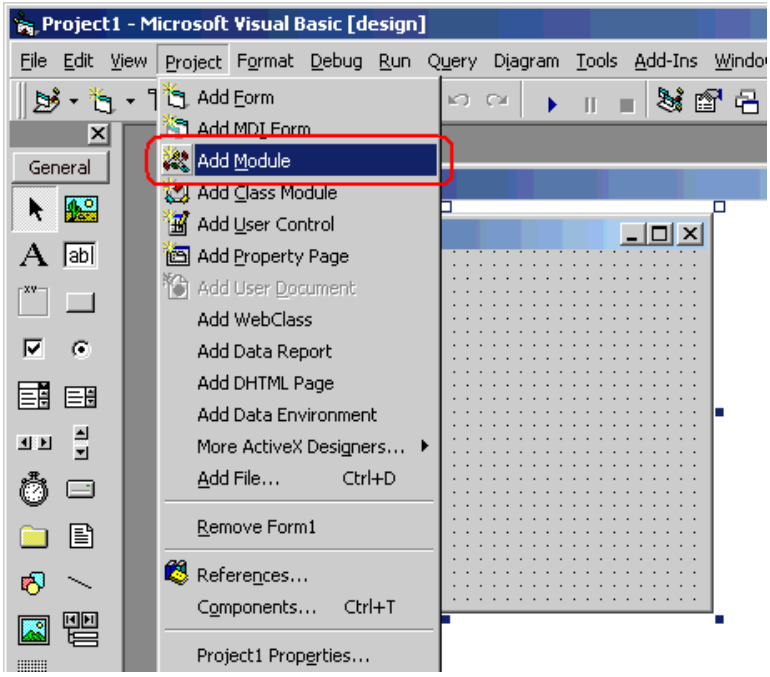
2 启动 Microsoft Visual Basic，从菜单栏的 [File] 中选择 [New Project]。



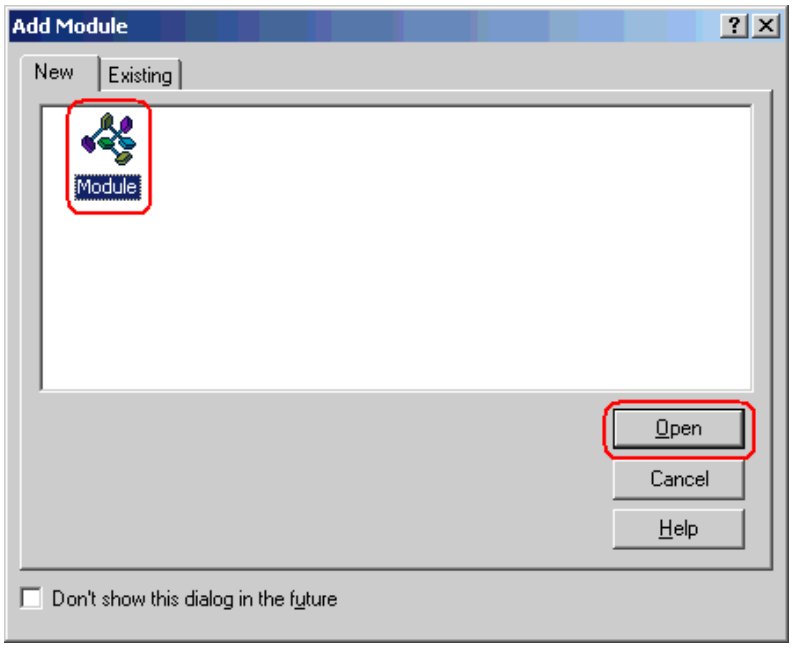
3 选择 [Standard EXE]，点击 [OK] 按钮。



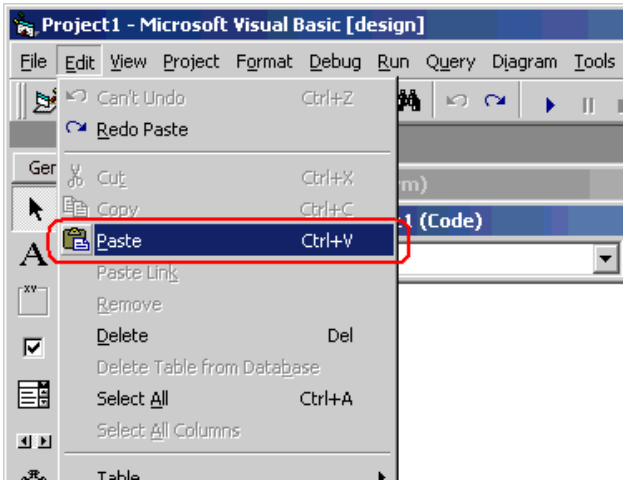
4 从 Microsoft Visual Basic 菜单的 [Project] 中选择 [Add Module]。



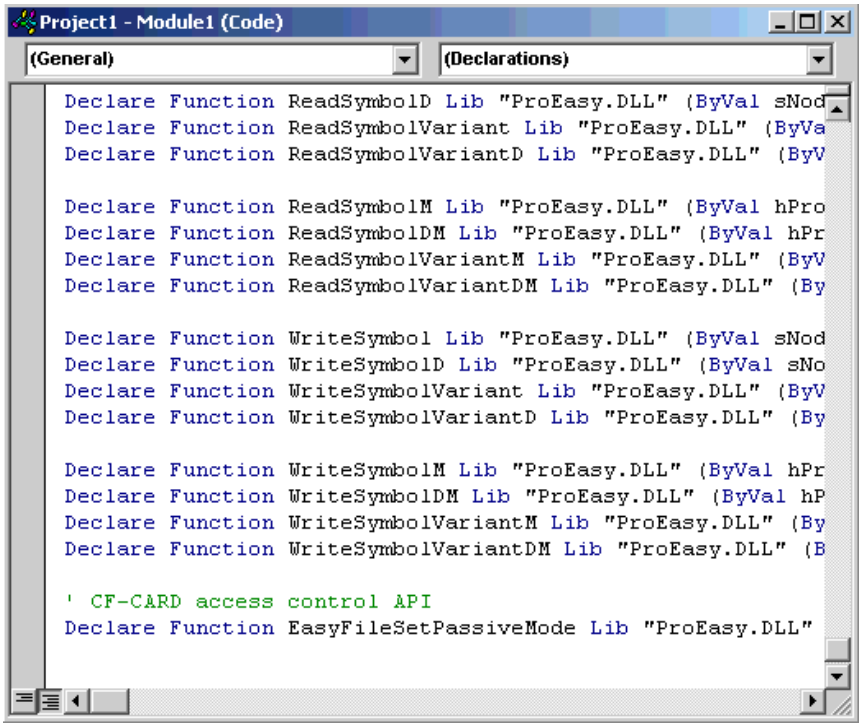
5 选择 [New] 选项卡中的 [Module]，点击 [Open] 按钮。



6 从 Microsoft Visual Basic 菜单的 [Edit] 中选择 [Paste]，将声明语句 (剪贴板上的数据) 粘贴到添加的标准模块上。



声明语句粘贴完毕。



函数 (读取 / 写入函数) 声明步骤至此结束。

上述 1 到 6 步对读取和写入应用程序均适用。

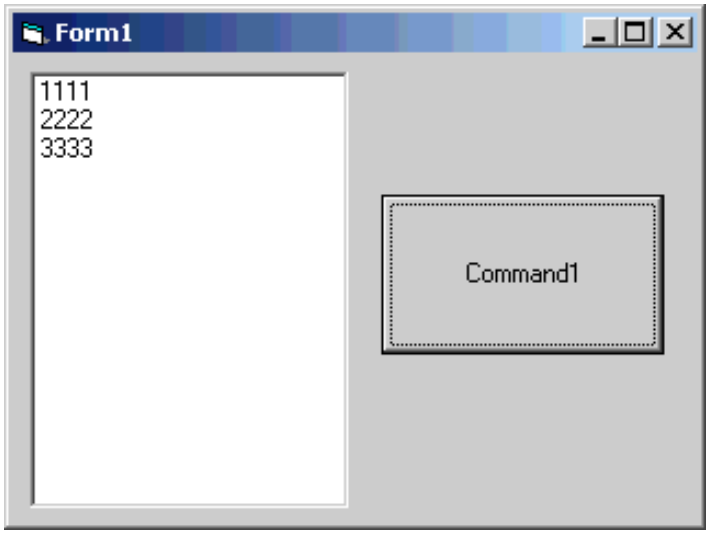
下述步骤则因应用程序是用于读取还是写入而不同，因此分别进行描述。

创建 “读取” 应用程序，请参阅 7 到 16 步。

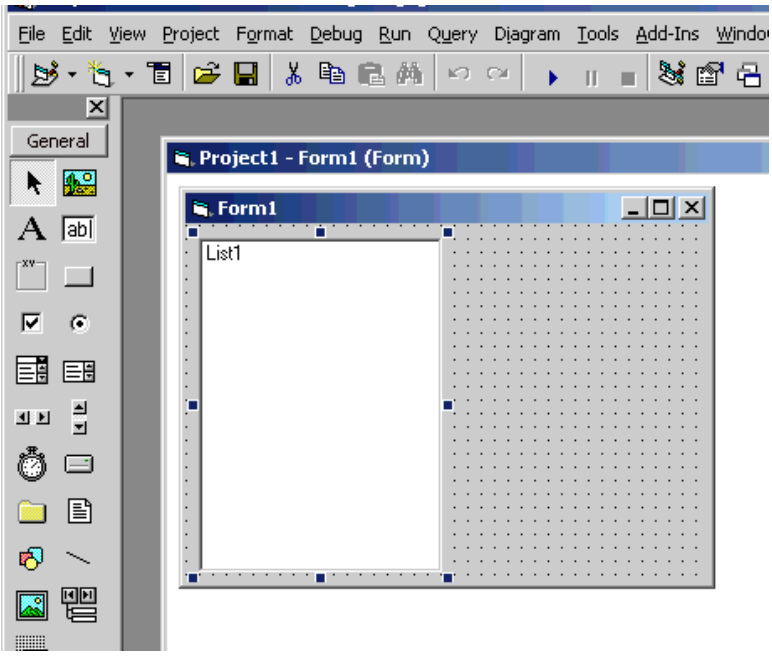
创建 “写入” 应用程序，请参阅 17 到 26 步。

创建 “读取” 应用程序

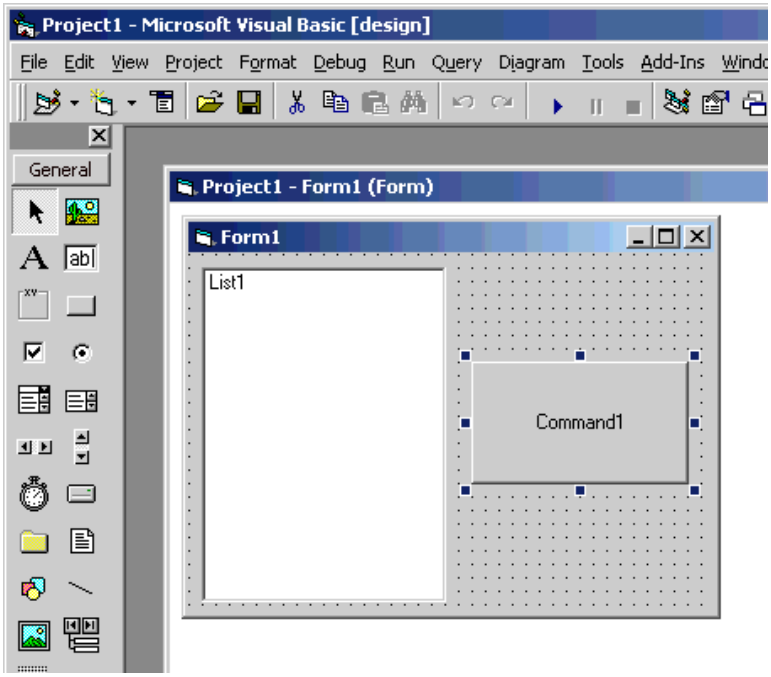
本节介绍创建下述应用程序的过程：点击 [Command1] 读取并显示三个数据 (16 位有符号数据)。



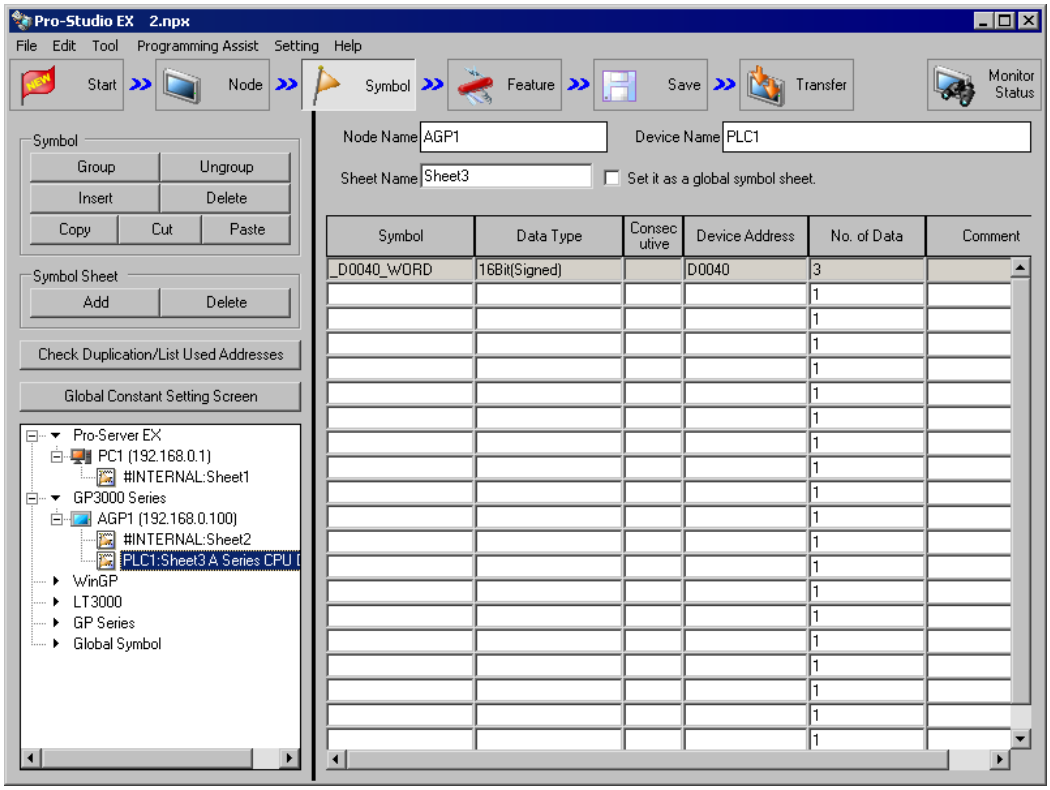
7 选择 [ListBox] 并将它粘贴到 [Form1]。



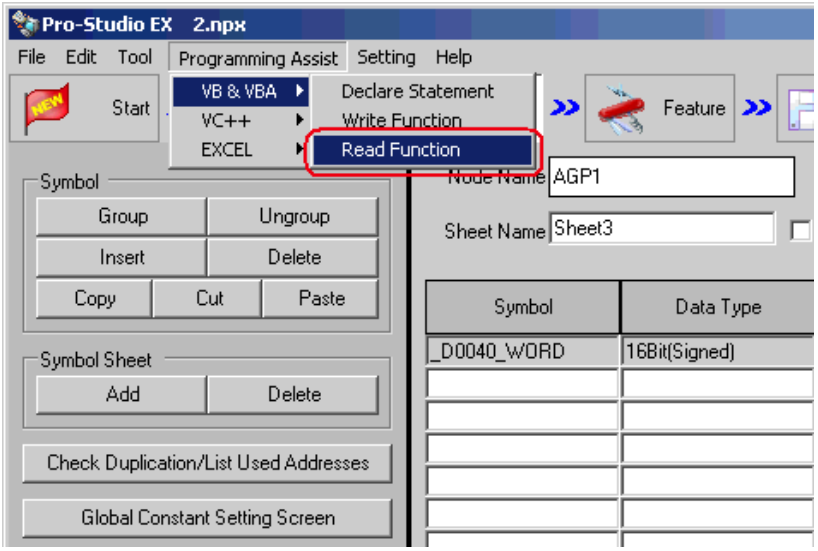
8 选择 [CommandButton] 并将它粘贴到 [Form1]。



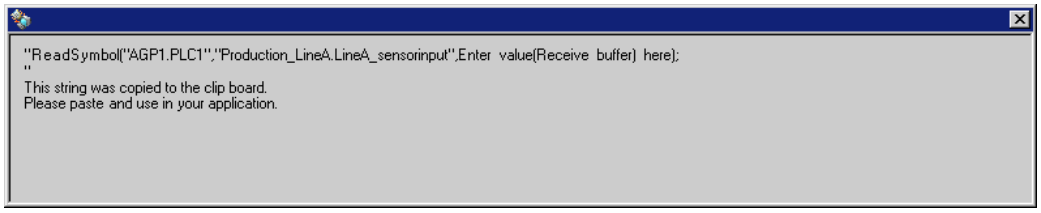
9 从 Pro-Server EX 中注册的符号名称中选择一个目标符号。(选择要读取的首地址对应的符号。)



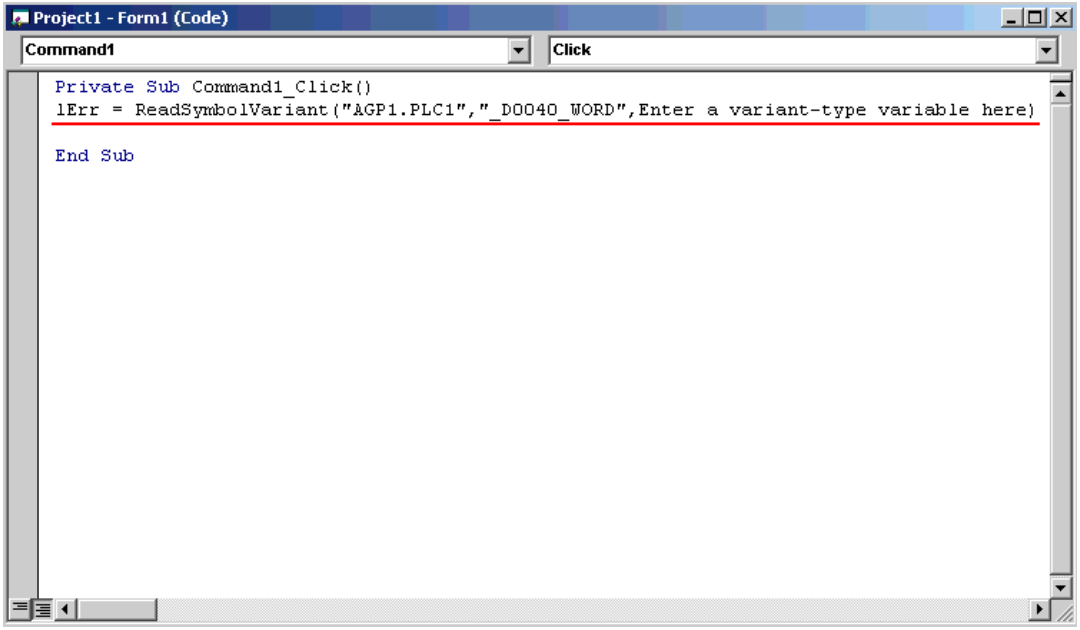
10 从菜单上选择 [Programming Assist] - [VB & VBA] - [Read Function]。



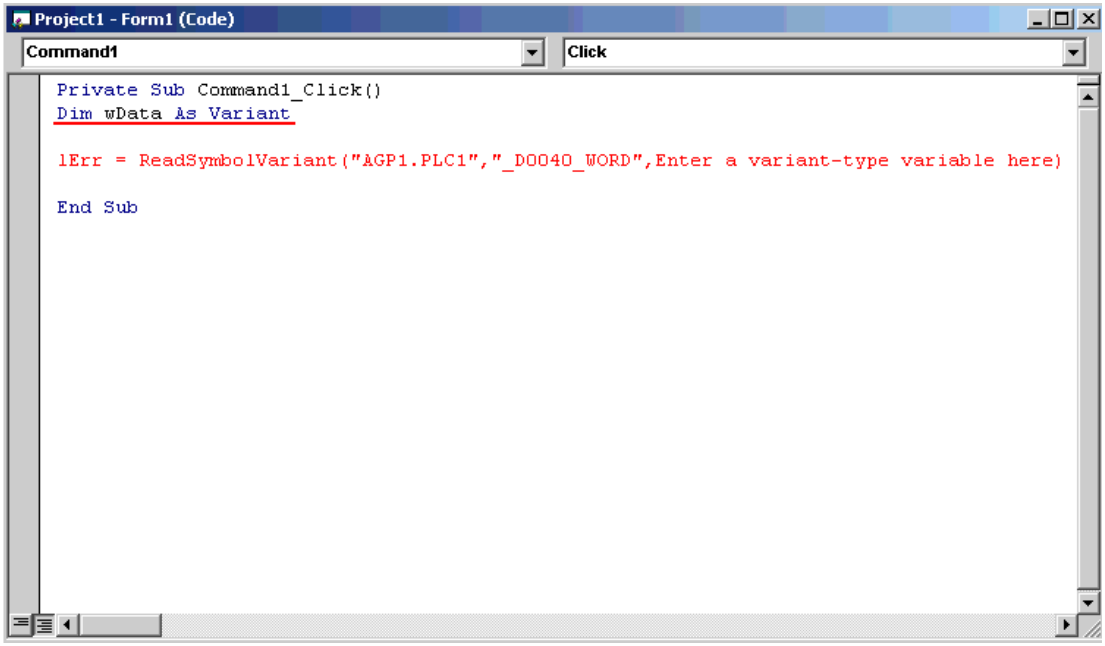
读取函数被复制到剪贴板上。



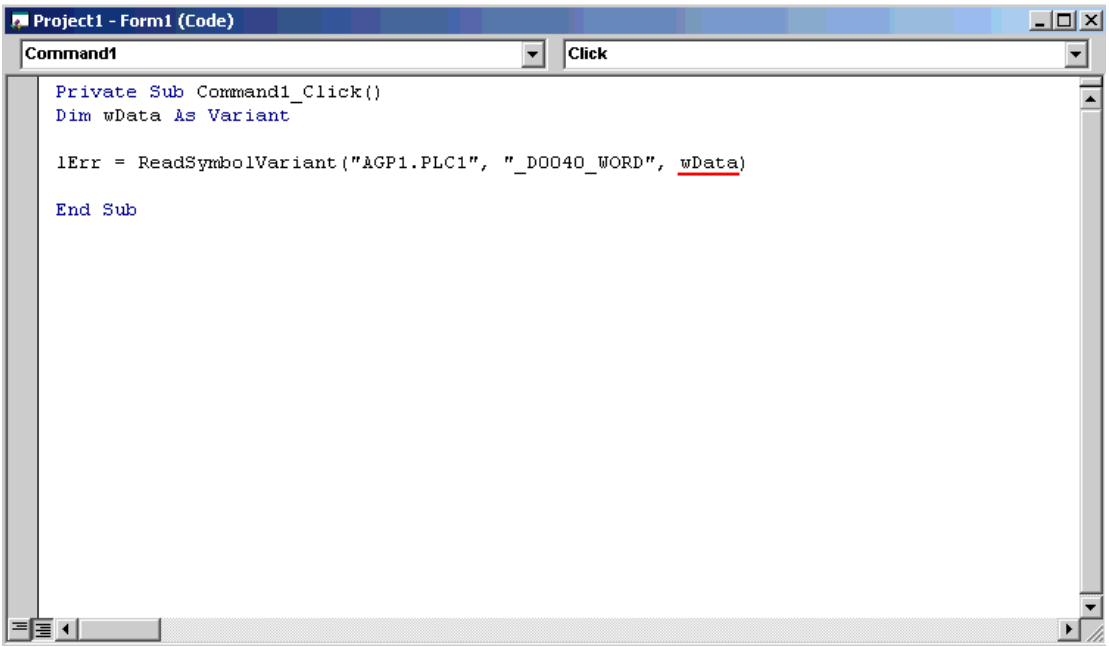
11 双击 [Form1] 上的 [Command1]，将剪贴板上的数据 (读取函数) 粘贴到 “private sub Command1_Click()” 与 “End Sub” 之间。



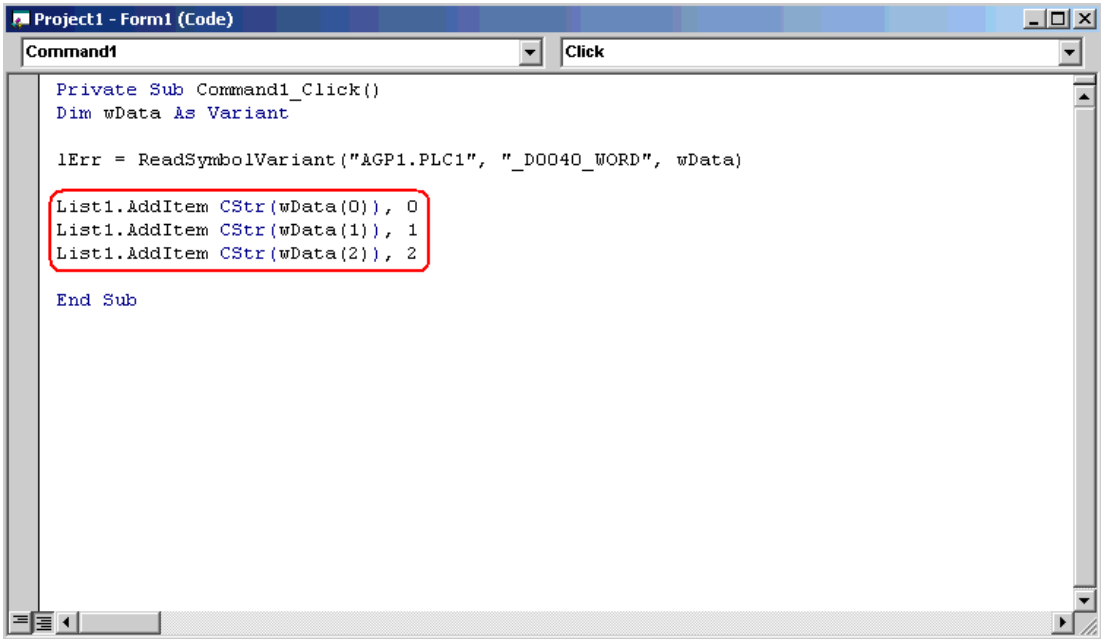
12 声明保存读取数据的区域 (数组)。确认数组类型 (此例中为 Variant 型) 与所用符号的数据类型一致。



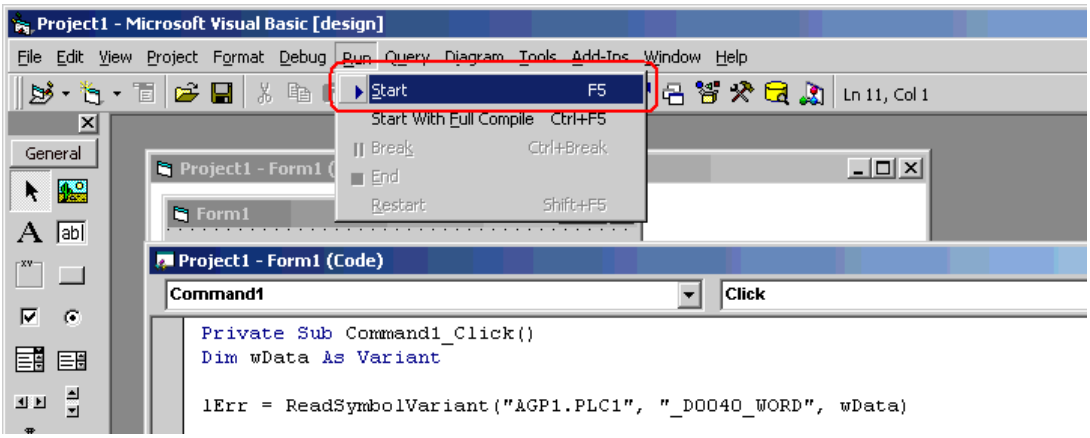
13 指定保存读取数据的首个区域 (wData)。



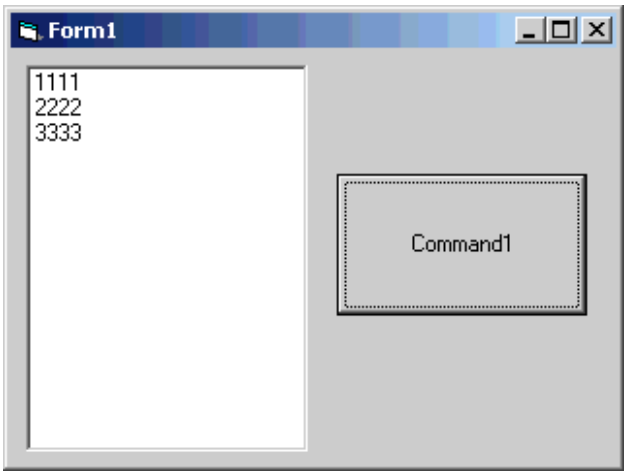
14 列表框按顺序显示三个读取的数据 (wData(0)、 wData(1) 和 wData(2))。



15 从 Microsoft Visual Basic 菜单的 [Run] 中选择 [Start]。

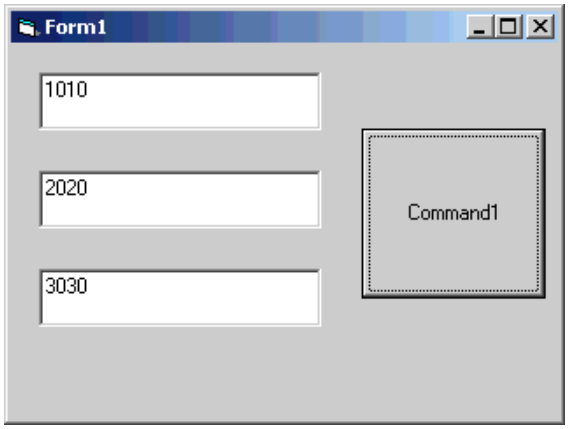


16 点击 [Command1]。然后，列表框从符号 “_D0040_WORD” 开始显示三个数据。

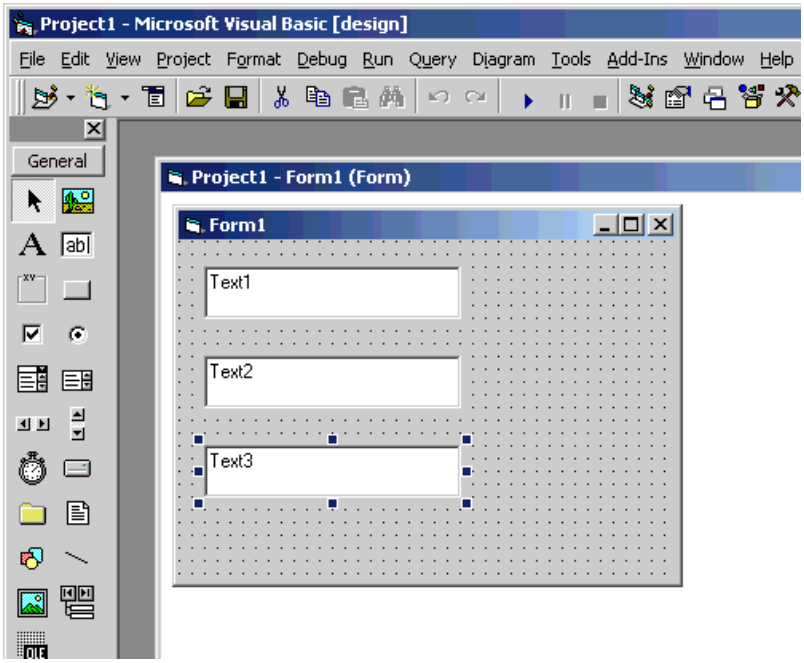


创建 “写入” 应用程序

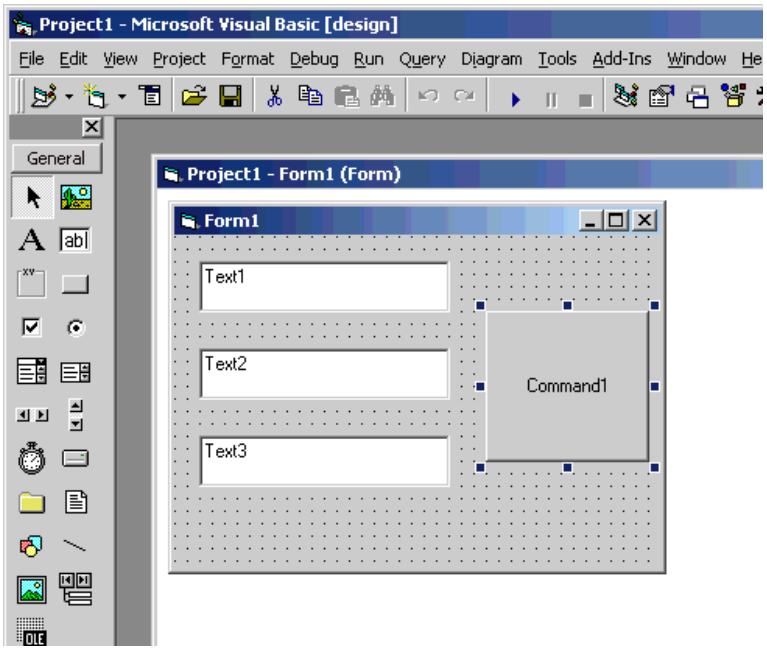
本节介绍创建下述应用程序的过程：点击 [Command1] 写入输入的三个数据 (16 位有符号数据)。



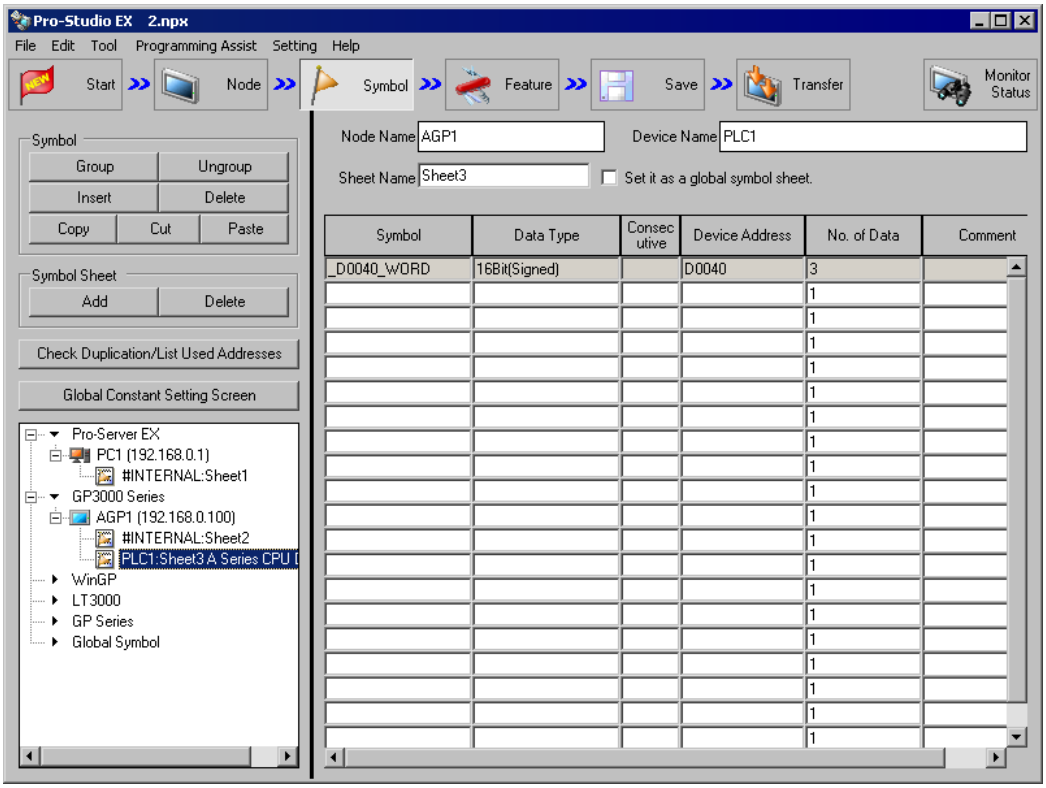
17 选择 [TextBox] 并将它粘贴到 [Form1]。粘贴三个 [TextBox]。



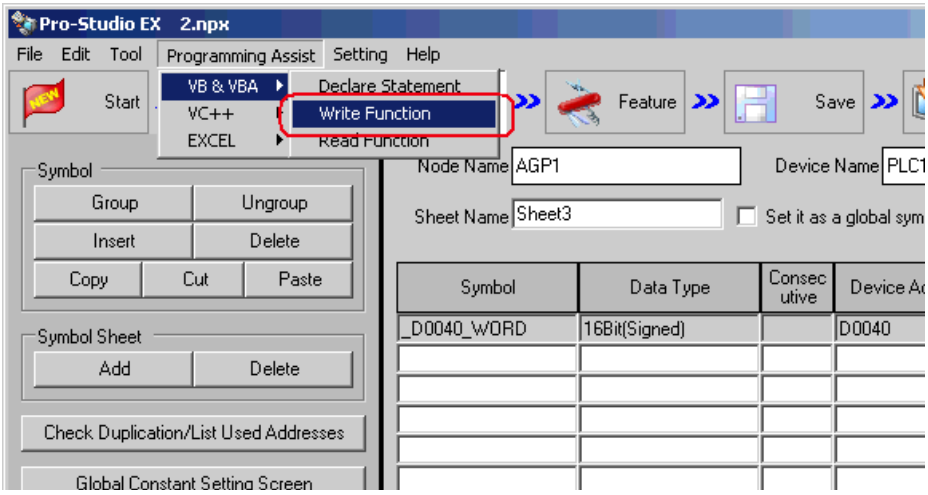
18 选择 [CommandButton] 并将它粘贴到 [Form1]。



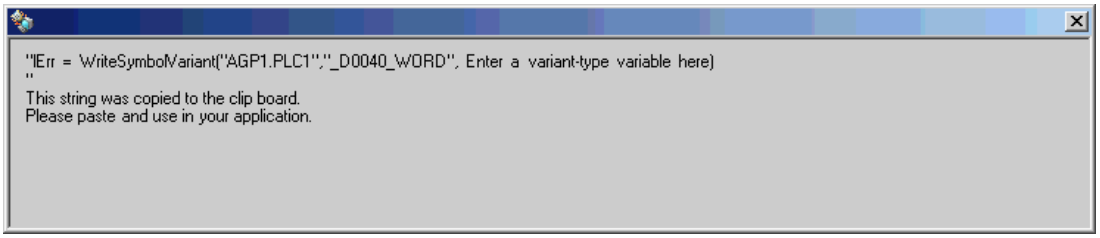
19 从 Pro-Server EX 中注册的符号名称中选择一个目标符号。（选择要写入的首地址对应的符号。）



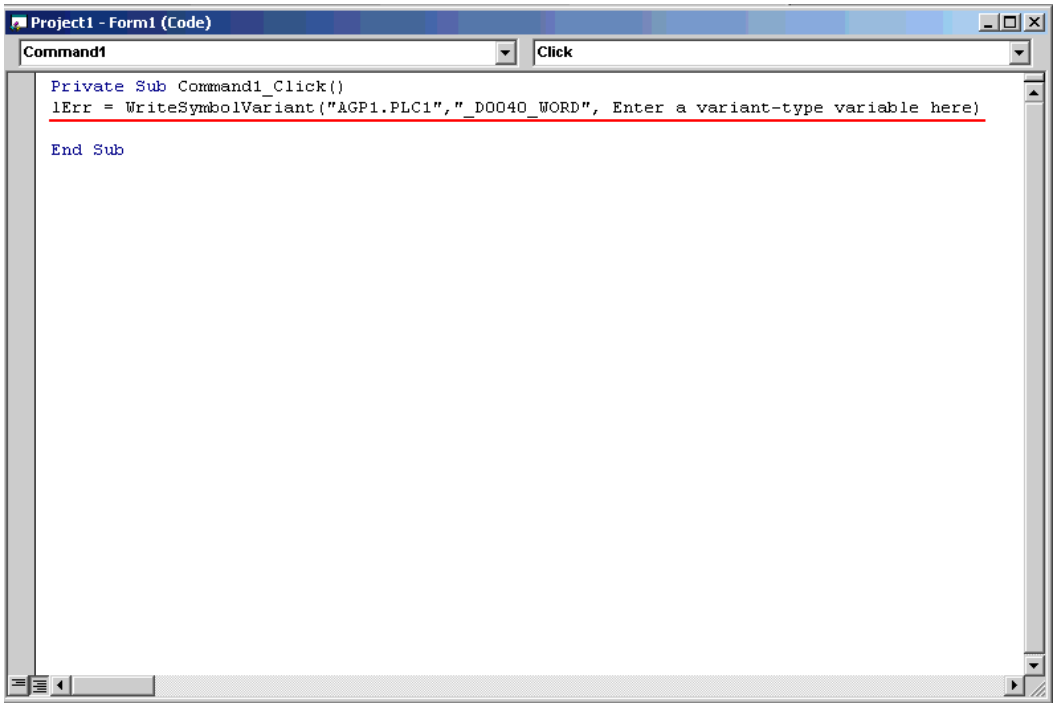
20 从菜单上选择 [Programming Assist] - [VB & VBA] - [Write Function]。



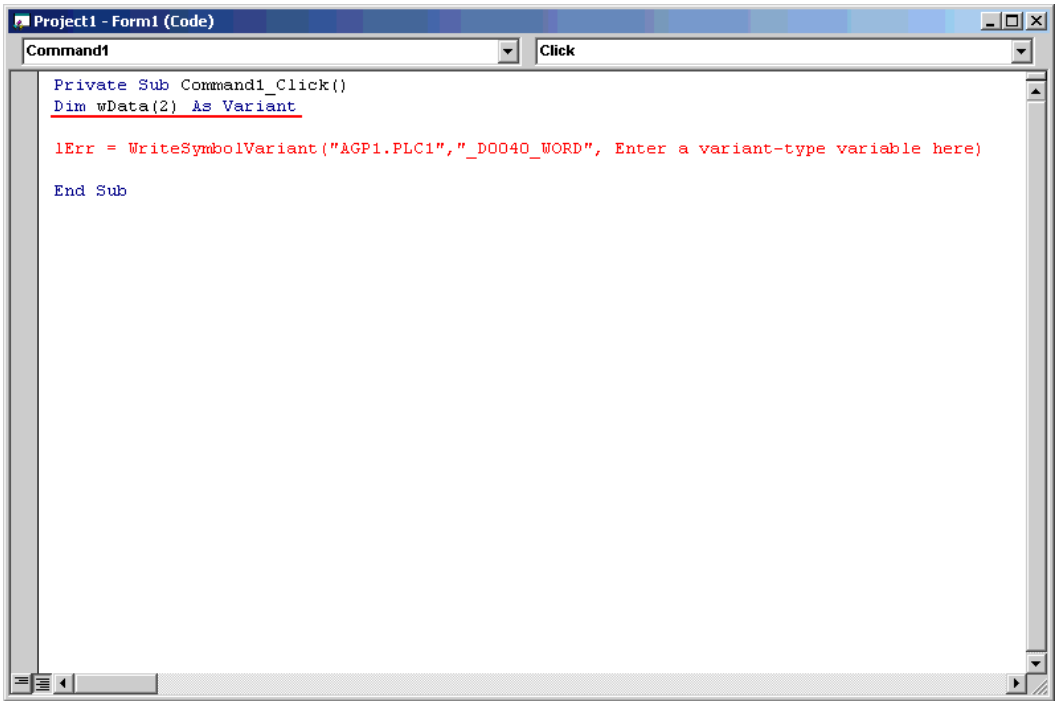
写入函数被复制到剪贴板上。



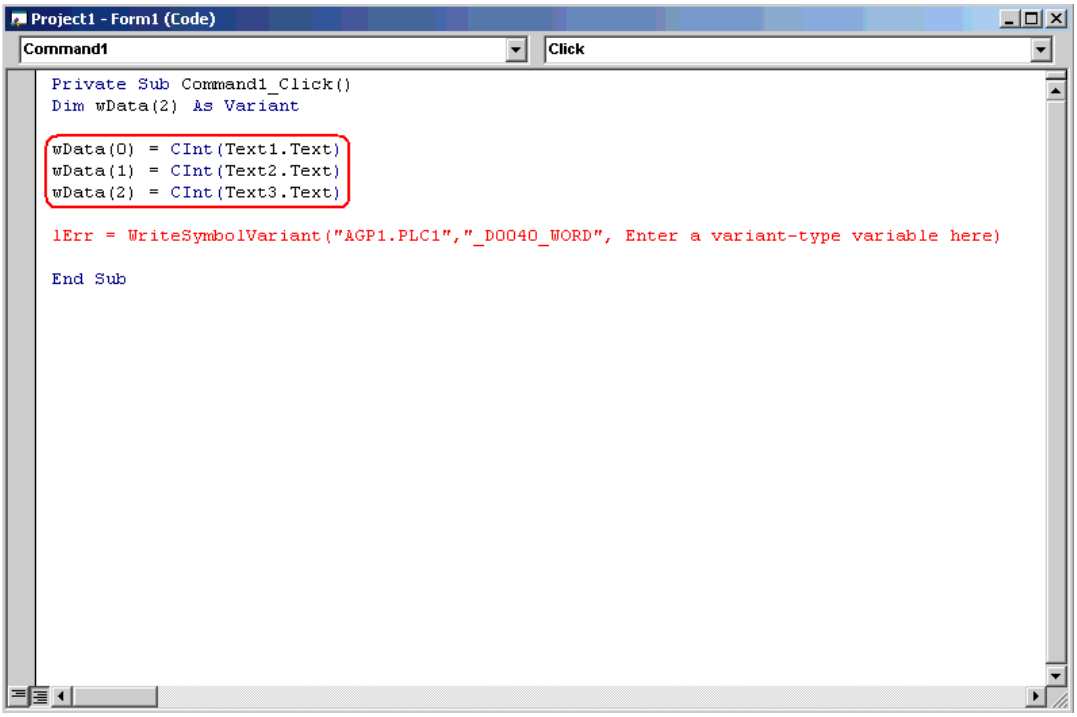
21 双击 [Form1] 上的 [Command1]，将剪贴板上的数据 (写入函数) 粘贴到 “Sub” 语句与 “End Sub” 语句之间。



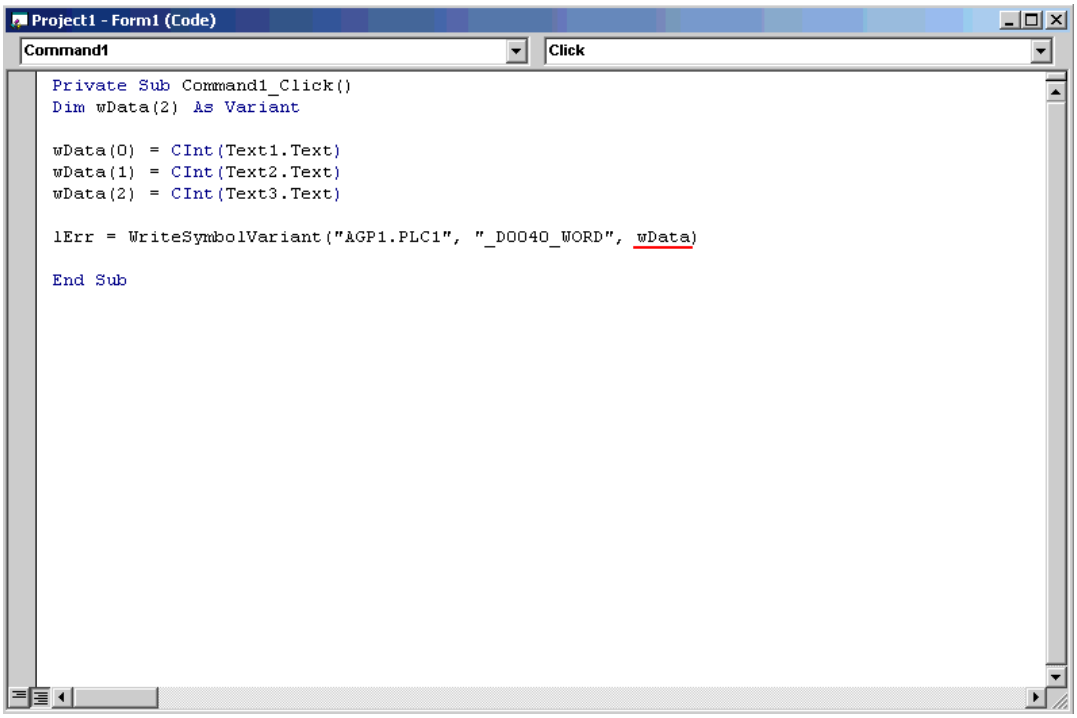
22 声明保存写入数据的区域 (数组)。确认数组类型 (此例中为 Variant 型) 与所用符号的数据类型一致。



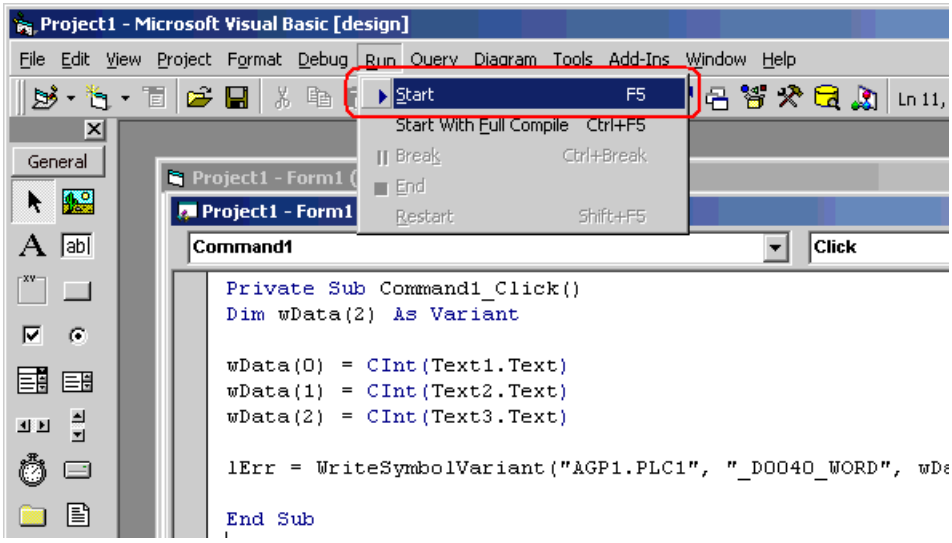
23 将输入 [TextBox] 的数据赋值给数组。



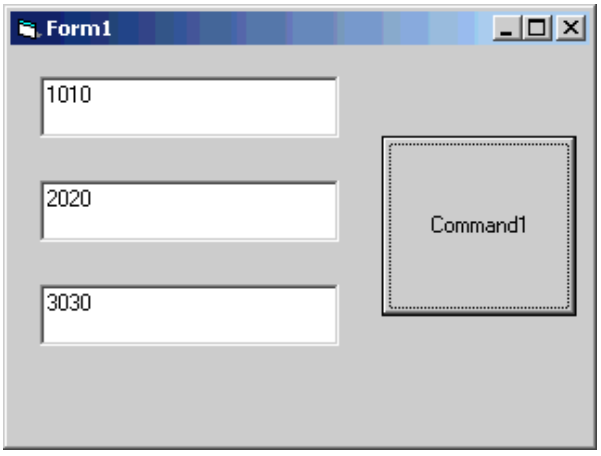
24 指定赋值写入数据的首个区域 (wData)。



25 从 Microsoft Visual Basic 菜单的 [Run] 中选择 [Start]。



26 在 [TextBox] 中输入数值 (三个值) 后，点击 [Command1]。然后， Pro-Server EX 将三个数据写入从符号 “_D0040_WORD” 开始的三个寄存器中。

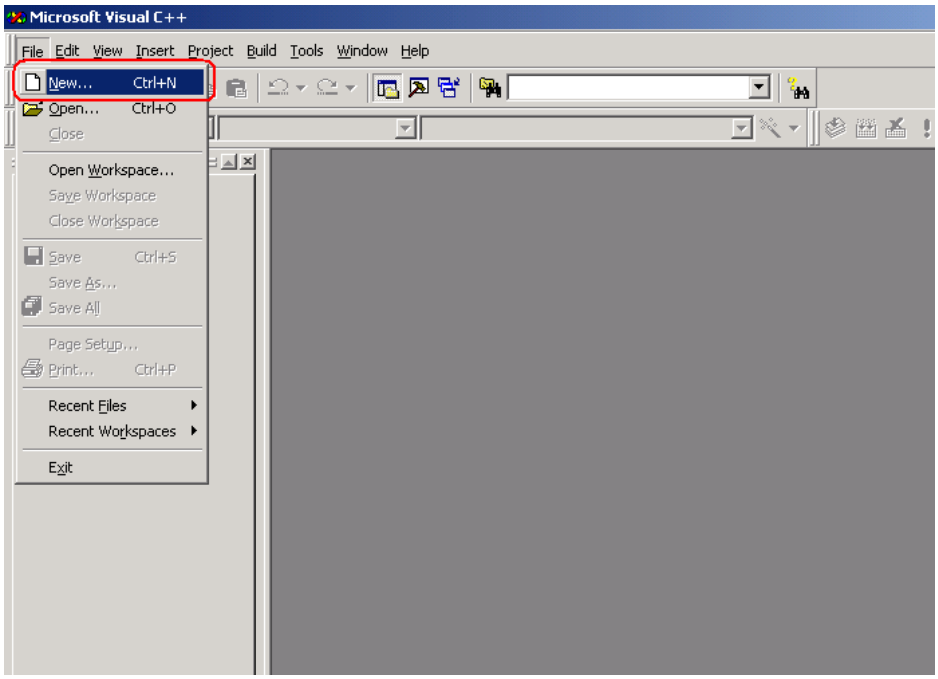


27.11.2 VC 支持的函数

本节用示例介绍用 MFC 创建基于对话的应用程序的步骤。

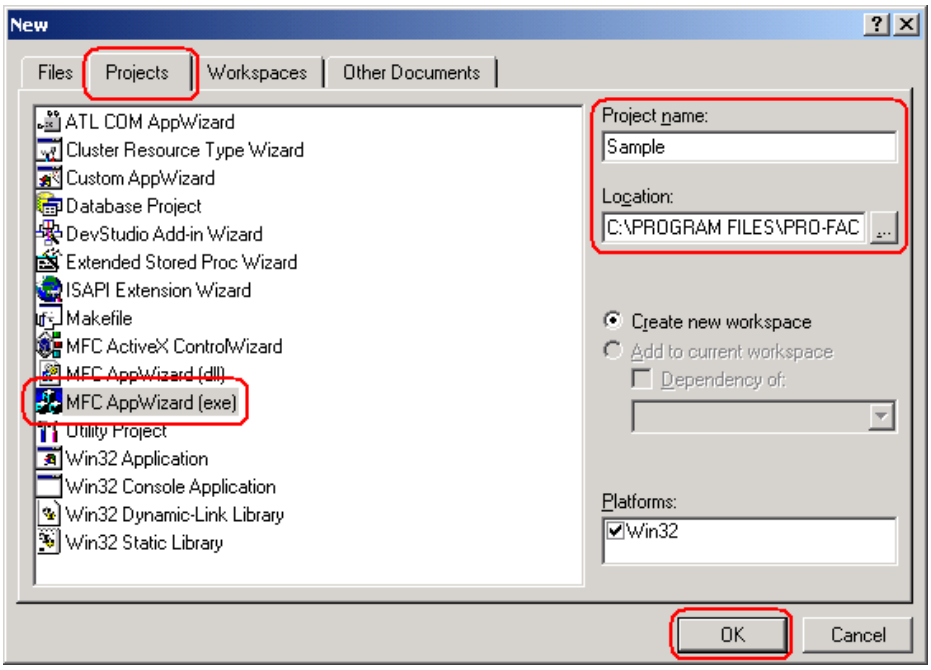
VC: 声明语句

1 启动 Microsoft Visual C++，从 [File] 中选择 [New]。

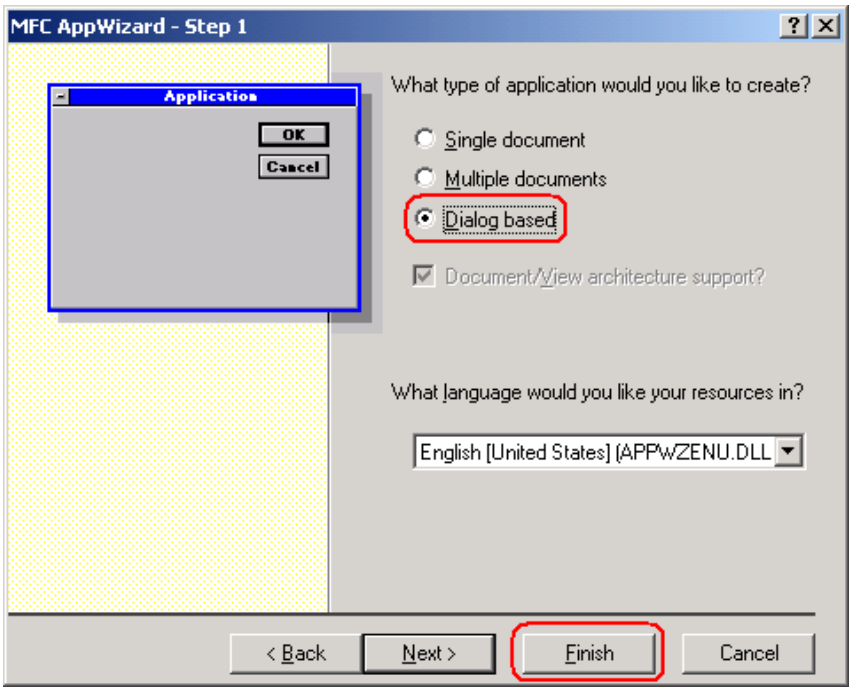


2 在 [Projects] 选项卡中选择 [MFC AppWizard(exe)]，输入 [Project name] 和 [Location]，然后点击 [OK] 按钮。

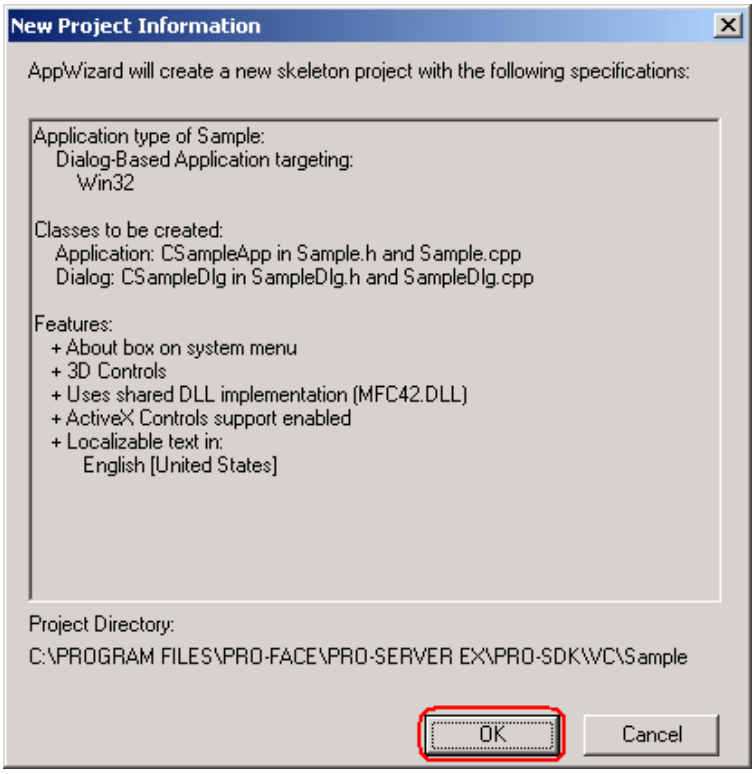
本例在 [Project name] 中输入 “Sample”，在 [Location] 中输入 C:\Program Files\Pro-face\Pro-Server EX\PRO-SDK\VC (Windows Vista 或以上："C:\Pro-face\Pro-Server EX\PRO-SDK\VC")。



3 在 “What type of application would you like to create?” 处选择 [Dialog Based]，点击 [Finish] 按钮。

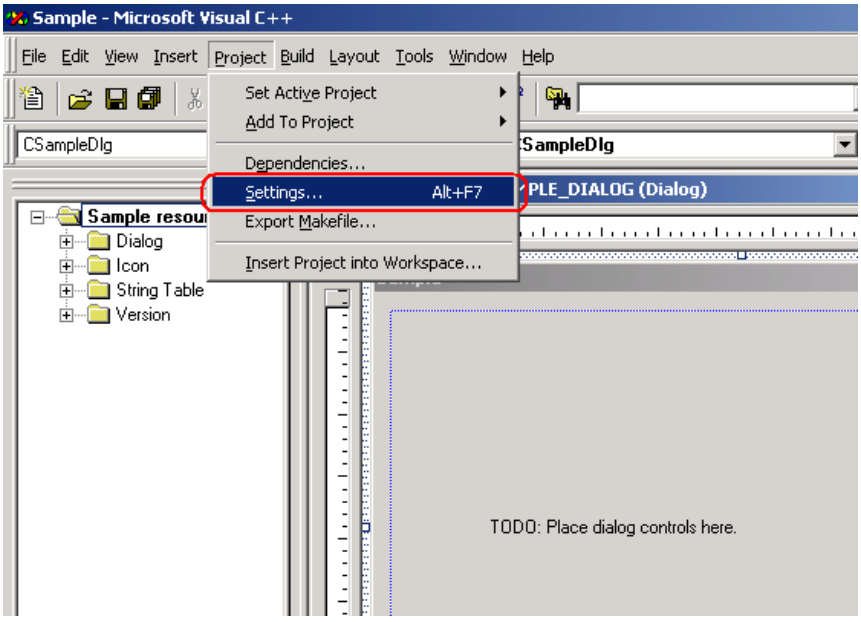


4 点击 [OK] 按钮完成工程。

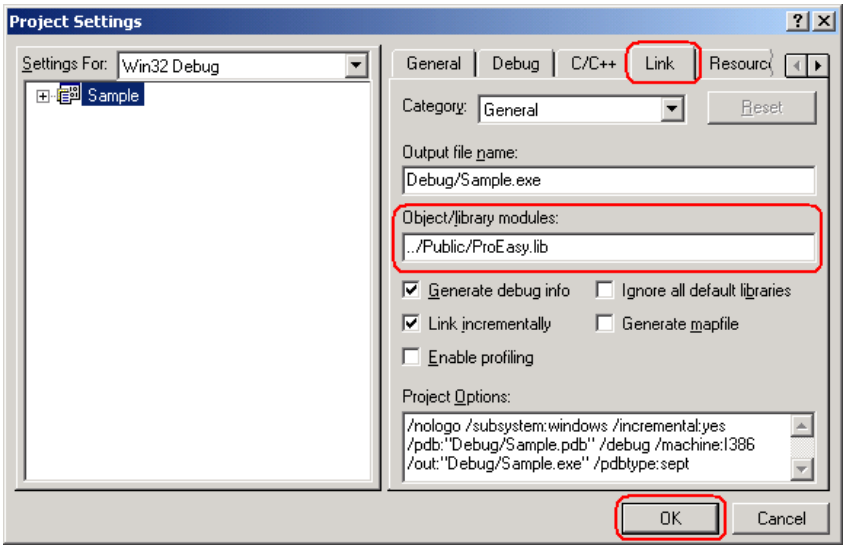


可将 Pro-Server EX 提供的读取 / 写入函数用作 DLL。若要使用 DLL，必须指定一个 LIB 文件。

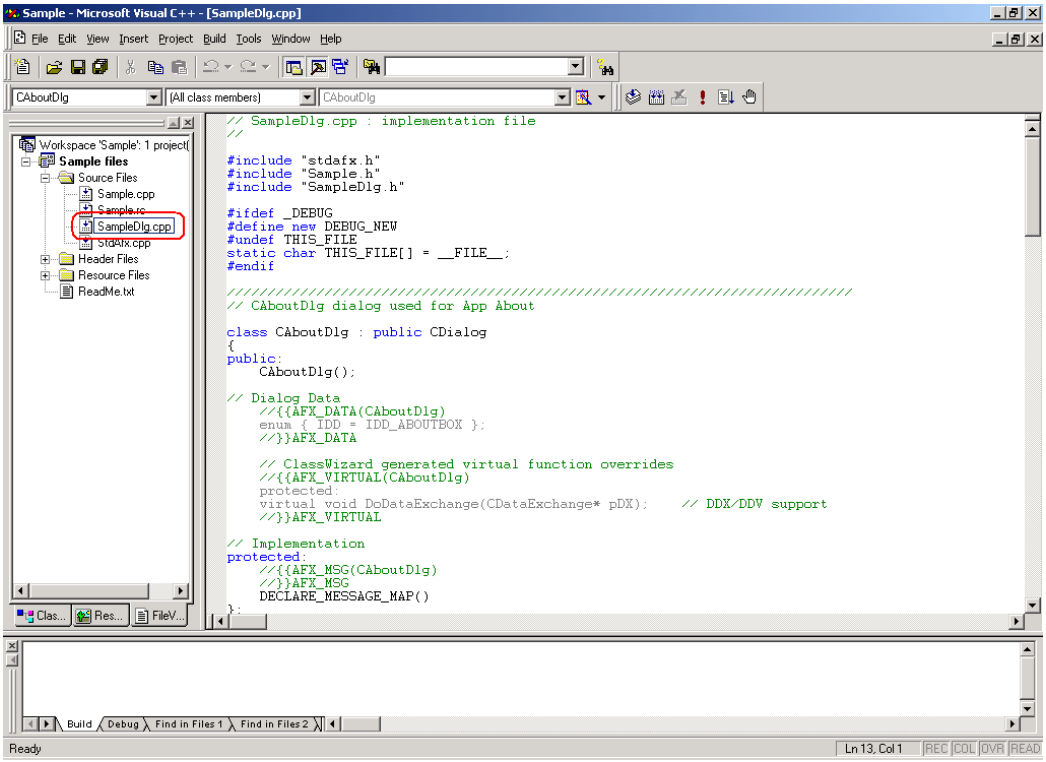
5 从 Microsoft Visual C++ 菜单的 [Project] 中选择 [Settings]。



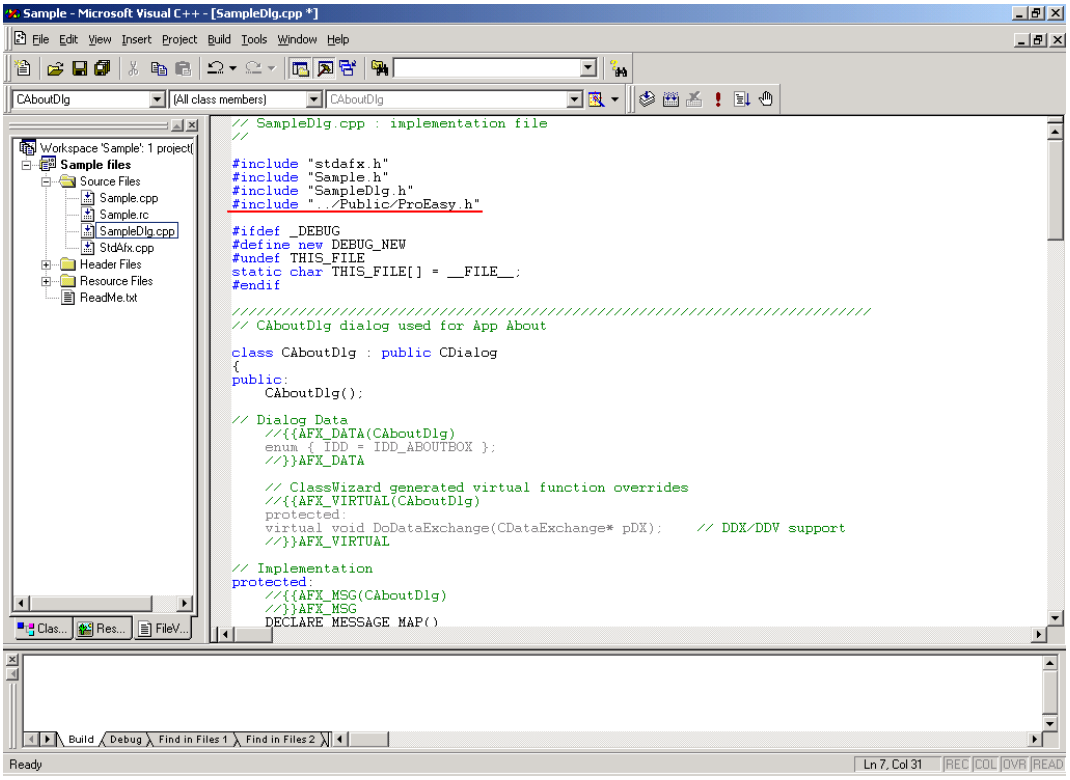
6 在 [Link] 选项卡的 [Object/library modules] 处指定一个 LIB 文件。然后点击 [OK] 按钮。
LIB 文件 (ProEasy.lib) 位于 Pro-Server EX 安装文件夹下的 PRO-SDK\Vc\Public 中。本例指定 “..\Public\ProEasy.lib”。



7 若要使用 Pro-Server EX 提供的读取 / 写入函数，必须包含一个头文件 (ProEasy.h)。点击 Microsoft Visual C++ 的 [Work Space] 窗口中的 [FileView] 选项卡，双击 “SampleDig.cpp” 文件。
本例在 “SampleDig.cpp” 文件中使用读取 / 写入函数。



8 在 SampleDig.cpp 文件中添加 #include "..\Public\ProEasy.h"。函数 (读取 / 写入函数) 声明步骤至此结束。



上述 1 到 8 步对读取和写入应用程序均适用。

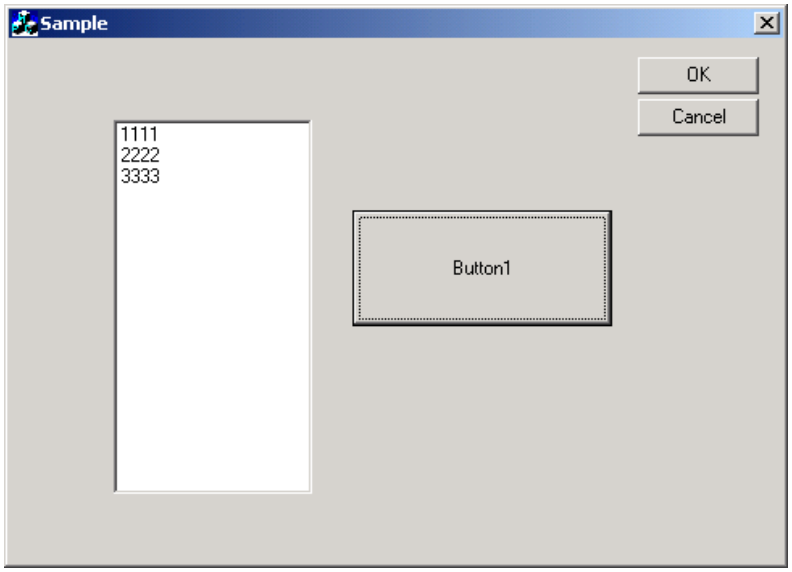
下述步骤则因应用程序是用于读取还是写入而不同，因此分别进行描述。

创建 “读取” 应用程序，请参阅 9 到 30 步。

创建 “写入” 应用程序，请参阅 31 到 47 步。

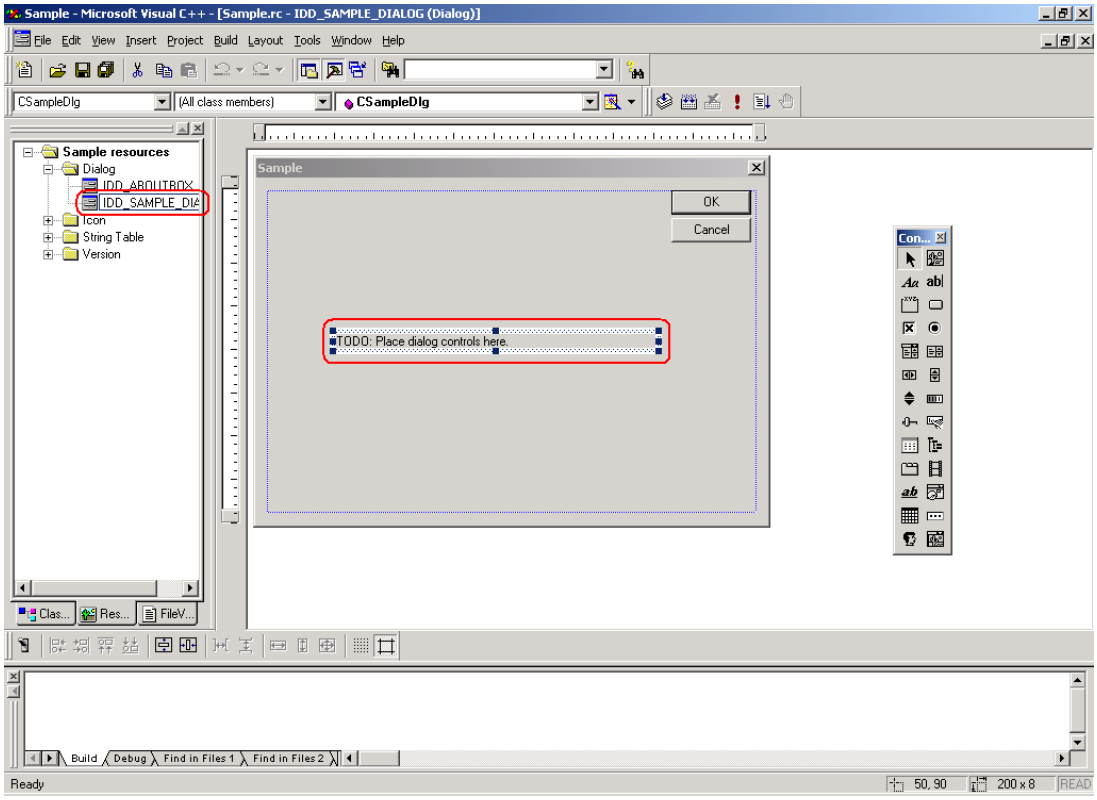
创建 “读取”应用程序

本节介绍创建下述应用程序的过程：点击 [Button1] 读取并显示三个数据 (16 位有符号数据)。

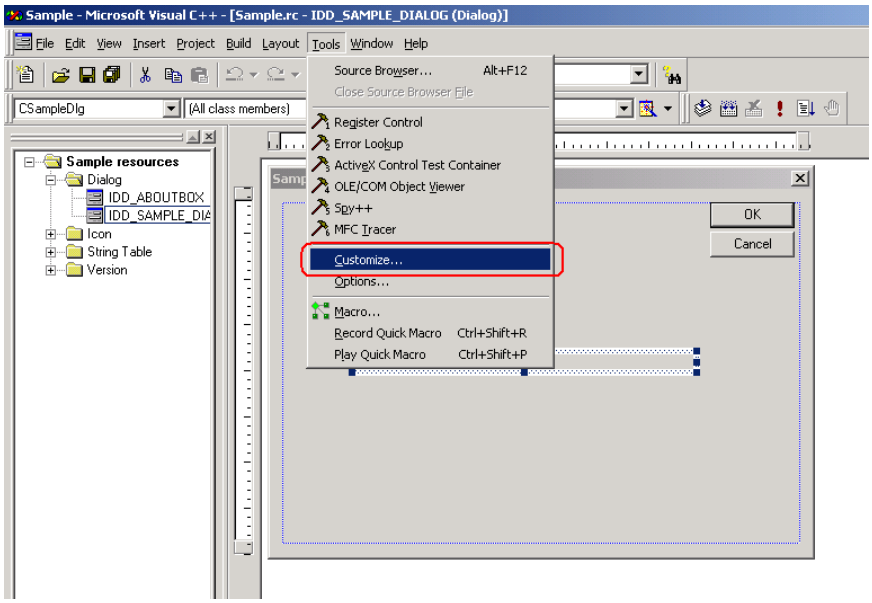


9 点击 Microsoft Visual C++ 的 [Work Space] 窗口中的 [ResourceView] 选项卡，双击 [IDD_SAMPLE_DIALOG]。

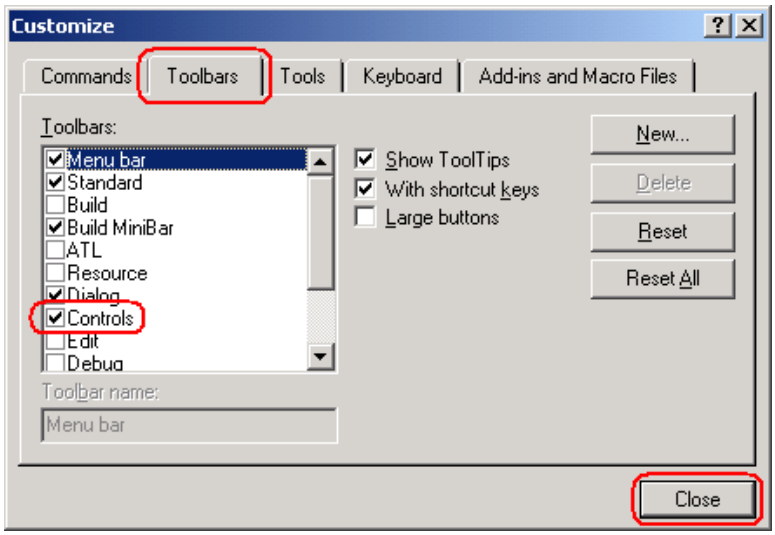
点击对话框中央的 [Static Text]，将其删除。



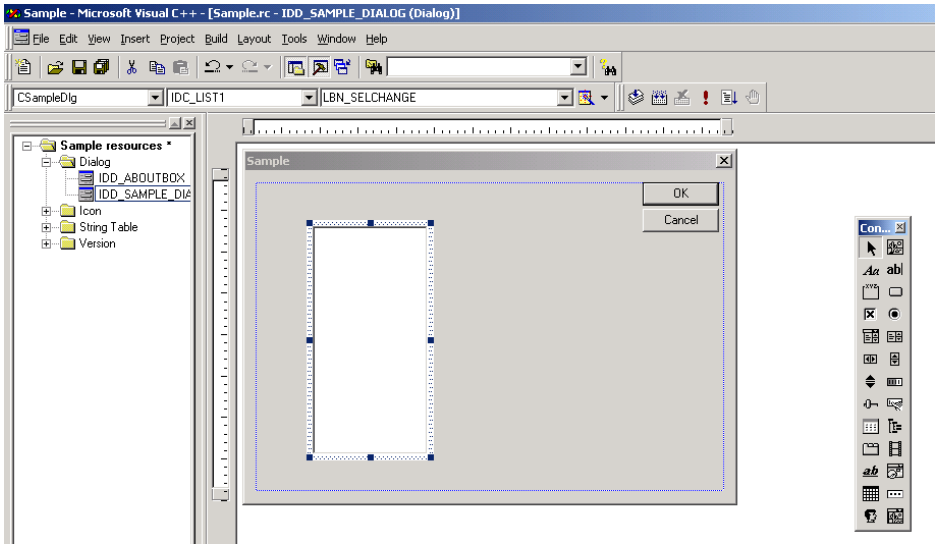
10 从 Microsoft Visual C++ 菜单的 [Tools] 中选择 [Customize]。



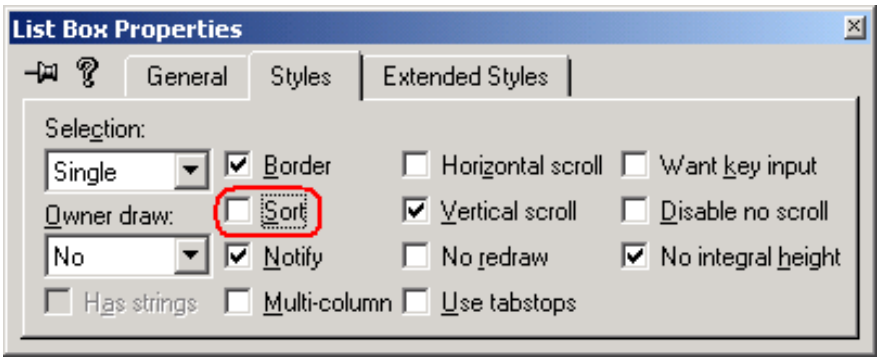
11 勾选 [Toolbars] 选项卡中的 [Controls] 复选框，点击 [Close] 按钮。



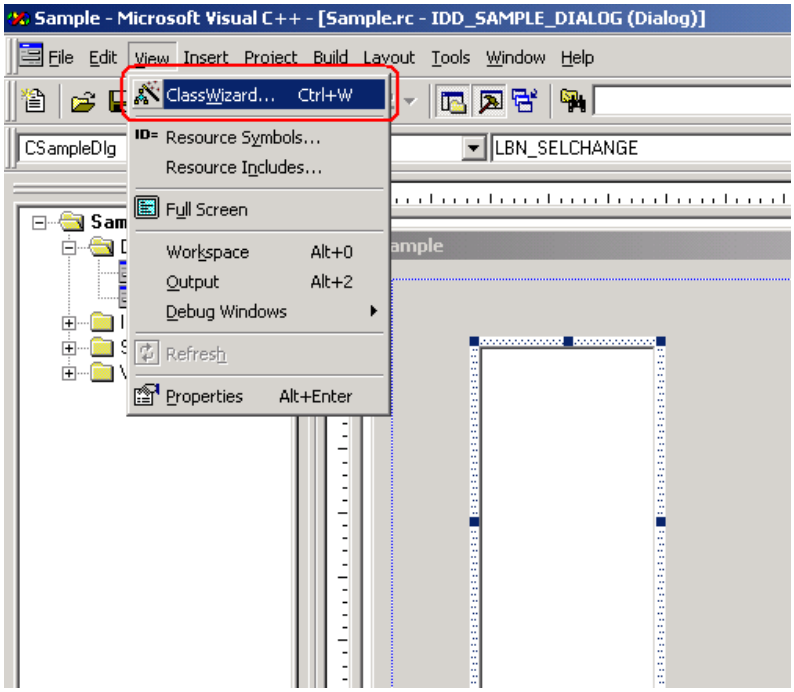
12 选择 [ListBox] 并将它粘贴到对话框。



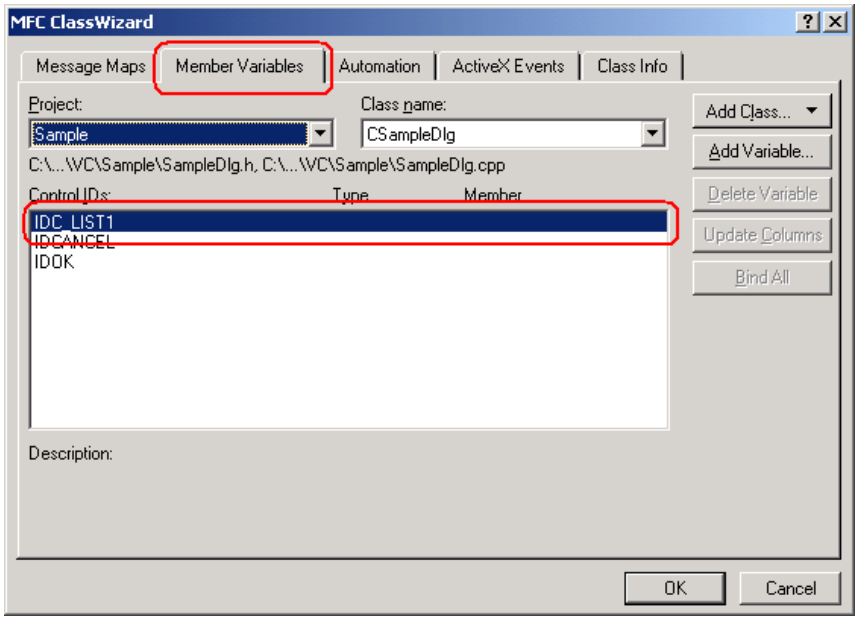
13 右击粘贴的 [ListBox]，选择 [Property]。弹出 [List Box Propertis] 对话框。取消勾选 [Sort] 复选框。



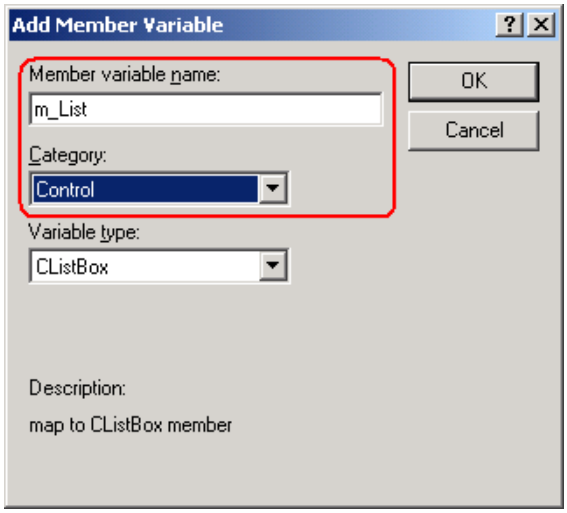
14 从 Microsoft Visual C++ 菜单的 [View] 中选择 [ClassWizard]。



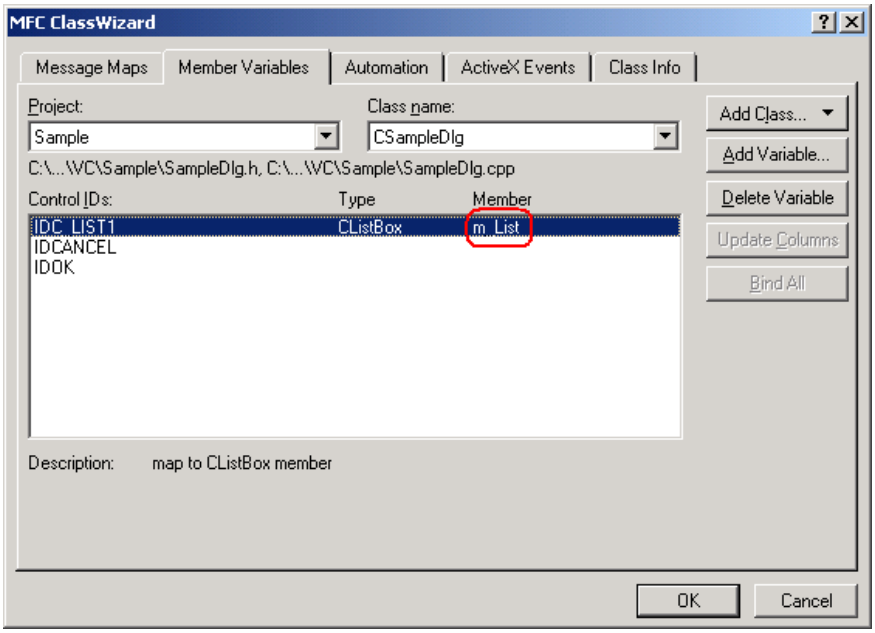
15 选择 [Member Variables] 选项卡，在 [Control IDs] 处选择 “IDC_LIST1”。



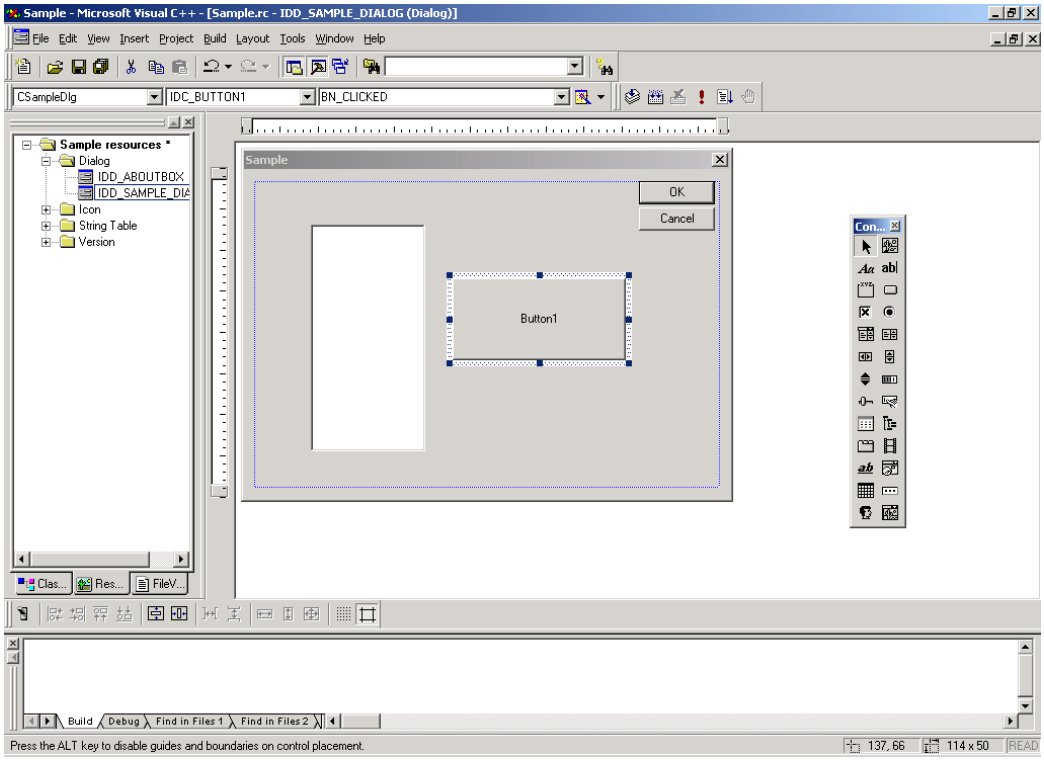
16 点击 [Add Variable]，在 [Member variable name] 处输入 “m_List”。在 [Category] 处选择 “Control”，点击 [OK] 按钮。



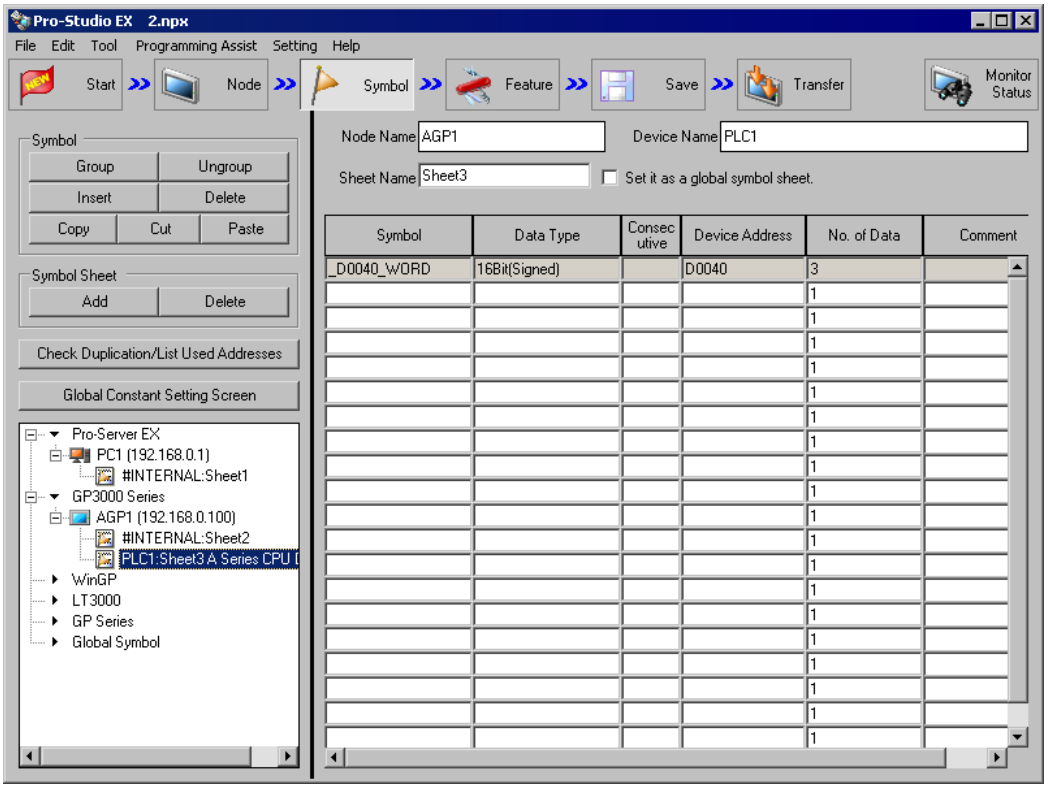
17 确认已添加成员变量后，点击 [OK] 按钮。



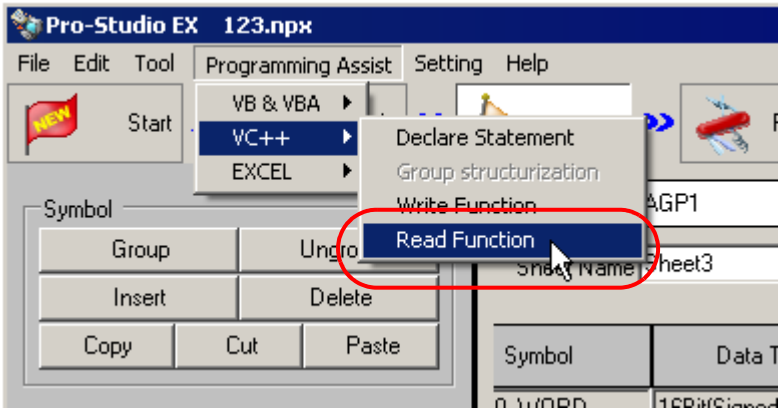
18 选择 [Button] 并将它粘贴到对话框。



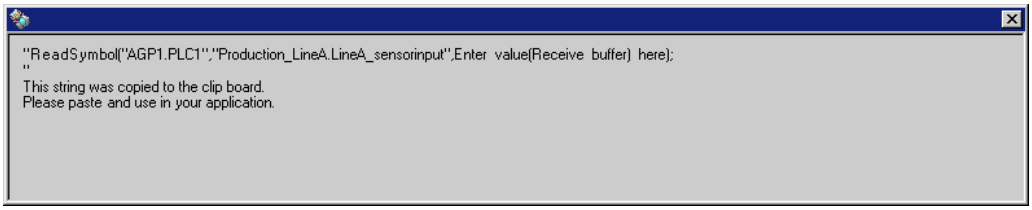
19 从 Pro-Studio EX 中注册的符号名称中选择一个目标符号。（选择要读取的首地址对应的符号。）



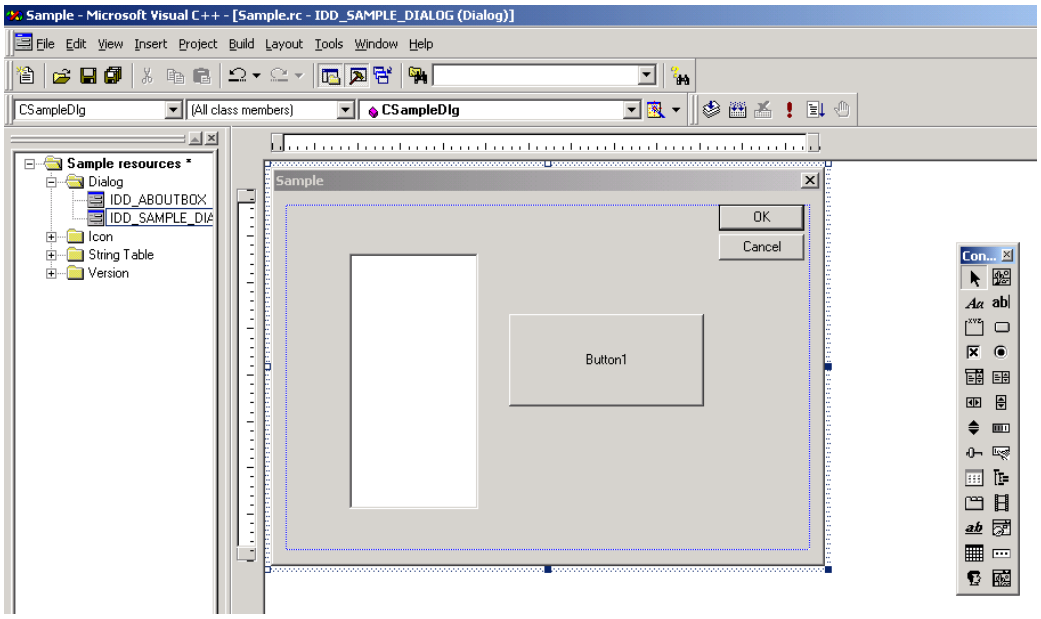
20 从菜单上选择 [Programming Assist] - [VC++] - [Read Function]。



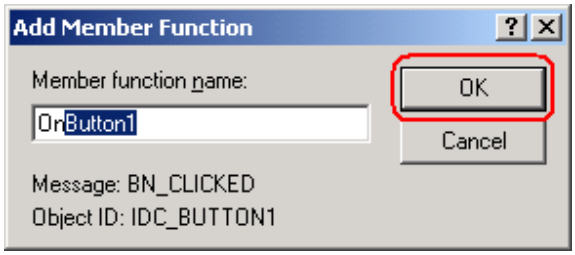
读取函数被复制到剪贴板上。



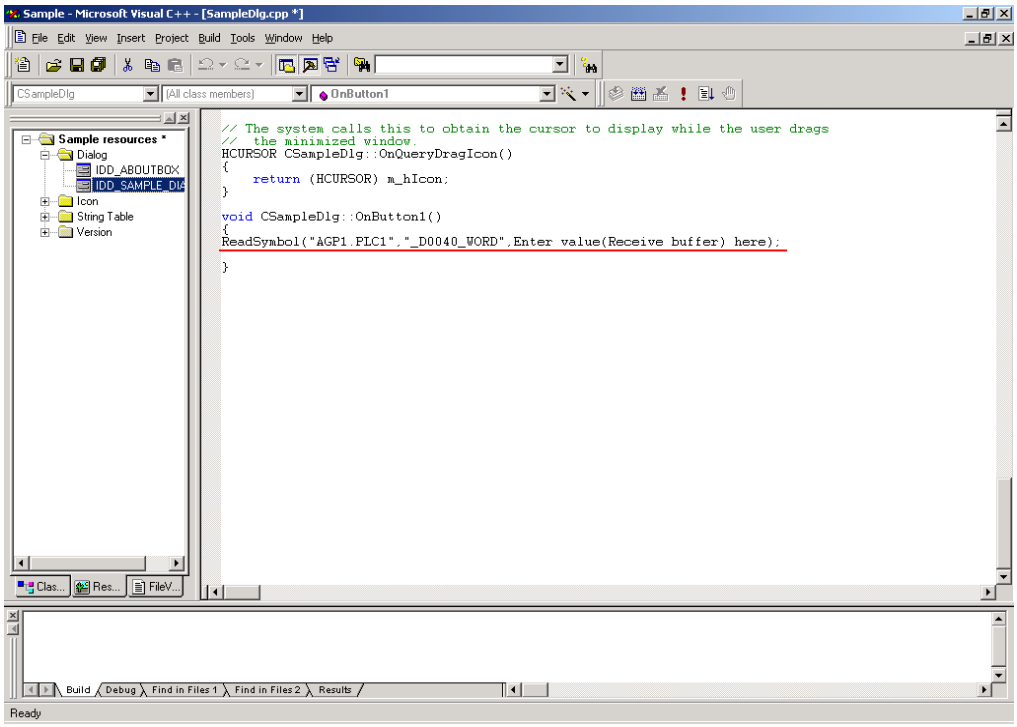
21 在 Microsoft Visual C++ 中双击已粘贴到对话框的 [Button1]。



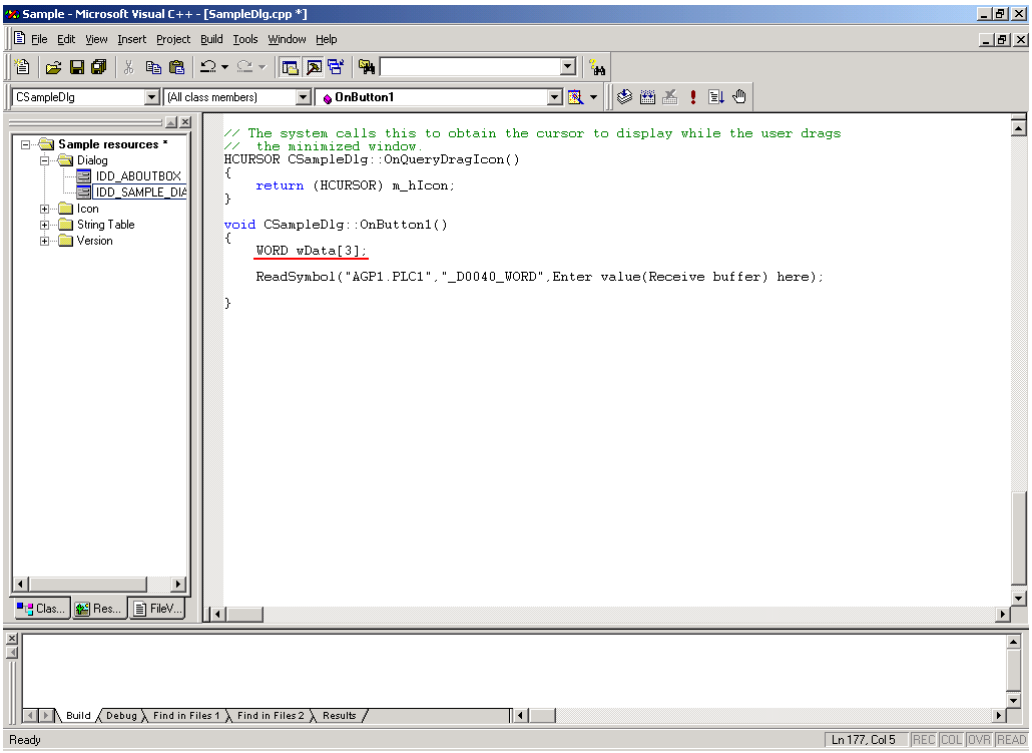
22 点击 [OK] 按钮。



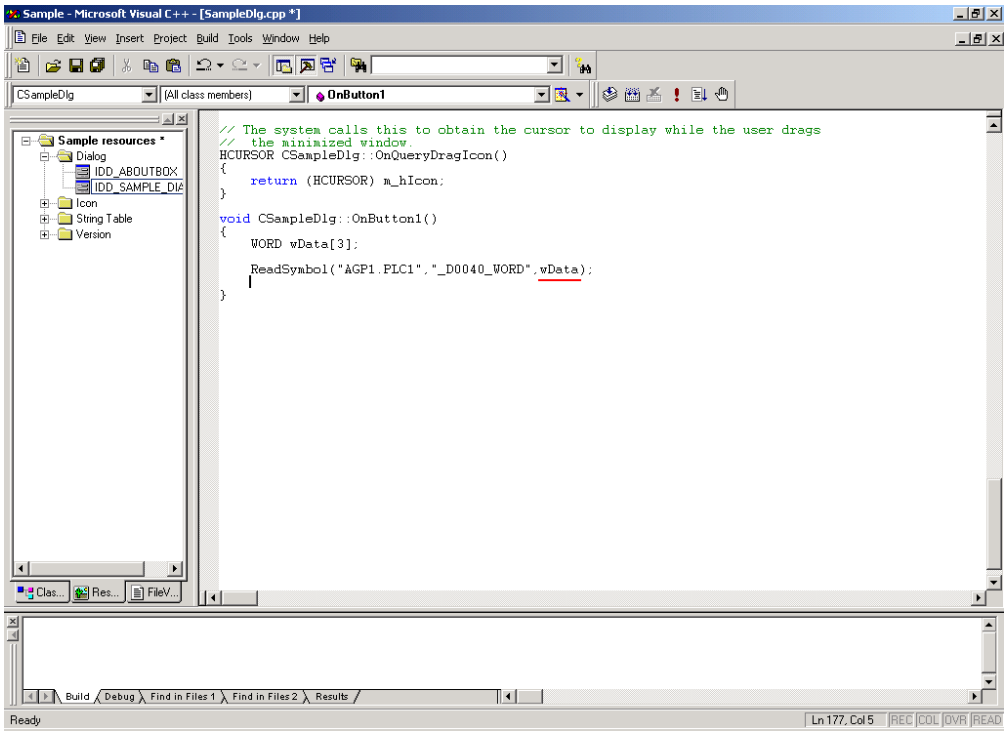
23 将剪贴板上的数据 (读取函数) 粘贴到 OnButton1 成员函数中。



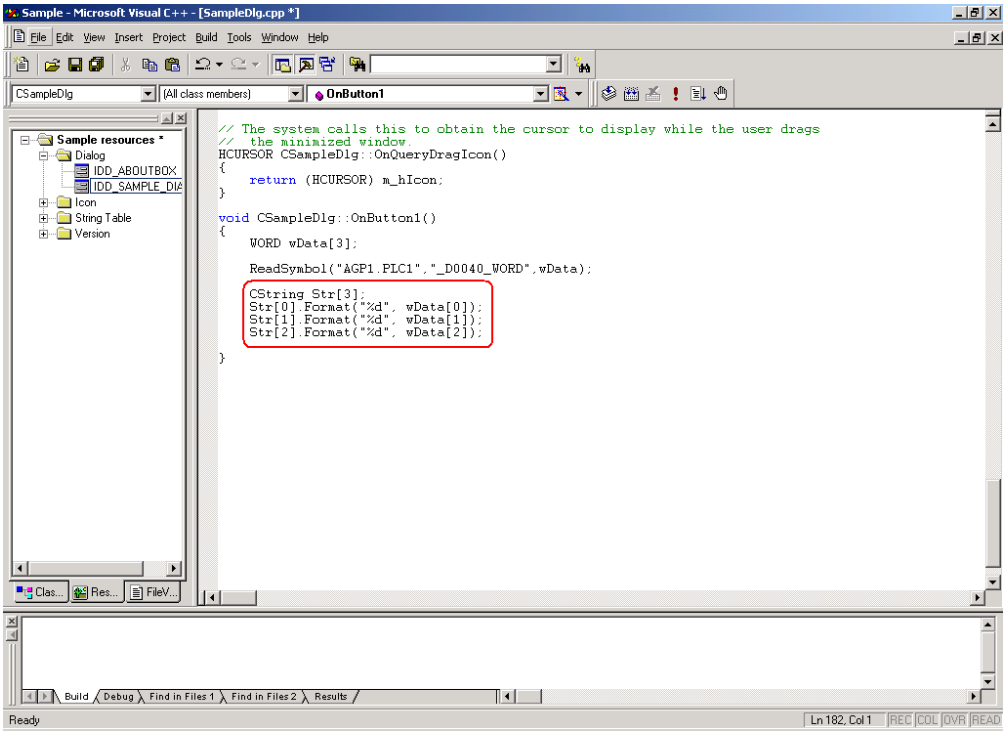
24 声明保存读取数据的区域 (数组)。



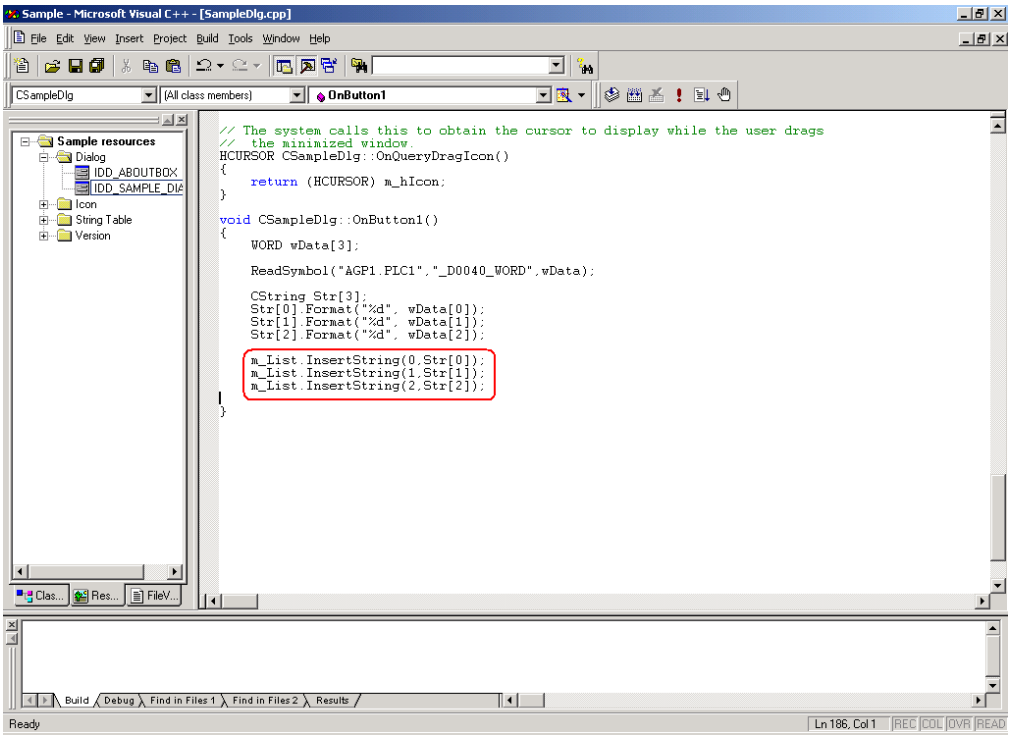
25 指定保存读取数据的首个区域 (wData)。



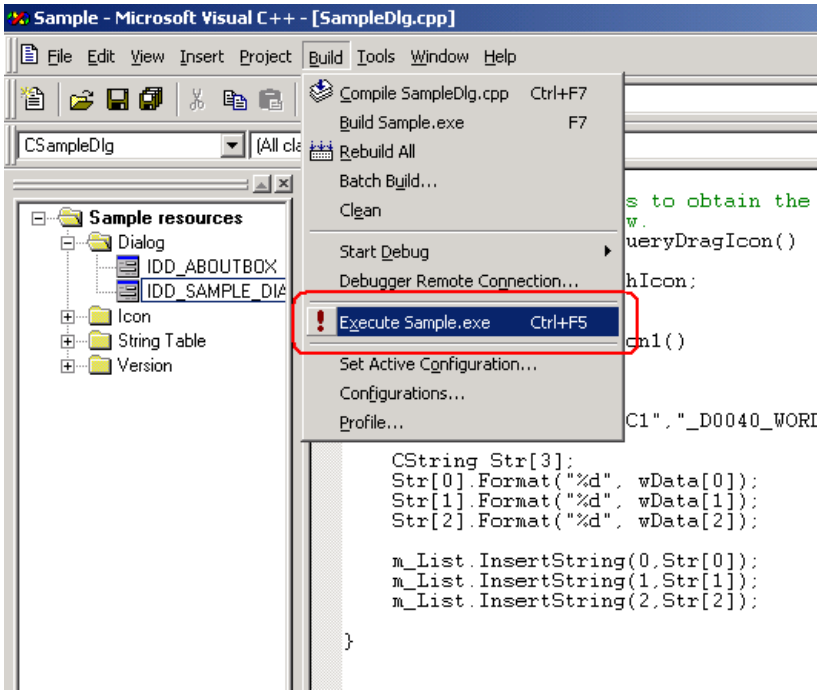
26 为能在列表框中显示三个读取的数据 (wData(0)、wData(1) 和 wData(2))，将数据转换成 Cstring 型的字符串数据。



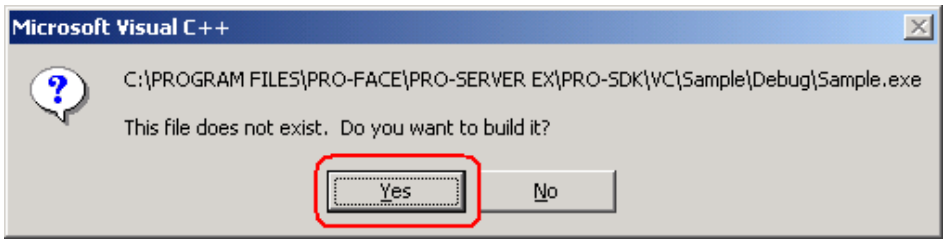
27 列表框 (m_List) 按顺序显示读取的数据 (已转换成字符串数据)。



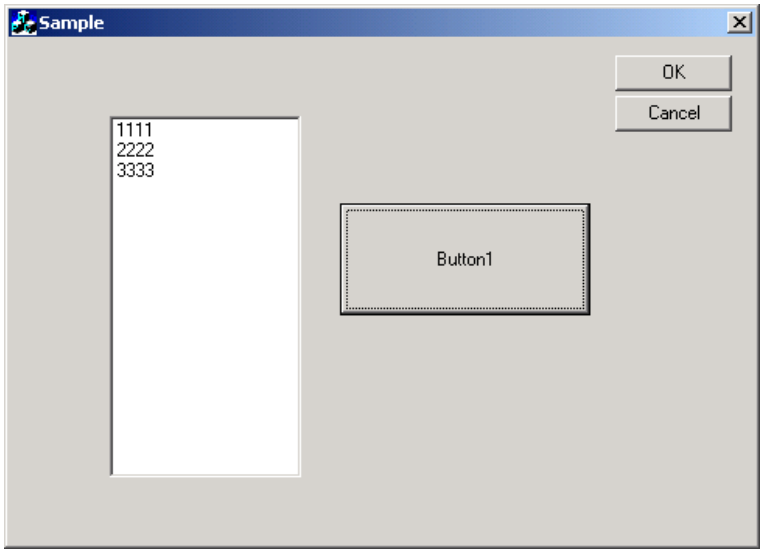
28 从 Microsoft Visual C++ 菜单的 [Build] 中选择 [Execute Sample.exe]。



29 点击 [Yes] 按钮。



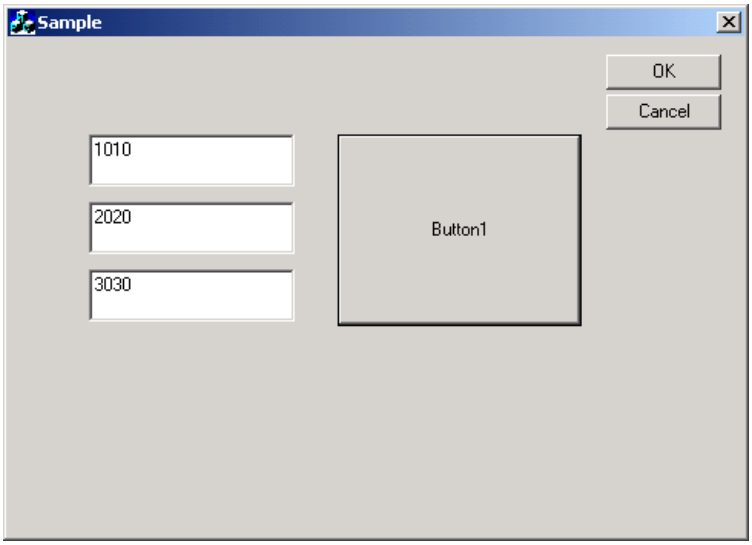
30 点击 [Button1]。然后，列表框从符号 “_D0040_WORD” 开始显示三个数据。



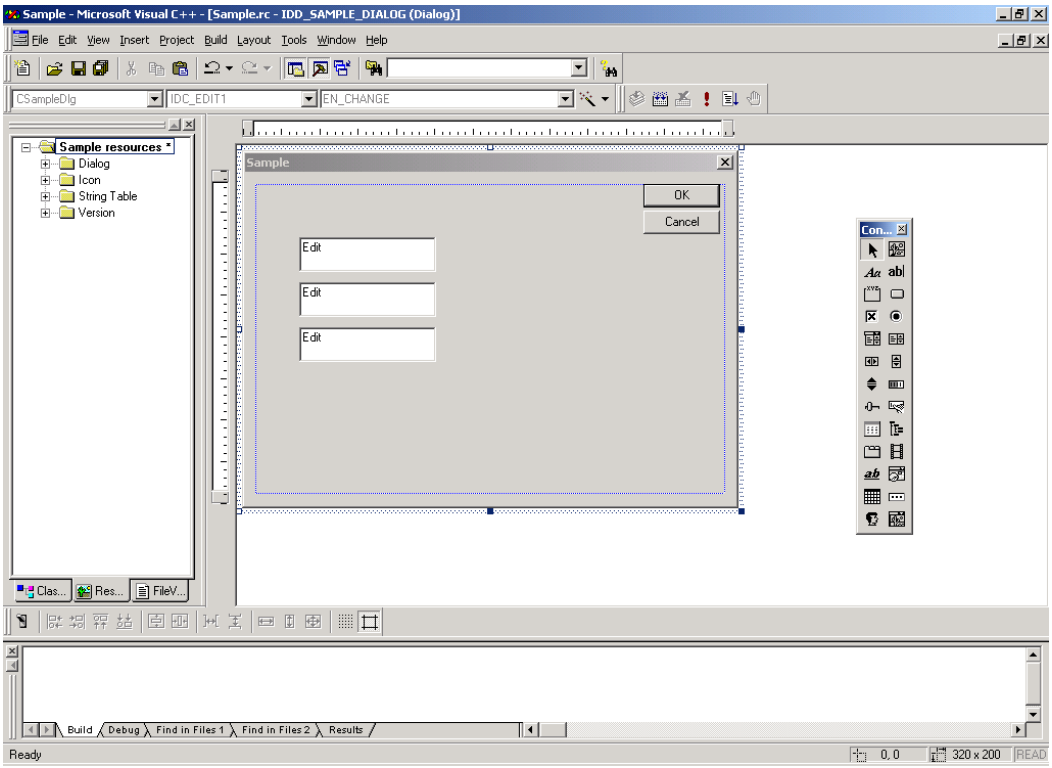
创建 “写入” 应用程序

本节介绍创建下述应用程序的过程：点击 [Button1] 写入输入的三个数据。

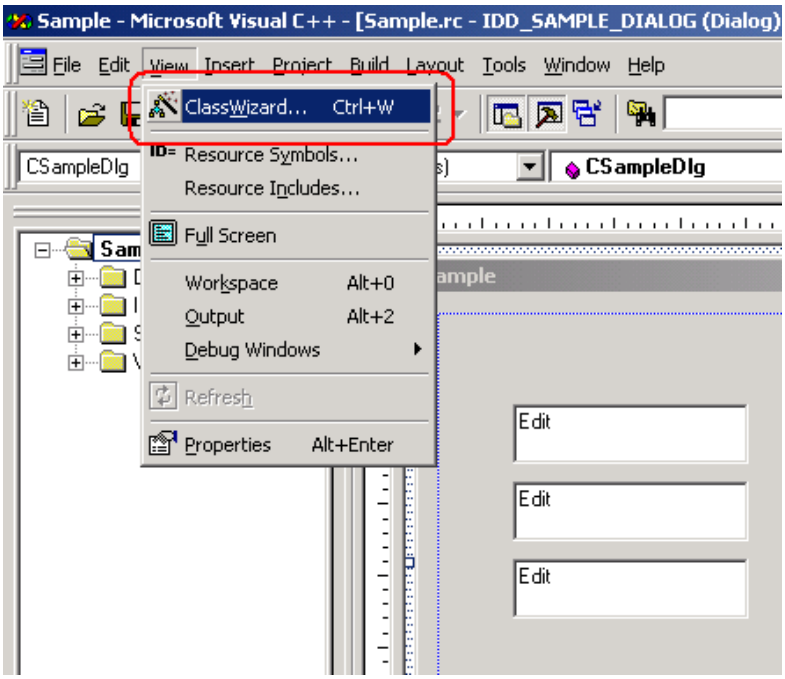
第 9 到 11 步与创建 “读取” 应用程序相同。



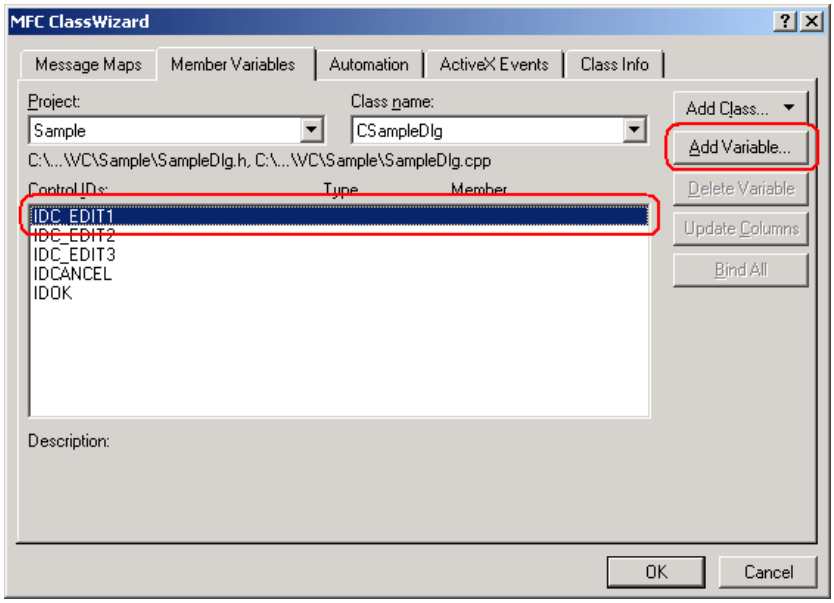
31 选择 [EditBox] 并将它粘贴到 [Dialog]。粘贴三个 [EditBox]。



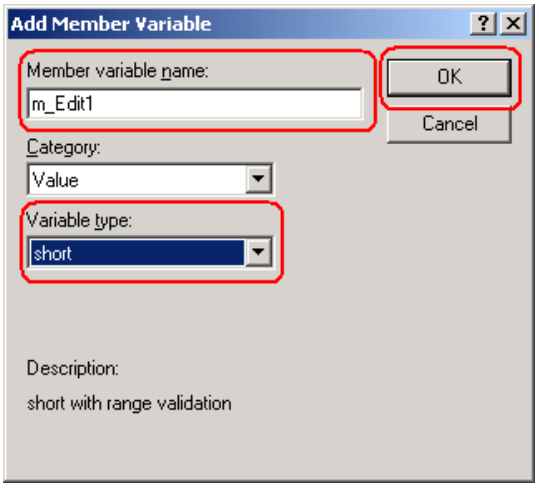
32 从 Microsoft Visual C++ 菜单的 [View] 中选择 [ClassWizard]。



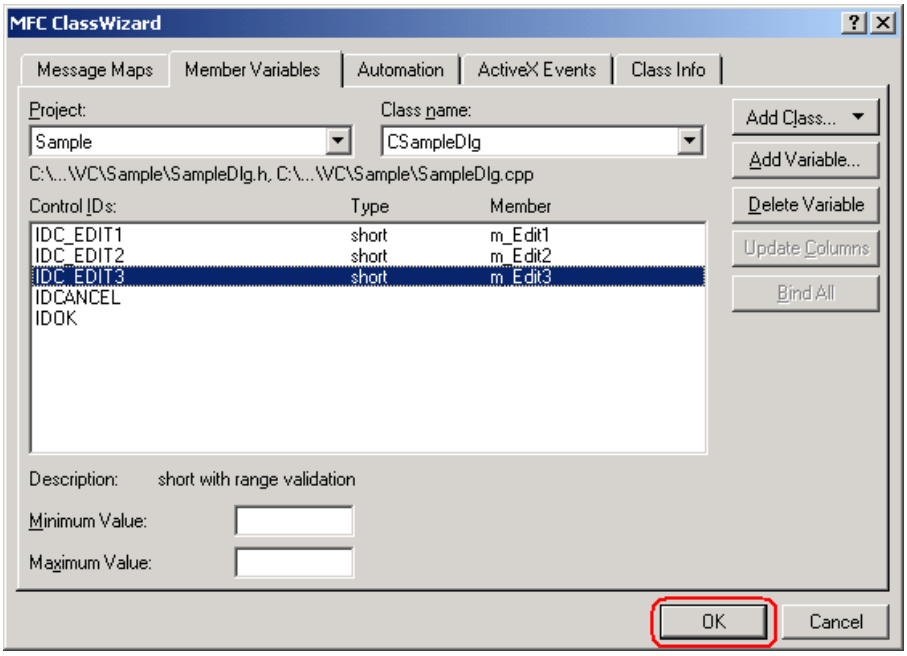
33 在 [Member Variables] 选项卡的 [Control IDs] 处选择 “IDC_EDIT1”，然后点击 [Add Variable] 按钮。



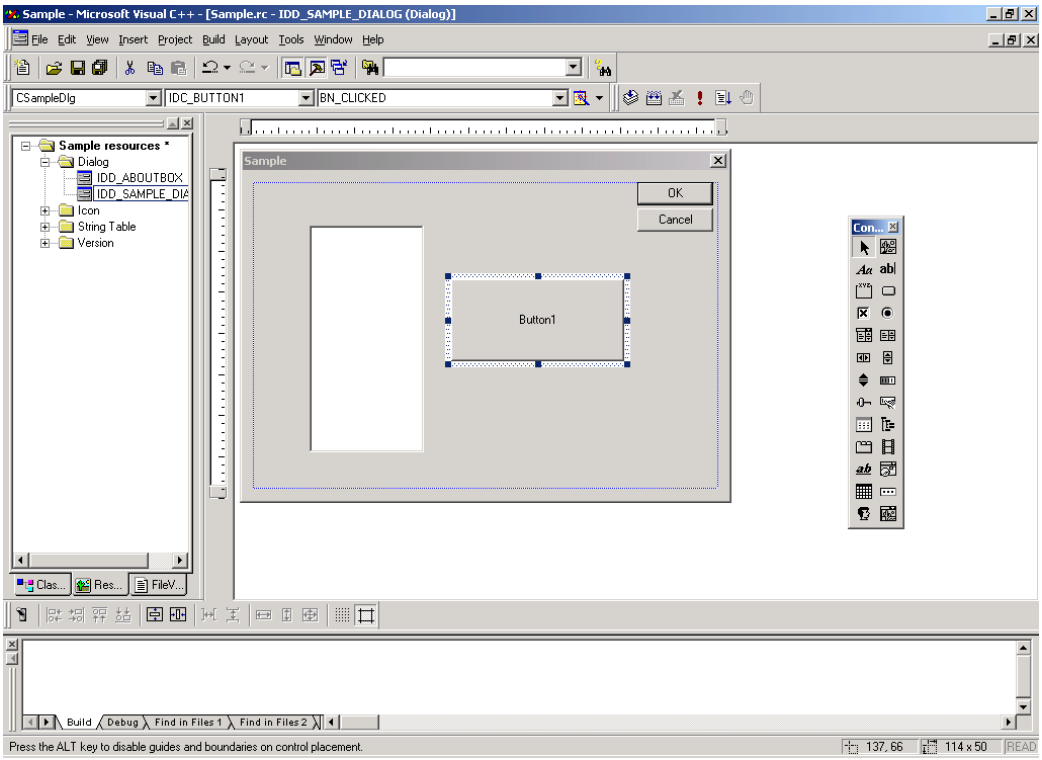
34 在 [Member Variable] 处输入 m_Edit1， [Variable type] 选择 “short”。然后点击 [OK] 按钮。
对其余两个 [Edit Box] 重复步骤 33 和 34，成员变量分别指定 “m_Edit2” 和 “m_Edit3”。



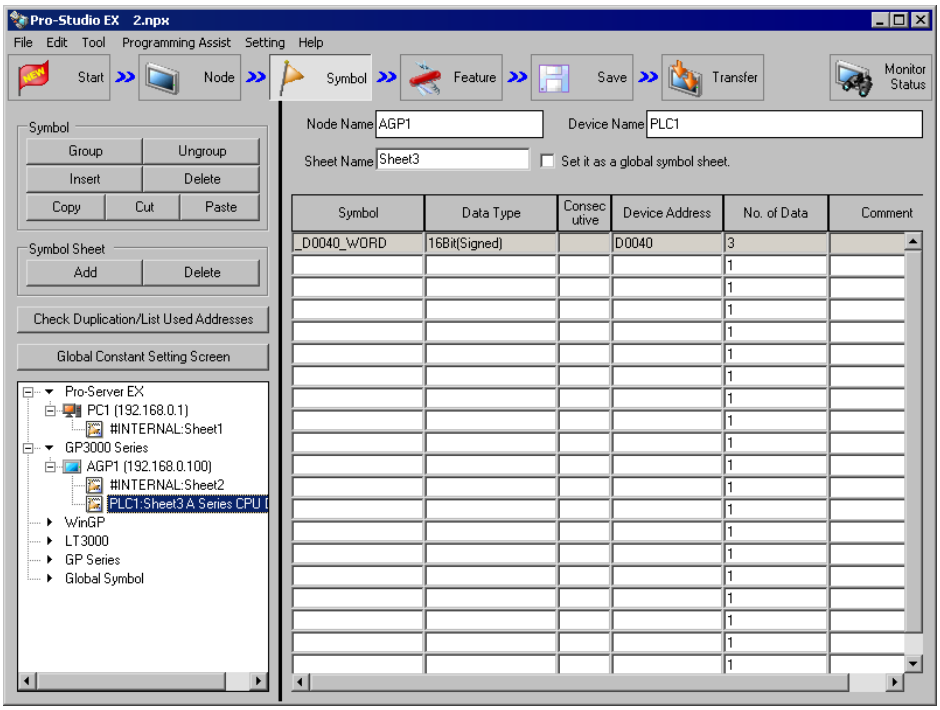
35 点击 [OK] 按钮。



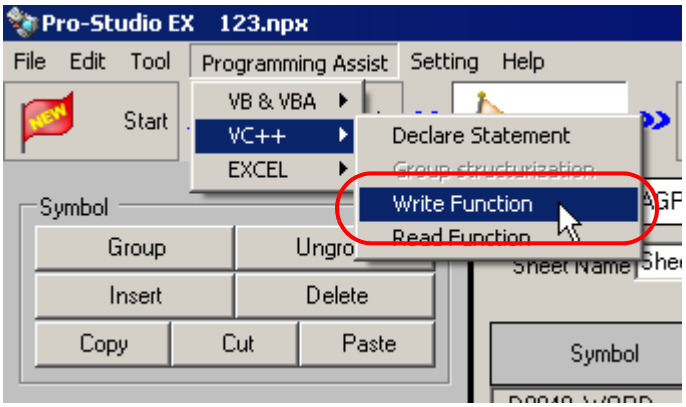
36 选择 [Button] 并将它粘贴到 [Dialog]。



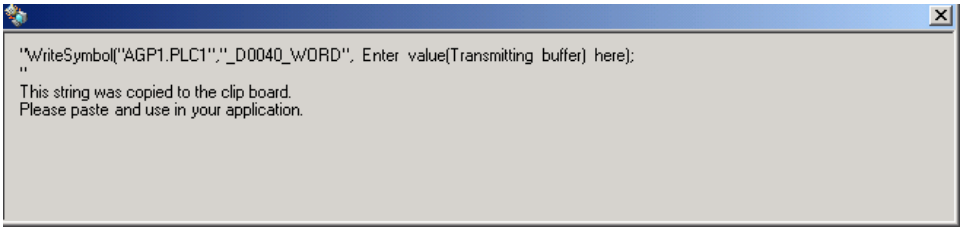
37 从 Pro-Studio EX 中注册的符号名称中选择一个目标符号。（选择要写入的首地址对应的符号。）



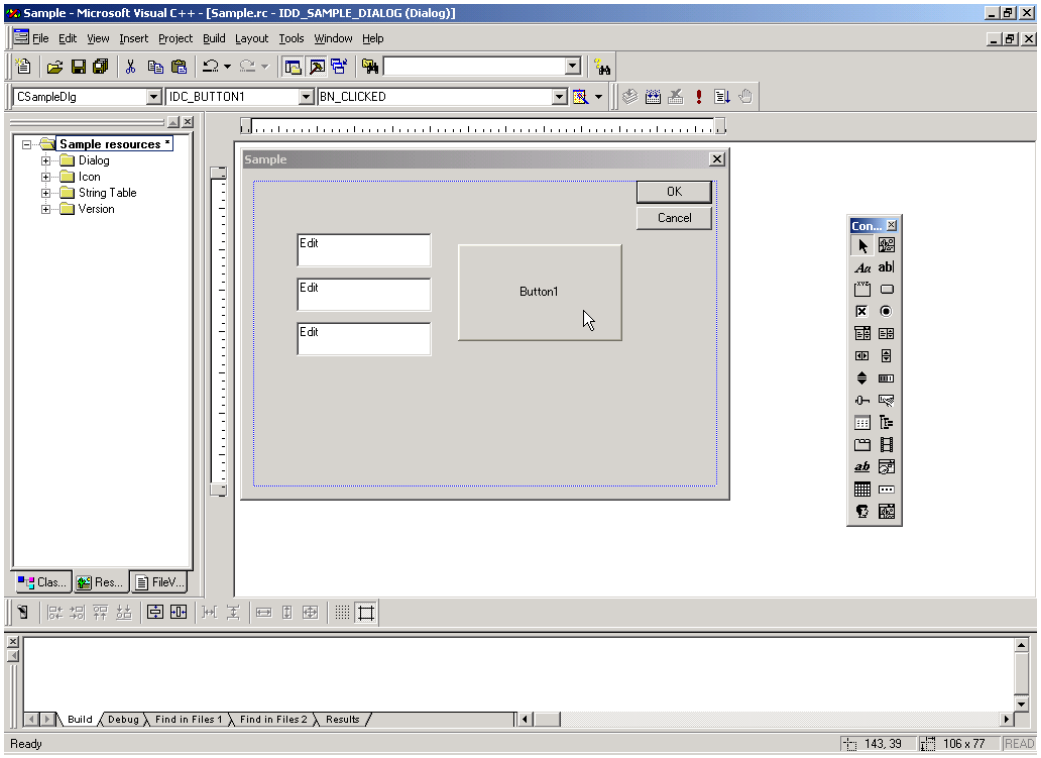
38 从菜单上选择 [Programming Assist] - [VC++] - [Write Function]。



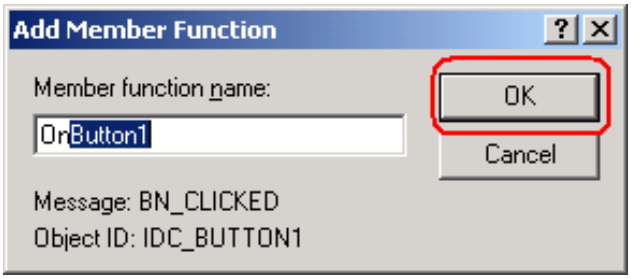
写入函数被复制到剪贴板上。



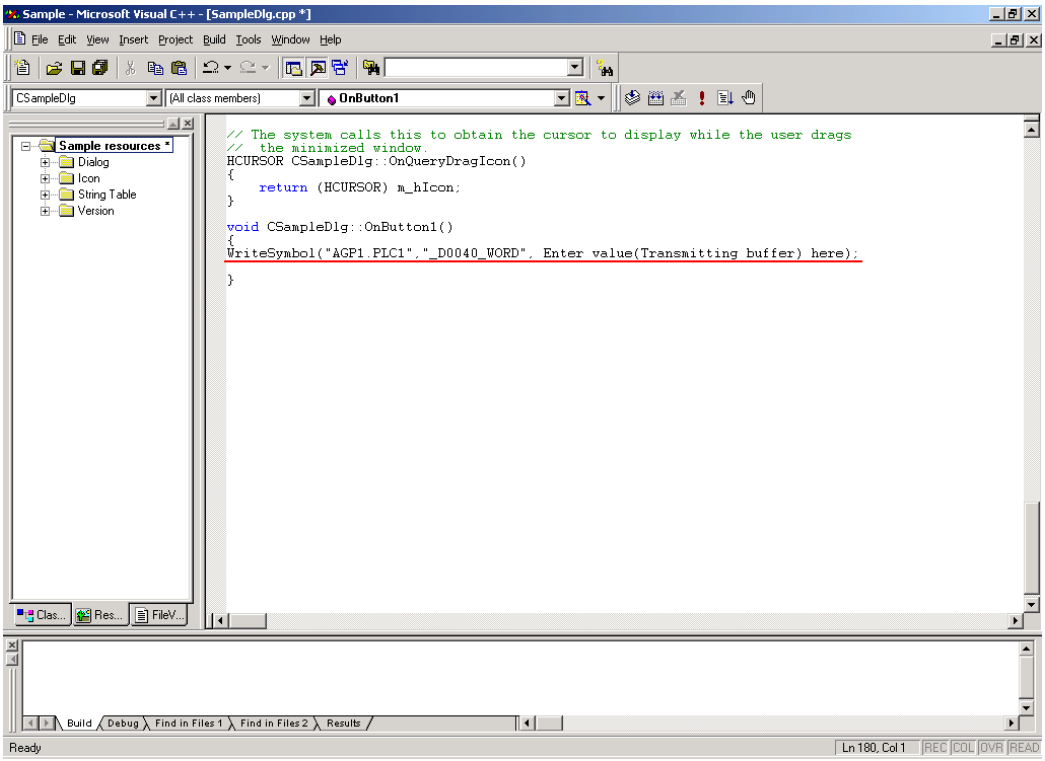
39 在 Microsoft Visual C++ 中双击已粘贴到对话框的 [Button1]。



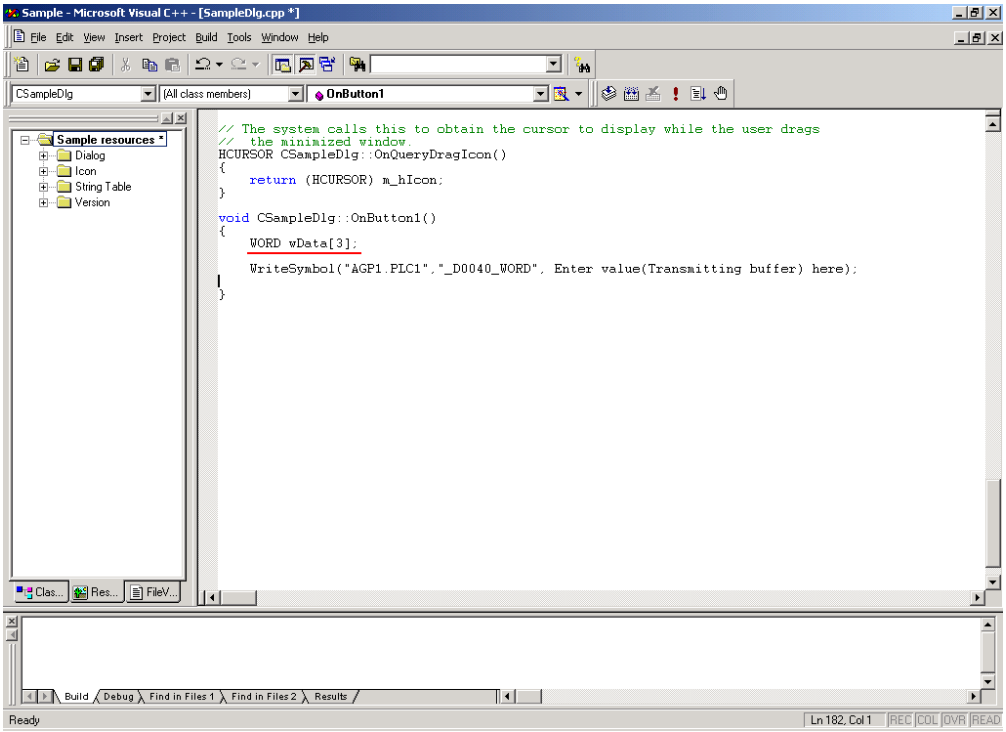
40 点击 [OK] 按钮。



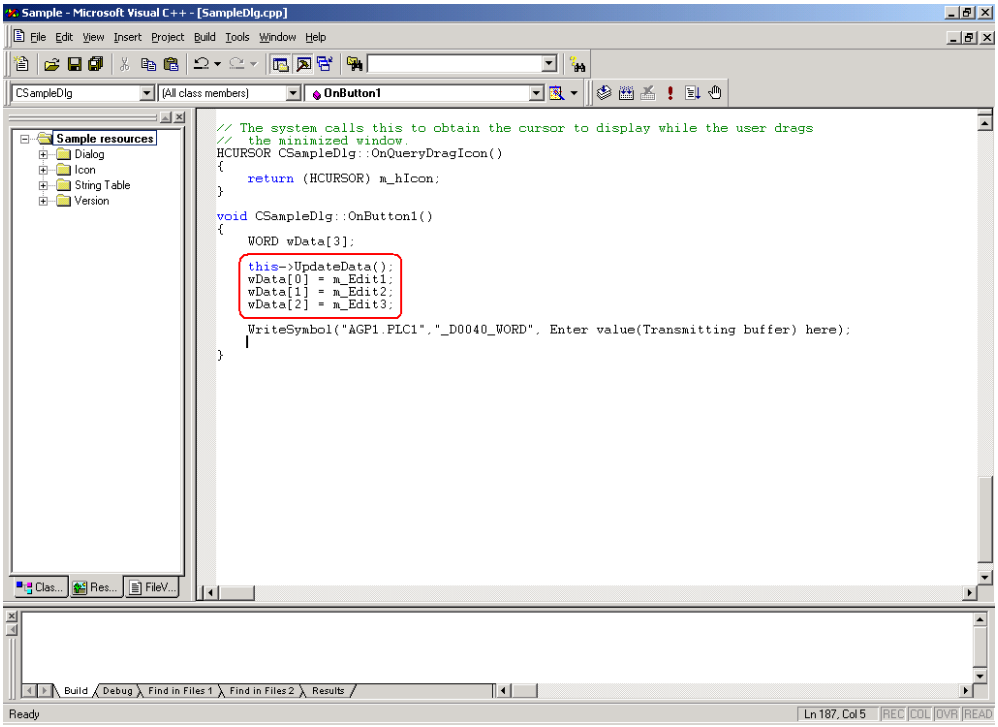
41 将剪贴板上的数据 (写入函数) 粘贴到 OnButton1 成员函数中。



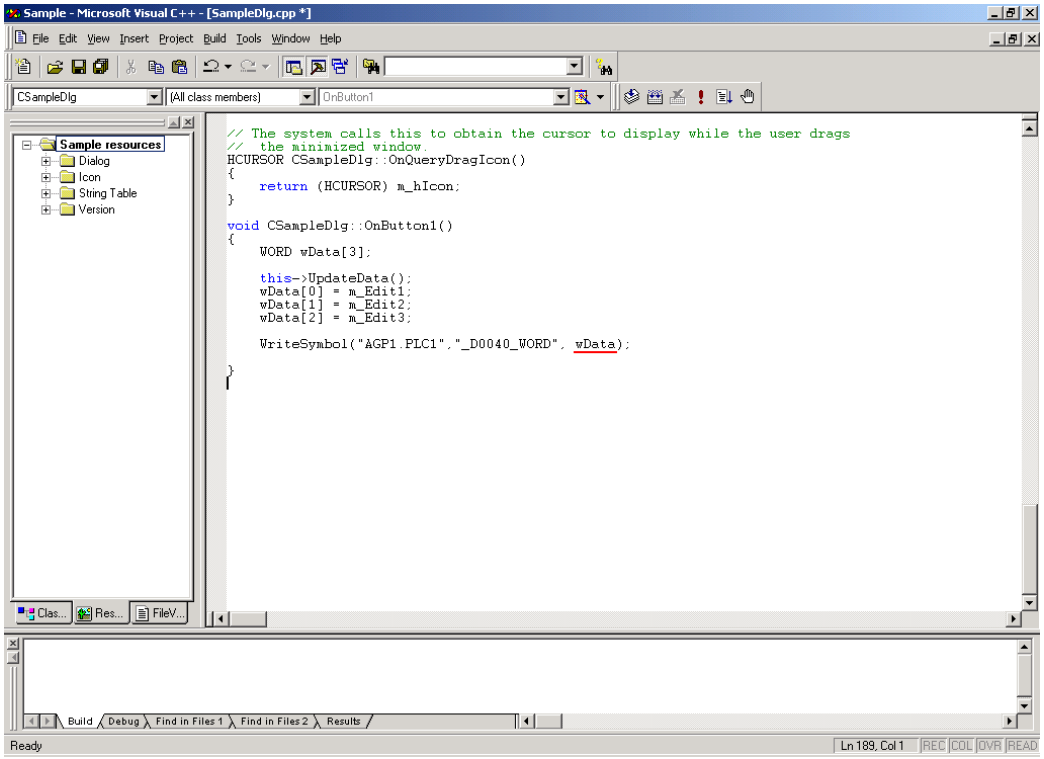
42 声明保存写入数据的区域 (数组)。如需写入三个以上的数据，请指定三个以上的数组元素。



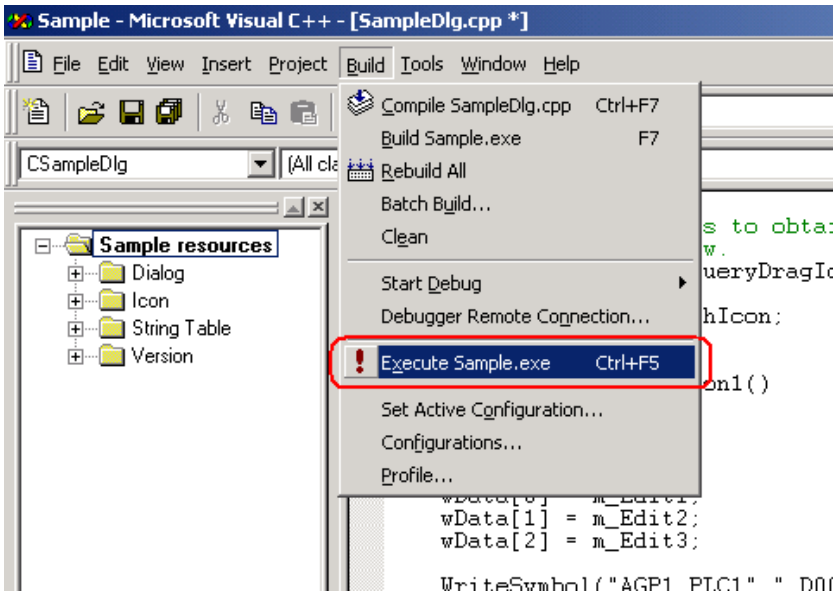
43 将输入 [Edit Box] 的数据 (三个) 赋值给数组。



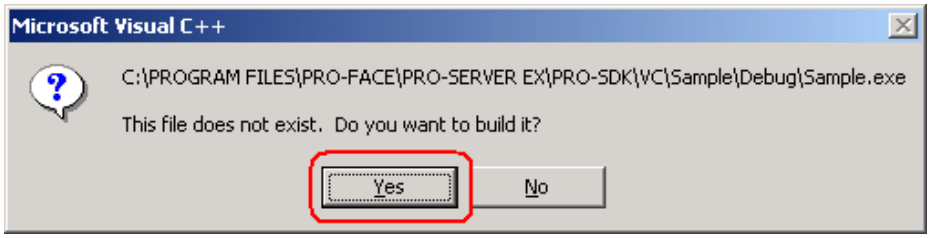
44 指定赋值写入数据的首个区域 (wData)。



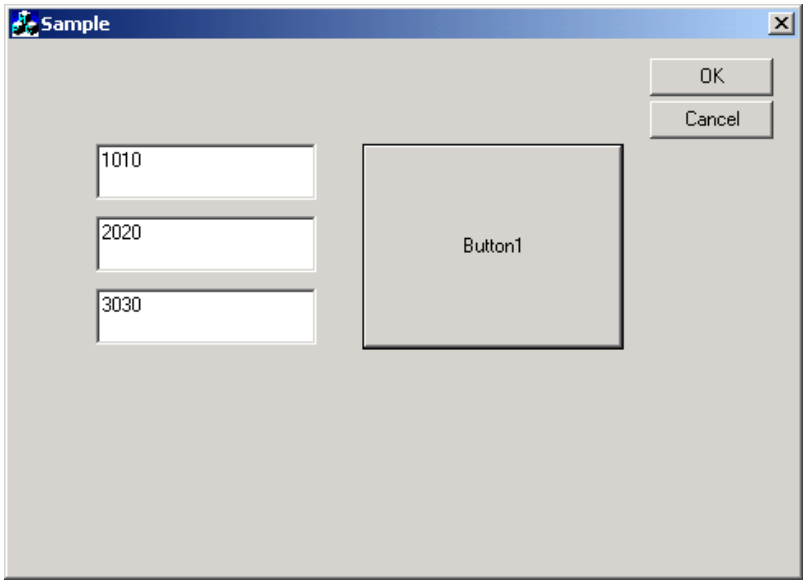
45 从 Microsoft Visual C++ 菜单的 [Build] 中选择 [Execute Sample.exe]。



46 点击 [Yes] 按钮。

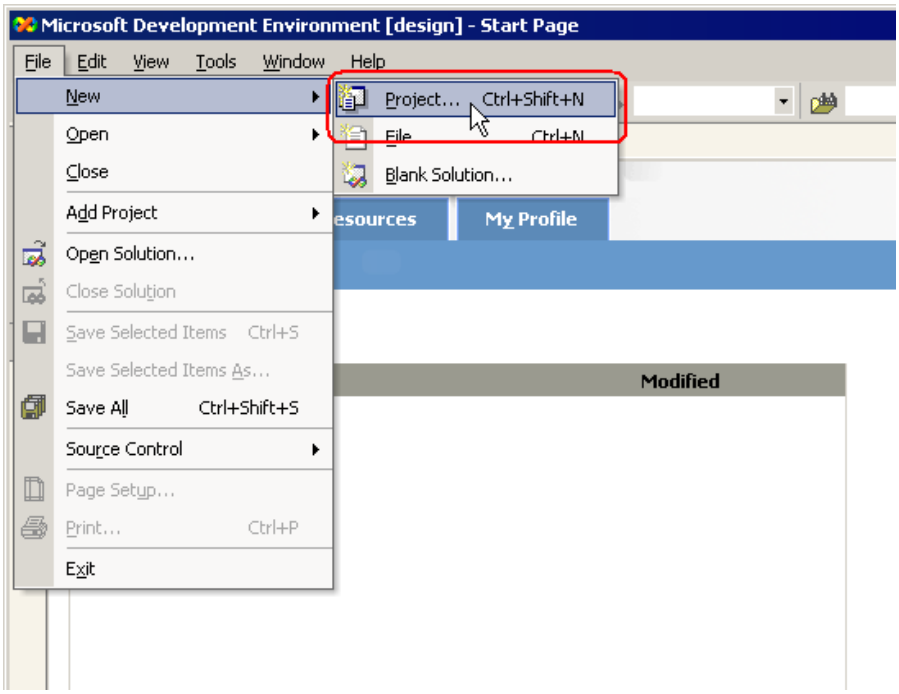


47 在各 [Edit Box] 中输入三个值后，点击 [Button1]。然后， Pro-Server EX 将三个数据写入从符号“_D0040_WORD”开始的三个寄存器中。

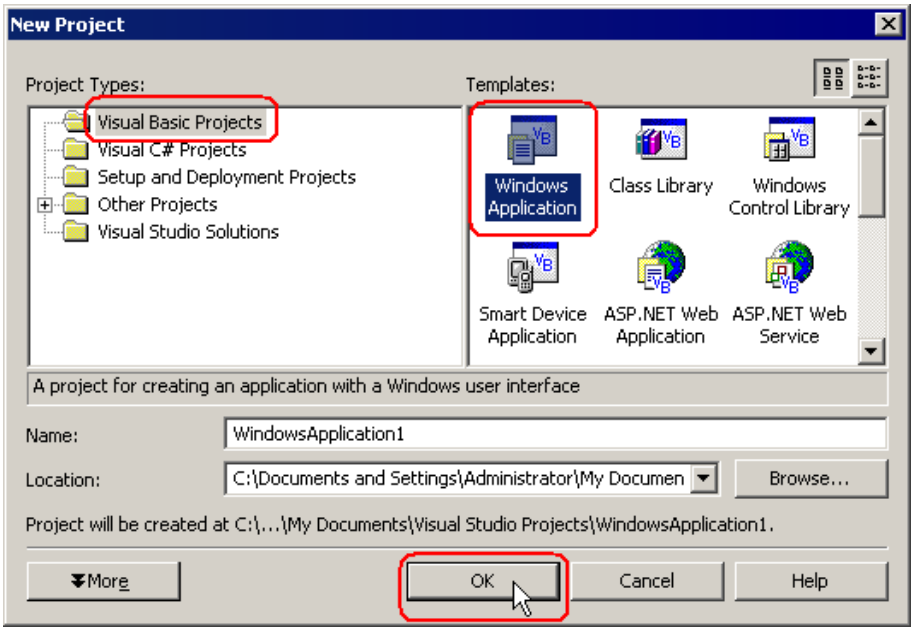


27.11.3 VB .NET 支持的函数

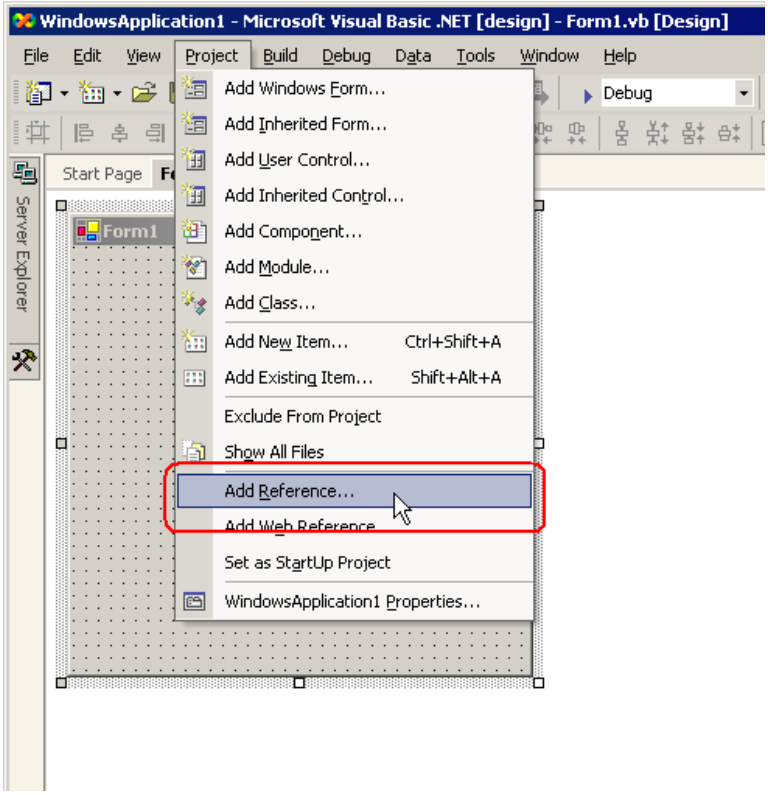
1 启动 Microsoft Visual Studio .NET 2003(或以上版本)，从 [File] 菜单中选择 [New]-[Project]。



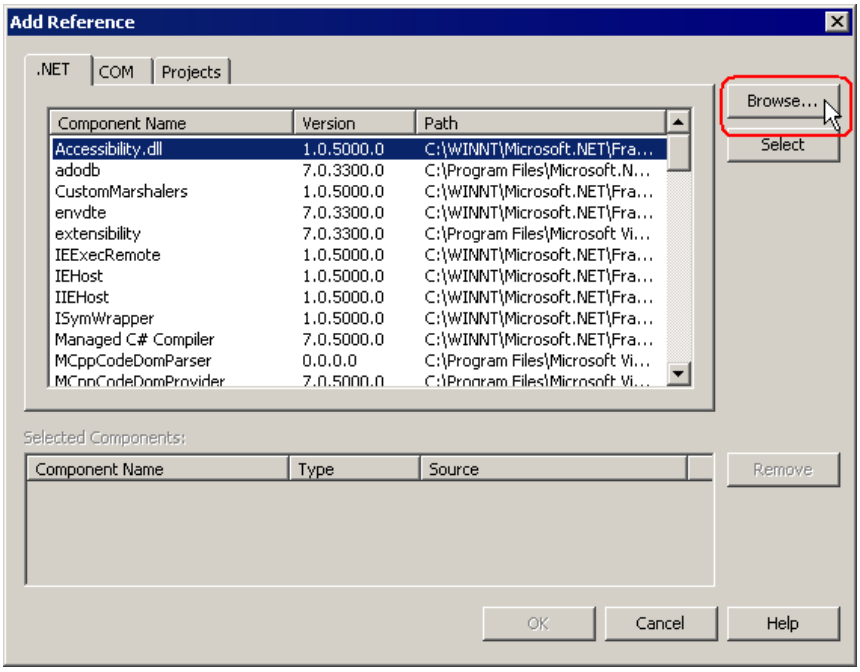
2 从 [Project Types:] 中选择 [Visual Basic Projects]，从 [Templates:] 中选择 [Windows Application]，点击 [OK] 按钮。



3 从 [Project] 菜单中选择 [Add Reference]。



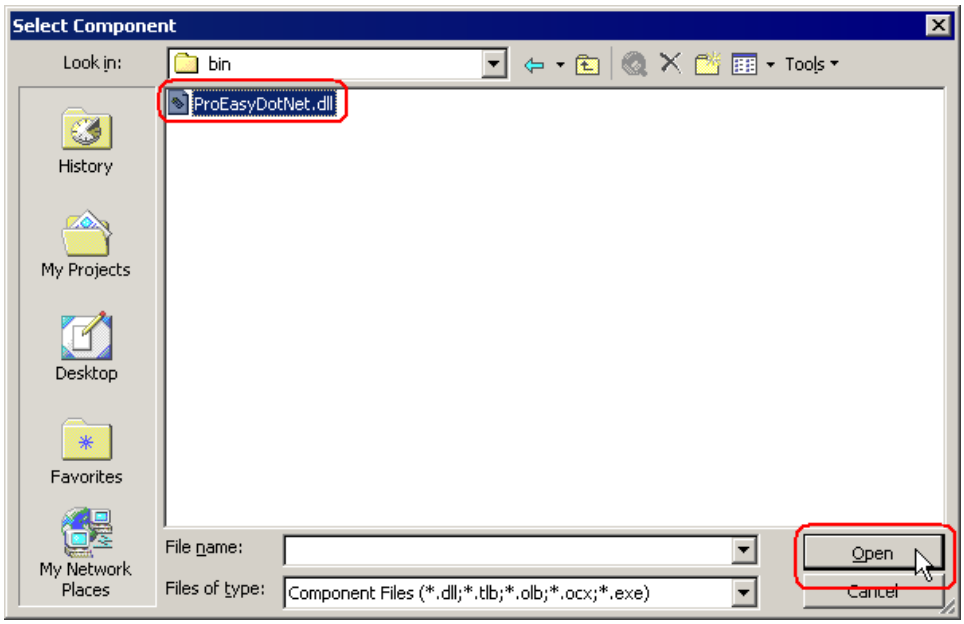
4 点击 [Browse] 按钮。



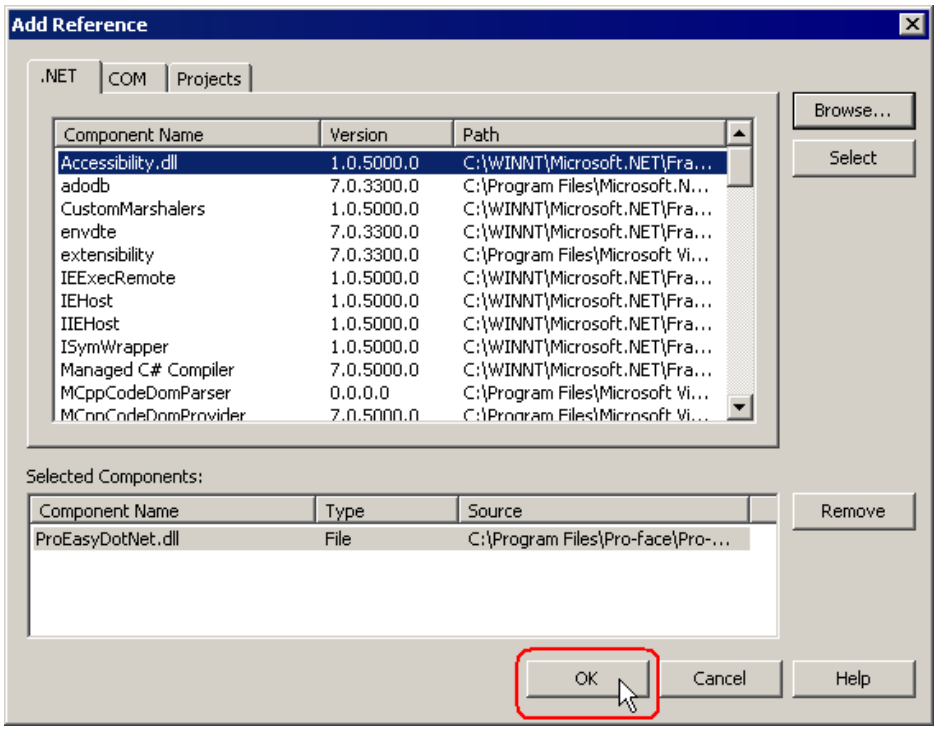
5 指定 ProEasyDotNet.dll 的安装目录，点击 [Open] 按钮。（采用标准安装时，文件保存目录为“C:\Program Files\Pro-face\Pro-Server EX\PRO-SDK\DotNet\bin\ProEazyDotNet.dll”。）

注 释

- ProEasyDotNet 对 Microsoft .NET Framework 1.1 的支持
  - Windows Vista 或以上
    C:\Pro-face\Pro-Server EX\PRO-SDK\DotNet\bin\ProEasyDotNet.dll
  - Windows 2000 / XP / Server 2003
    C:\Program Files\Pro-face\Pro-Server EX\PRO-SDK\DotNet\bin\ProEasyDotNet.dll
- ProEasyDotNet 对 Microsoft .NET Framework 2.0 的支持
  - Windows Vista 或以上
    C:\Pro-face\Pro-Server EX\PRO-SDK\DotNet20\bin\ProEasyDotNet.dll
  - Windows 2000 / XP / Server 2003
    C:\Program Files\Pro-face\Pro-Server EX\PRO-SDK\DotNet20\bin\ProEasyDotNet.dll



6 点击 [OK] 按钮。



“ProEasyDotNet.dll”将被注册。

VB.NET 的运行环境设置至此结束。

上述 1 到 6 步对读取和写入应用程序均适用。

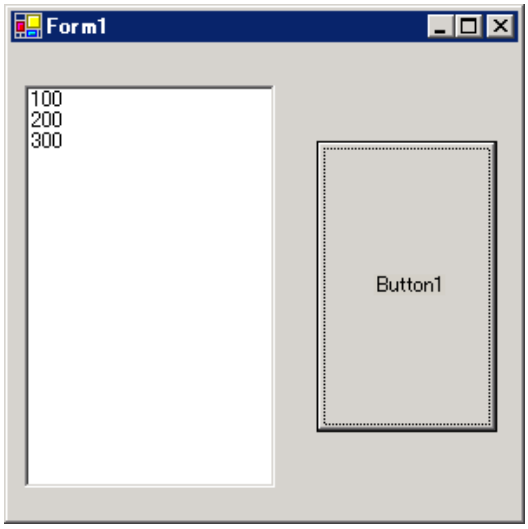
下述步骤则因应用程序是用于读取还是写入而不同，因此分别进行描述。

创建 “读取”应用程序，请参阅 7 到 19 步。

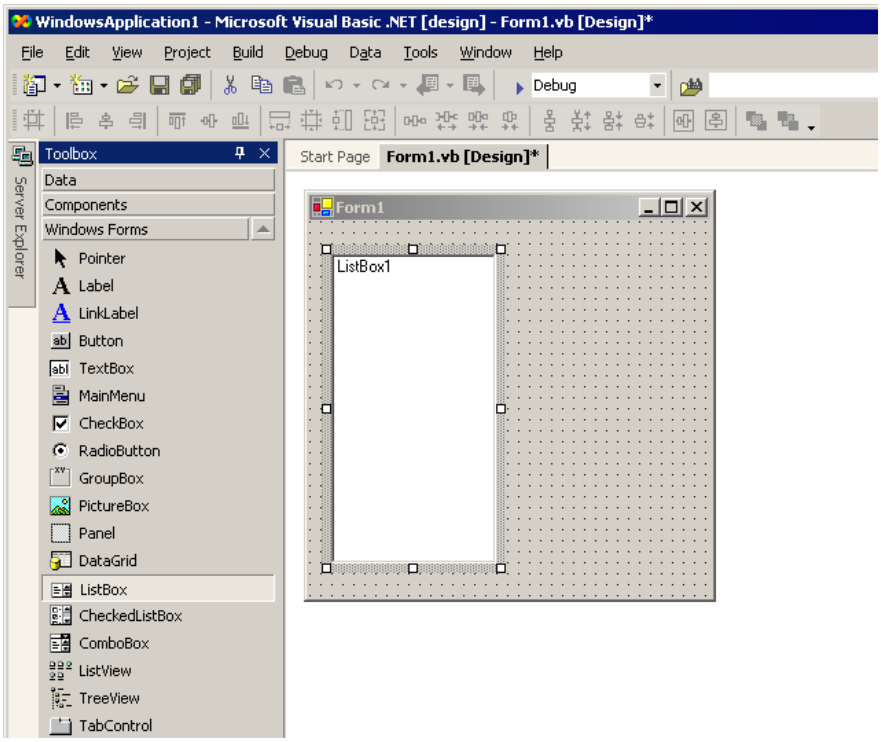
创建 “写入”应用程序，请参阅 20 到 32 步。

创建 “读取” 应用程序

本节介绍下述程序的创建步骤：点击 [Button1] 读取并显示三个数据 (16 位有符号)。

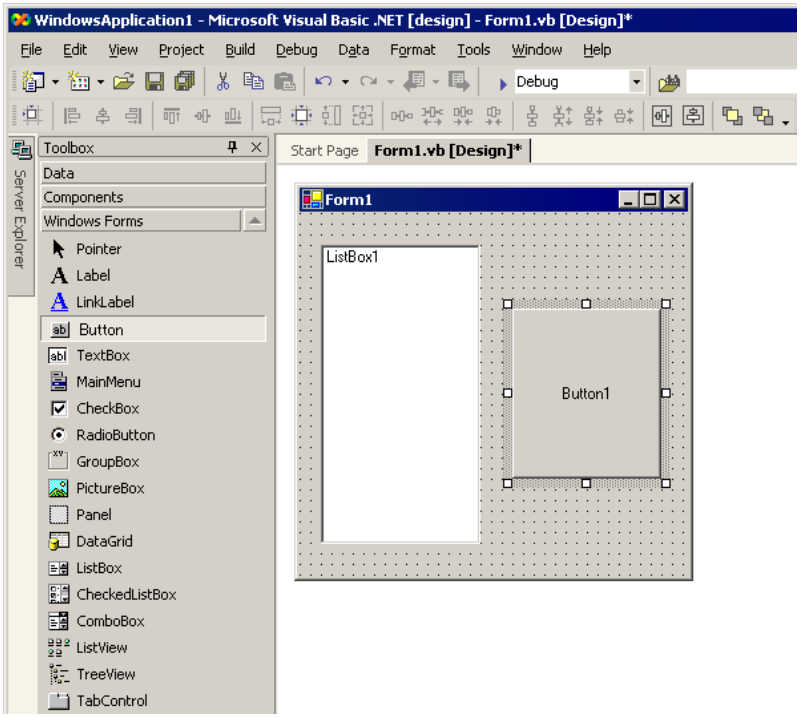


7 选择 [Toolbox] 中的 [ListBox]，将其剪切并粘贴到 [Form1]。

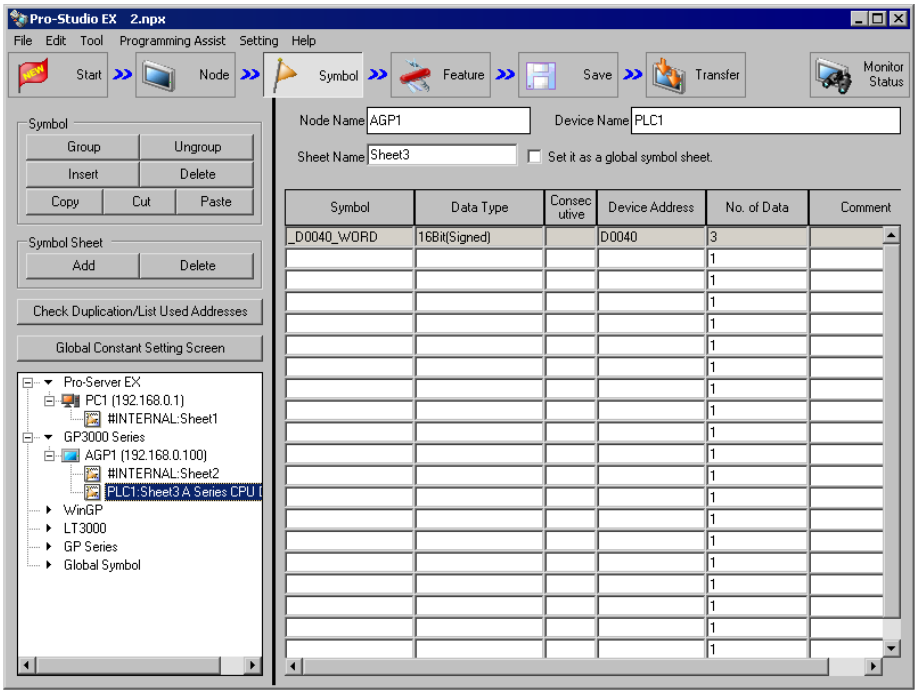


* 如果未显示 [Toolbox]，请从 [View] 菜单中选择 [Toolbox]。

8 选择 [Toolbox] 中的 [Button]，将其剪切并粘贴到 [Form1]。

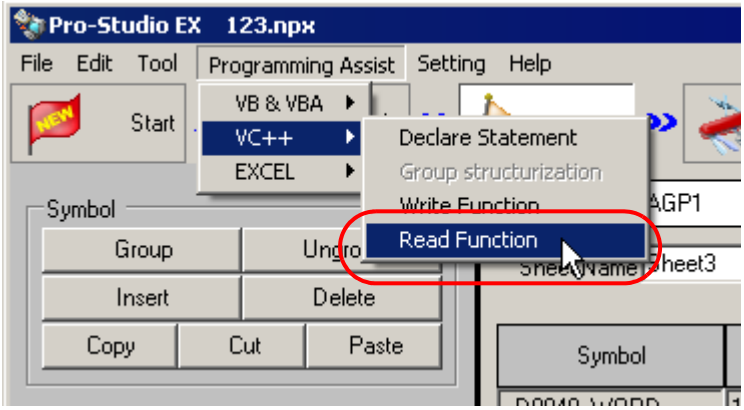


9 从 Pro-Studio EX 中注册的符号名称中选择一个目标符号。

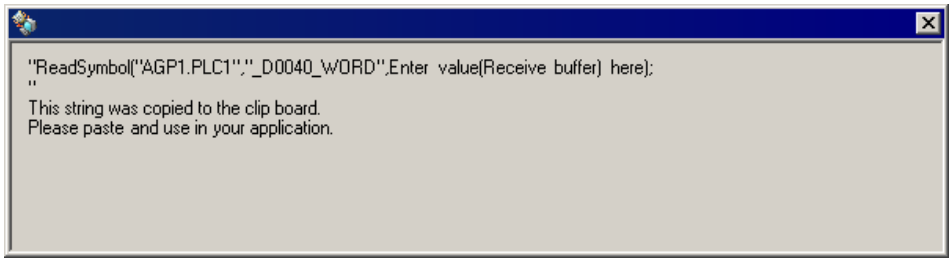


* 上例显示数据类型为 [16Bit(Signed)]，数据数量为 “3”。

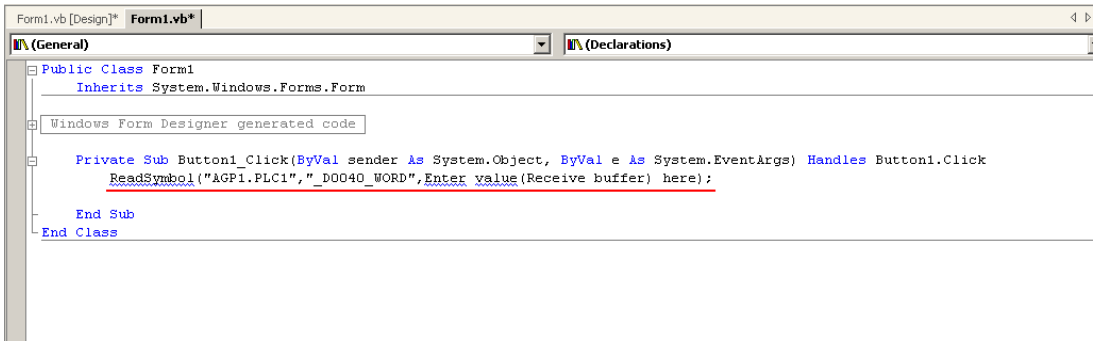
10 从菜单上选择 [Programming Assist] - [VC++] - [Read Function]。



读取函数被复制到剪贴板上。

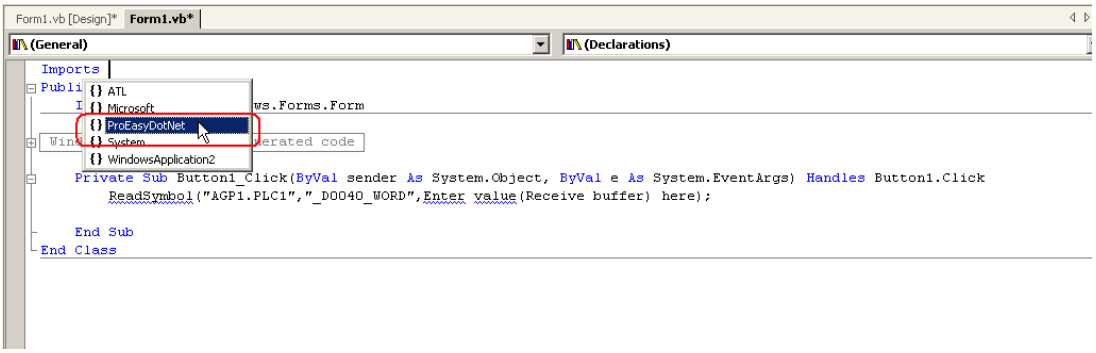


11 双击 [Form1] 上的 [Button1]，将剪贴板上的数据 (读取函数) 粘贴到 “Sub” 语句与 “End Sub” 语句之间。



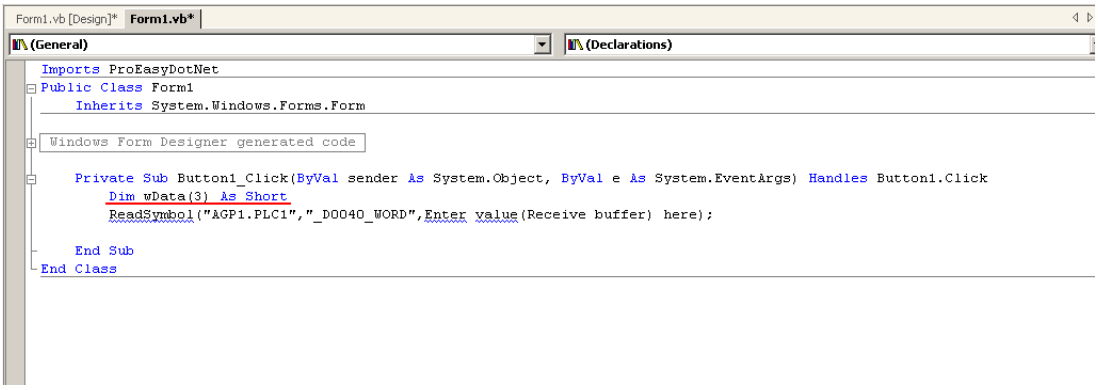
12 导入 ProEasyDotNet 库。

在源代码头部输入 “Imports”，从显示的列表框中选择 [ProEasyDotNet]。

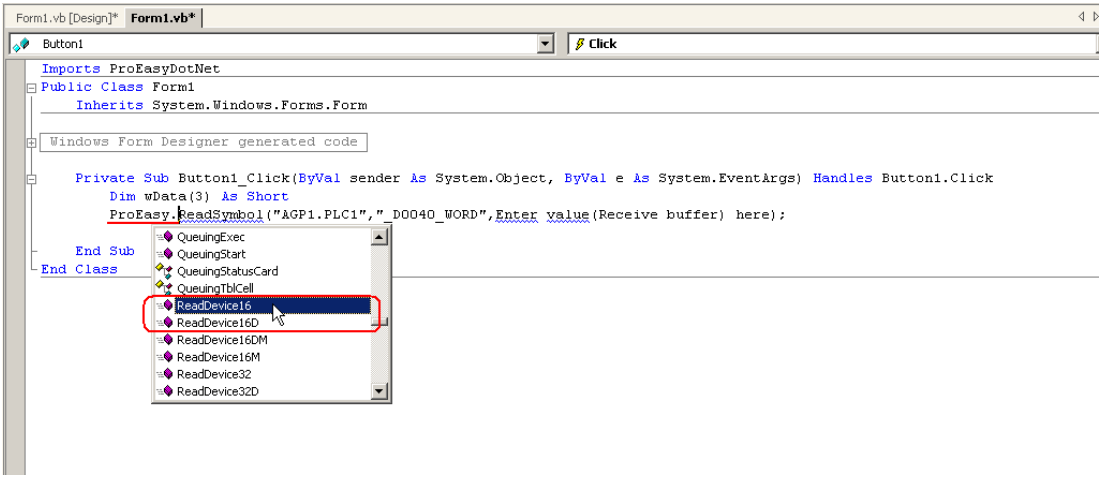


13 为读取数据的保存区域声明一个变量 “wData”。

数组类型 (本例中为 “Short”) 必须与目标符号的数据类型一致。指定与目标符号一致的数据长度 (本例中为 “3”)。



14 在 “ReadSymbol” 前输入 “ProEasy.”，从显示的列表框中选择 [ReadDevice16]。



15 从字符串 (读取函数) 中删除 “ReadSymbol”，该字符串是先前从剪贴板粘贴而来。

```
Form1.vb [Design]*  Form1.vb*
(General)                                   (Declarations)
Imports ProEasyDotNet
Public Class Form1
    Inherits System.Windows.Forms.Form

Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim wData(3) As Short
        ProEasy.ReadDevice16("AGP1.PLC1","_D0040_WORD",Enter value(Receive buffer) here);

    End Sub
End Class
```

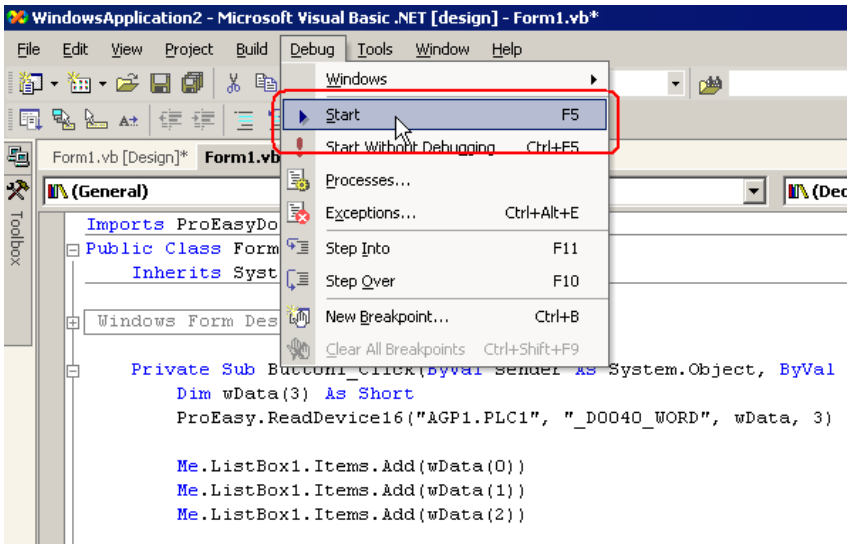
16 指定数据保存区 “wData” 作为第三个参数。在第三个参数后输入 “，” (逗号)，然后输入目标符号的长度 “3” 作为第四个参数。删除行末的 “;” (分号)。

```
Form1.vb [Design]*  Form1.vb*
(General)                                   (Declarations)
Imports ProEasyDotNet
Public Class Form1
    Inherits System.Windows.Forms.Form

Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim wData(3) As Short
        ProEasy.ReadDevice16("AGP1.PLC1","_D0040_WORD",wData,3)

    End Sub
End Class
```

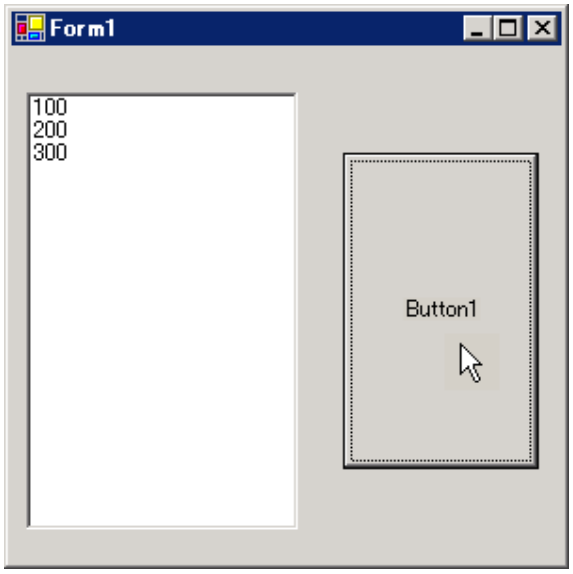
17 依次将三个读取到的数据 (wData(0)、 wData(1)、 wData(2)) 添加到 [ListBox1] 中。

```
Form1.vb [Design]*  Form1.vb*
(General)                                   (Declarations)
Imports ProEasyDotNet
Public Class Form1
    Inherits System.Windows.Forms.Form

Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim wData(3) As Short
        ProEasy.ReadDevice16("AGP1.PLC1", "_D0040_WORD", wData, 3)

        Me.ListBox1.Items.Add(wData(0))
        Me.ListBox1.Items.Add(wData(1))
        Me.ListBox1.Items.Add(wData(2))

    End Sub
End Class
```

18 从 [Debug] 菜单中选择 [Start]。

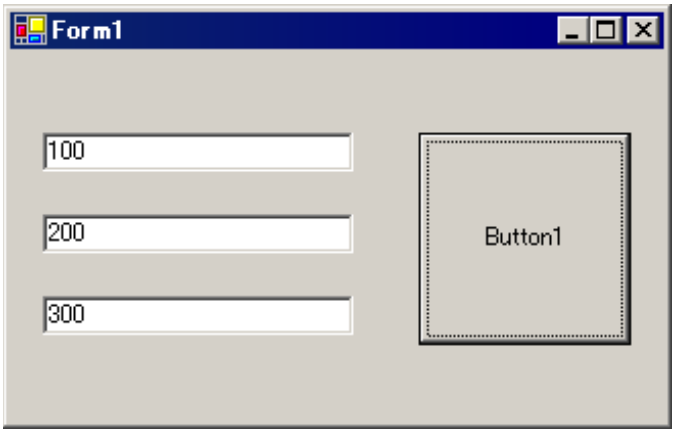


19 点击 [Button1]，将在 [ListBox] 中显示目标符号数据 (三个)。

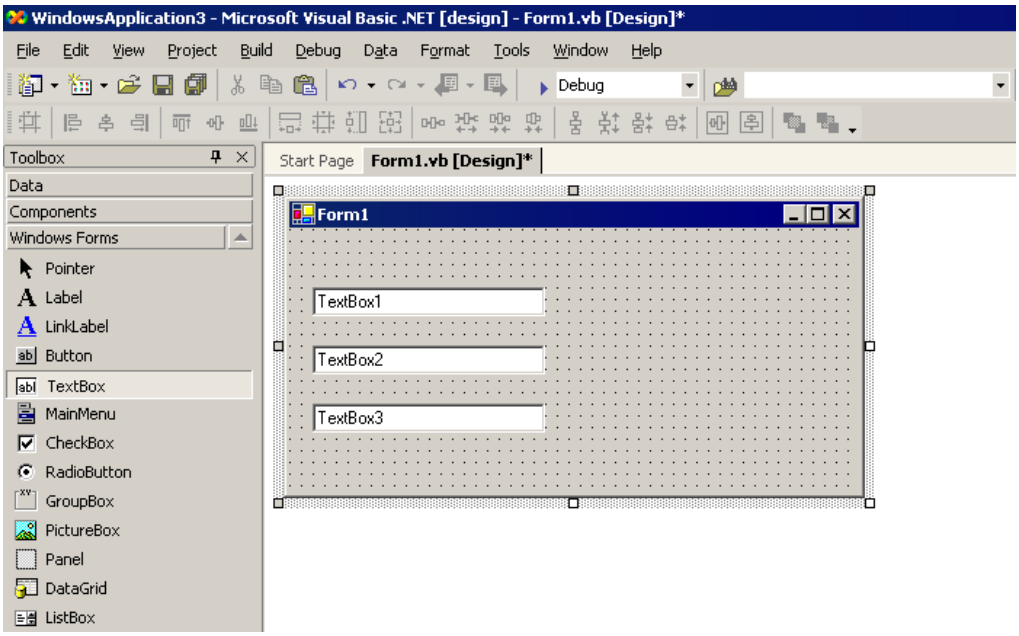


创建 “写入” 应用程序

本节介绍下述程序的创建步骤：点击 [Button1] 写入三个数据 (16 位有符号)。

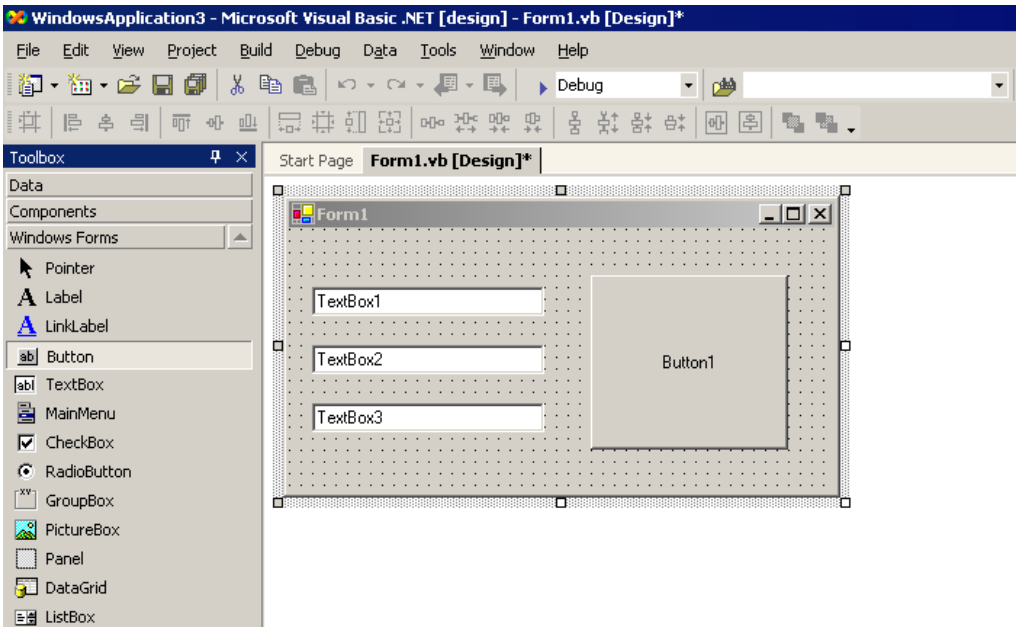


20 选择 [Toolbox] 中的 [TextBox]，将三个文本框剪切并粘贴到 [Form1]。

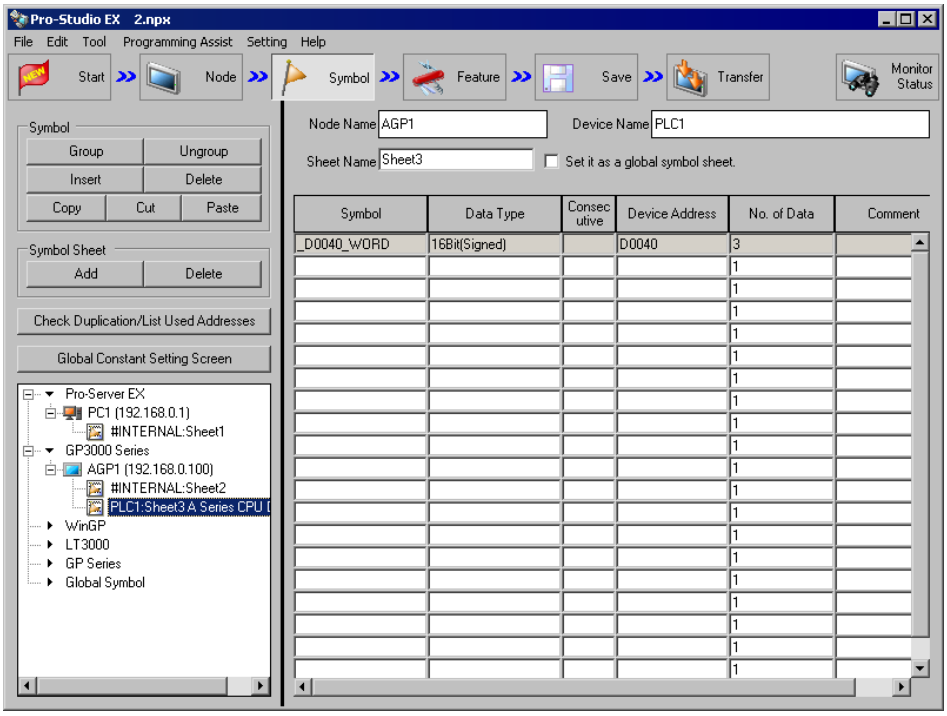


* 如果未显示 [Toolbox]，请从 [View] 菜单中选择 [Toolbox]。

21 选择 [Toolbox] 中的 [Button]，将其剪切并粘贴到 [Form1]。

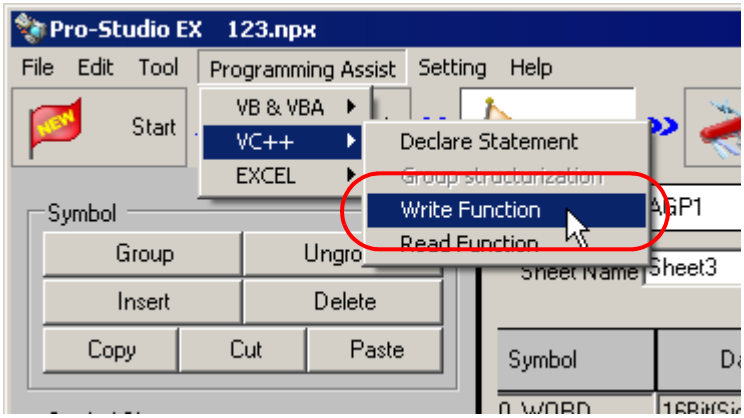


22 从 Pro-Studio EX 中注册的符号名称中选择一个目标符号。（选择首个写入区。）

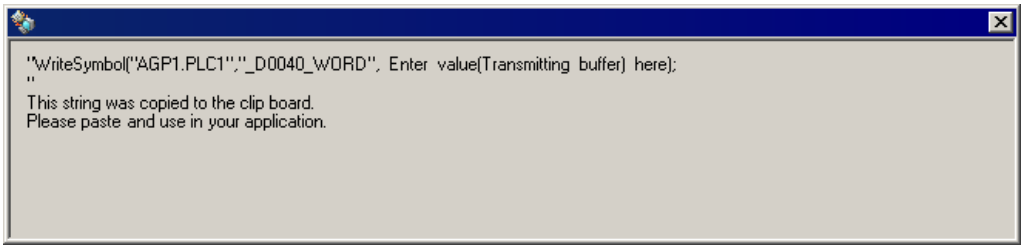


* 上例显示数据类型为 [16Bit(Signed)]，数据数量为 “3”。

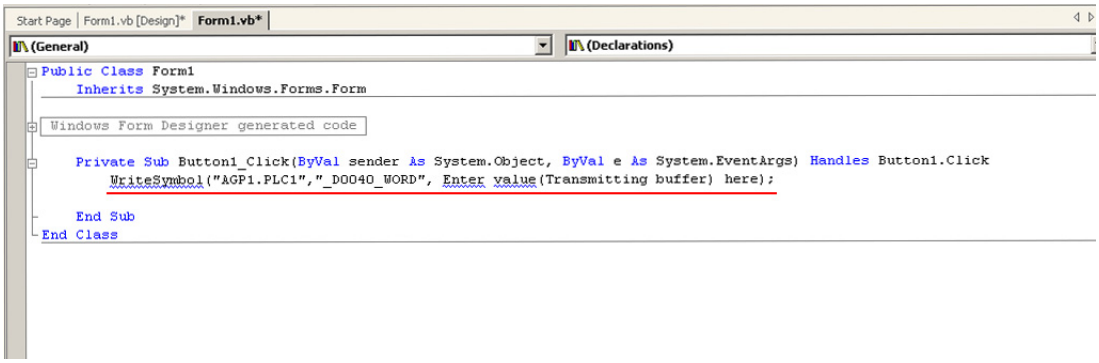
23 从菜单上选择 [Programming Assist] - [VC++] - [Write Function]。



写入函数被复制到剪贴板上。



24 双击 [Form1] 上的 [Button1]，将剪贴板上的数据 (写入函数) 粘贴到 [Button1_Click] 方法 (“Private Sub Button1_Click...” 字符串) 下。



25 导入 ProEasyDotNet 库。

在源代码头部输入 “Imports”，从显示的列表框中选择 [ProEasyDotNet]。

```
Start Page | Form1.vb [Design]* | Form1.vb*
(General)                          (Declarations)
Imports
Publi  {} ATL
       {} Microsoft
       {} ProEasyDotNet
       {} System
       {} WindowsApplication3
       I                    ws.Forms.Form
Win                         nerated code
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        WriteSymbol("AGP1.PLC1","_D0040_WORD", Enter value(Transmitting buffer) here);

    End Sub
End Class
```

26 为写入数据的保存区域声明一个变量 “wData”。

数组类型 (本例中为 “Short”) 必须与目标符号的数据类型一致。指定与目标符号一致的数据长度 (本例中为 “3”)。

```
Start Page | Form1.vb [Design]* | Form1.vb*
(General)                          (Declarations)
Imports ProEasyDotNet
Public Class Form1
    Inherits System.Windows.Forms.Form

Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim wData(3) As Short
        WriteSymbol("AGP1.PLC1","_D0040_WORD", Enter value(Transmitting buffer) here);

    End Sub
End Class
```

27 将准备输入的数据赋值给数组中的 [TextBox1]~[TextBox3]。

```
Start Page | Form1.vb [Design]* | Form1.vb*
(General)                          (Declarations)
Imports ProEasyDotNet
Public Class Form1
    Inherits System.Windows.Forms.Form

Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim wData(3) As Short
        wData(0) = CShort(Me.TextBox1.Text)
        wData(1) = CShort(Me.TextBox2.Text)
        wData(2) = CShort(Me.TextBox3.Text)

        WriteSymbol("AGP1.PLC1","_D0040_WORD", Enter value(Transmitting buffer) here);

    End Sub
End Class
```

28 在 “WriteSymbol” 前输入 “ProEasy.”，从显示的列表框中选择 [WriteDevice16]。

```
Start Page | Form1.vb [Design]* | Form1.vb*
Button1                                  Click
Imports ProEasyDotNet
Public Class Form1
    Inherits System.Windows.Forms.Form

Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim wData(3) As Short
        wData(0) = CShort(Me.TextBox1.Text)
        wData(1) = CShort(Me.TextBox2.Text)
        wData(2) = CShort(Me.TextBox3.Text)

        ProEasy.WriteSymbol("AGP1.PLC1","_D0040_WORD", Enter value(Transmitting buffer) here);
    End Sub
End Class
```

ReadSymbolVariantD
ReadSymbolVariantDM
ReadSymbolVariantM
SymbolInformation
WriteDevice16
WriteDevice16D
WriteDevice16DM
WriteDevice16M
WriteDevice32
WriteDevice32D

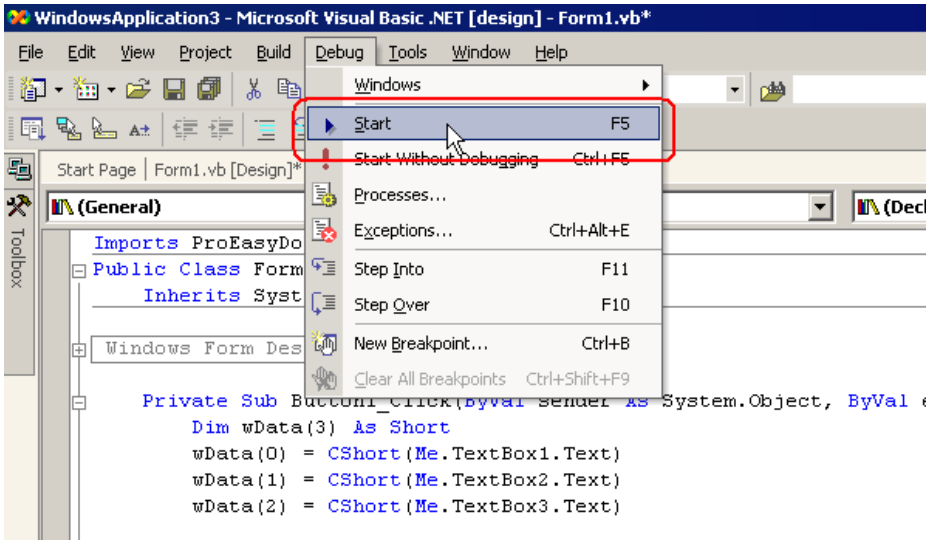
29 从字符串 (写入函数) 中删除 “WriteSymbol”，该字符串是先前从剪贴板粘贴而来。

```
Start Page | Form1.vb [Design]* | Form1.vb*
(General)                                (Declarations)
Imports ProEasyDotNet
Public Class Form1
    Inherits System.Windows.Forms.Form

Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim wData(3) As Short
        wData(0) = CShort(Me.TextBox1.Text)
        wData(1) = CShort(Me.TextBox2.Text)
        wData(2) = CShort(Me.TextBox3.Text)

        ProEasy.WriteDevice16("AGP1.PLC1","_D0040_WORD", Enter value(Transmitting buffer) here);

    End Sub
End Class
```

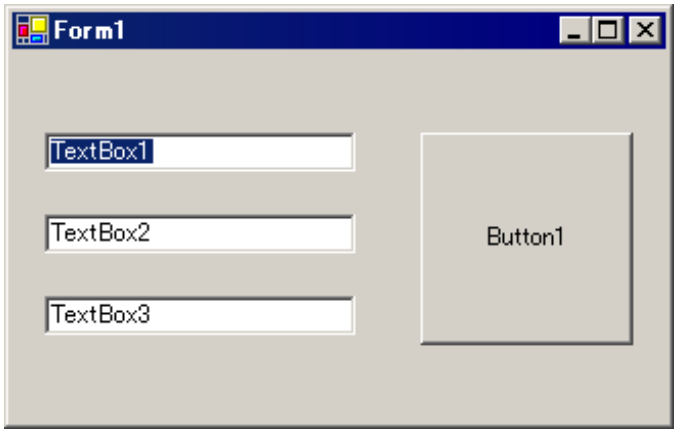
30 指定数据保存区 “wData” 作为第三个参数。在第三个参数后输入 “，” (逗号)，然后输入目标符号的长度 “3” 作为第四个参数。删除行末的 “;” (分号)。

```
Imports ProEasyDotNet
Public Class Form1
    Inherits System.Windows.Forms.Form

Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim wData(3) As Short
        wData(0) = CShort(Me.TextBox1.Text)
        wData(1) = CShort(Me.TextBox2.Text)
        wData(2) = CShort(Me.TextBox3.Text)

        ProEasy.WriteDevice16("AGP1.PLC1", "_D0040_WORD", wData, 3)

    End Sub
End Class
```

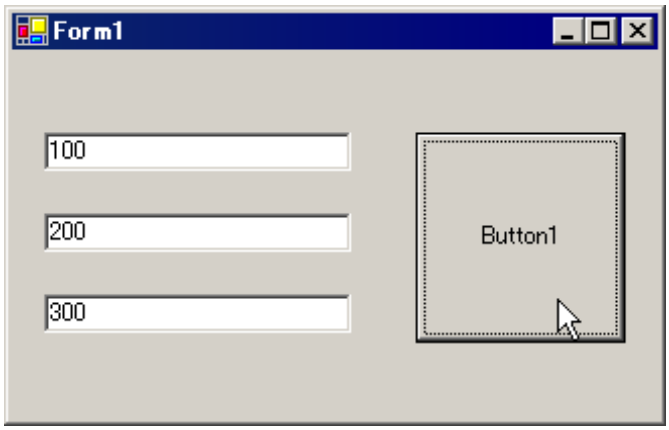
31 从 [Debug] 菜单中选择 [Start]。



32 启动后，在 [TextBox] 中立刻显示字符串 “TextBox*”。

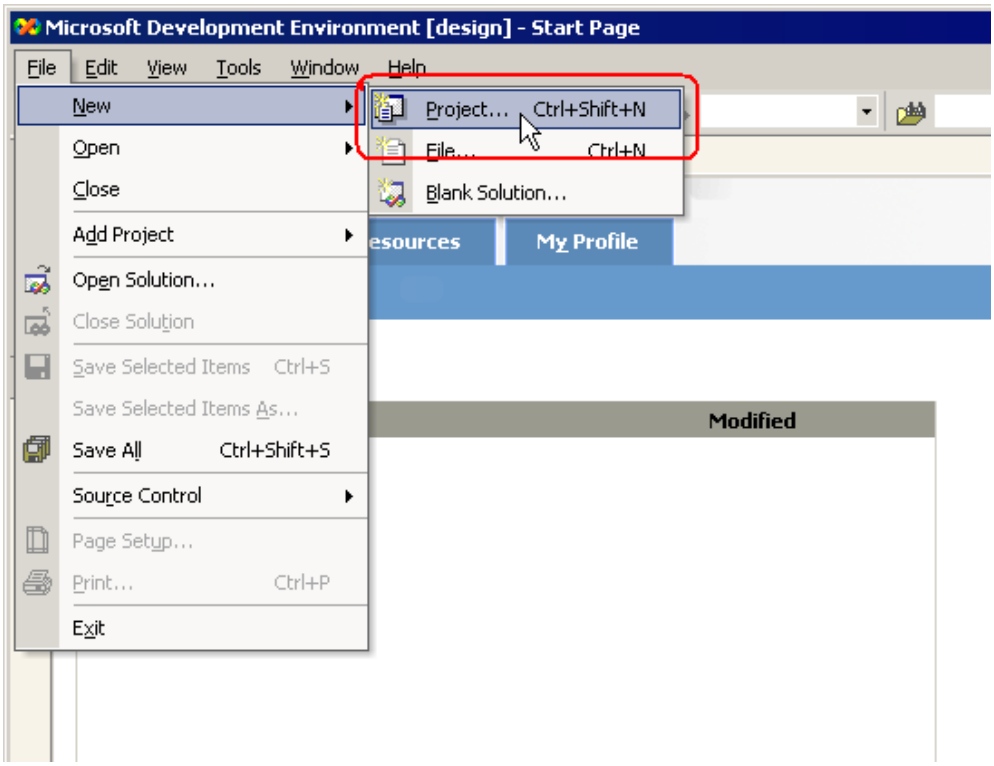


在 [TextBox] 中输入写入数据 (三个) 后，点击 [Button1]。之后即会将数据写入由符号指定的区域。

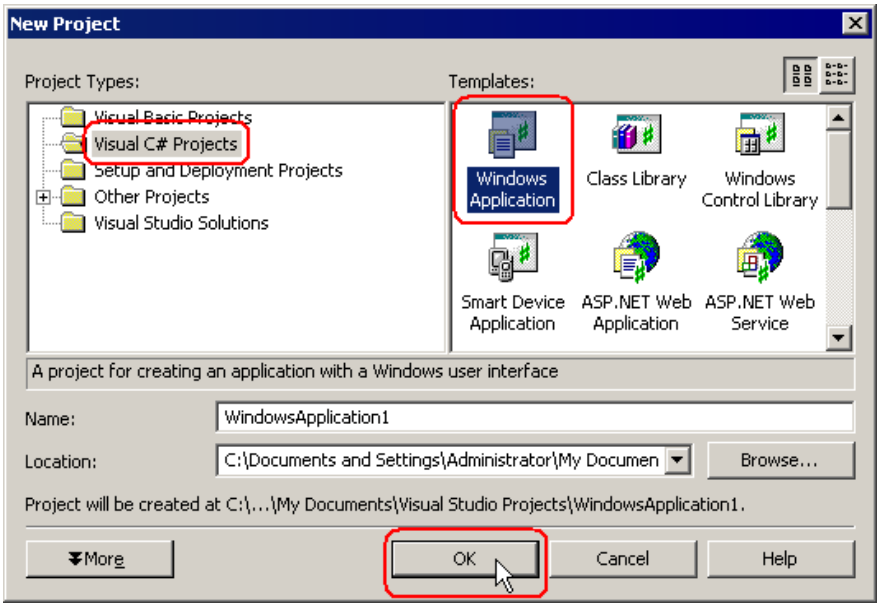


27.11.4 C# .NET 支持的函数

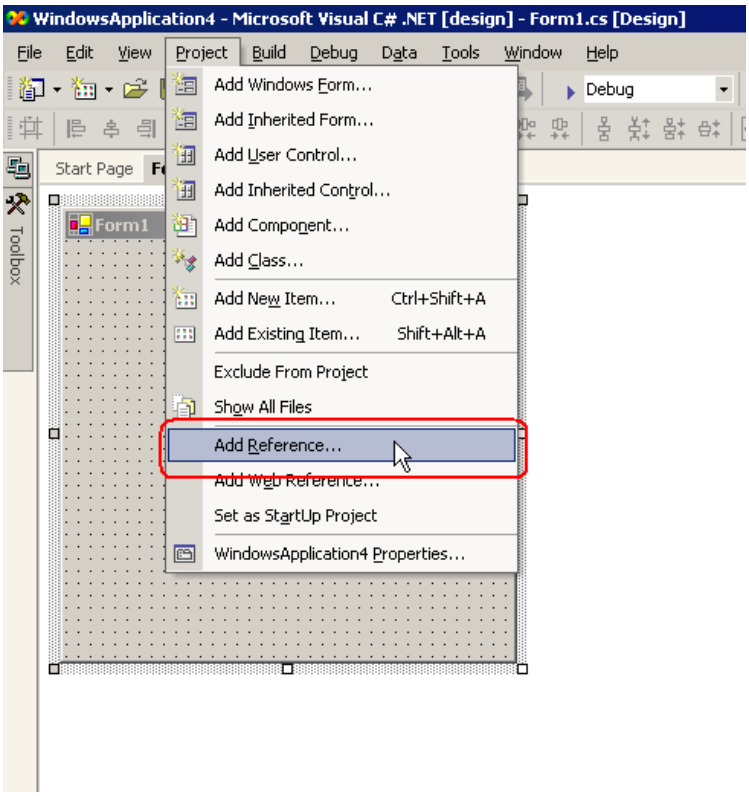
1 启动 Microsoft Visual Studio .NET 2003(或以上版本)，从 [File] 菜单中选择 [New]-[Project]。



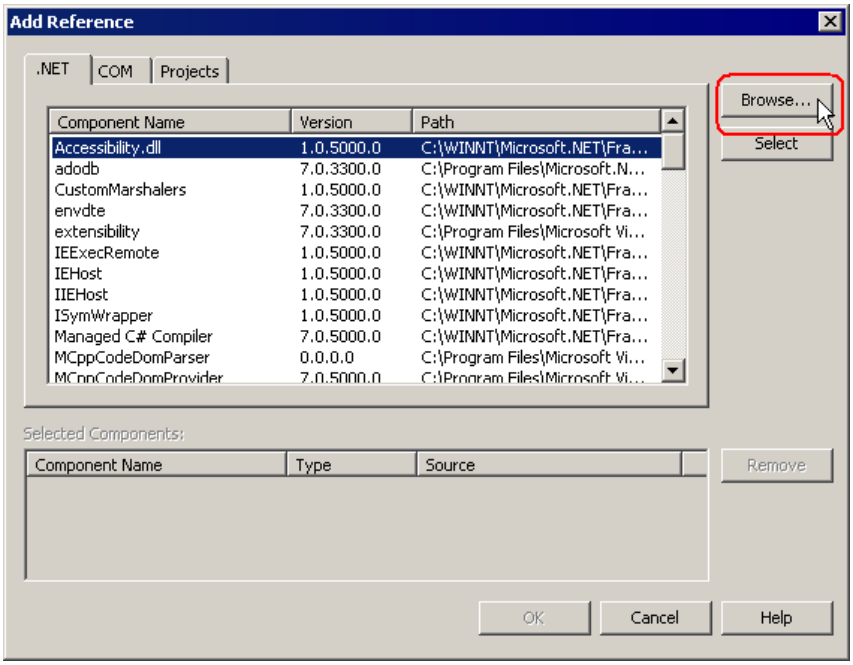
2 从 [Project Types:] 中选择 [Visual C# Projects]，从 [Templates:] 中选择 [Windows Application]，点击 [OK] 按钮。



3 从 [Project] 菜单中选择 [Add Reference]。



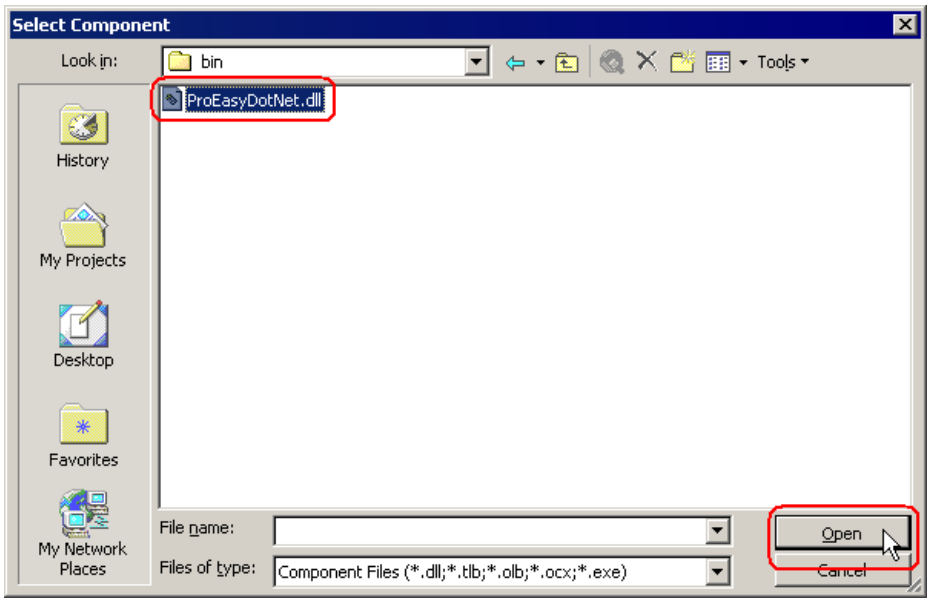
4 点击 [Browse] 按钮。



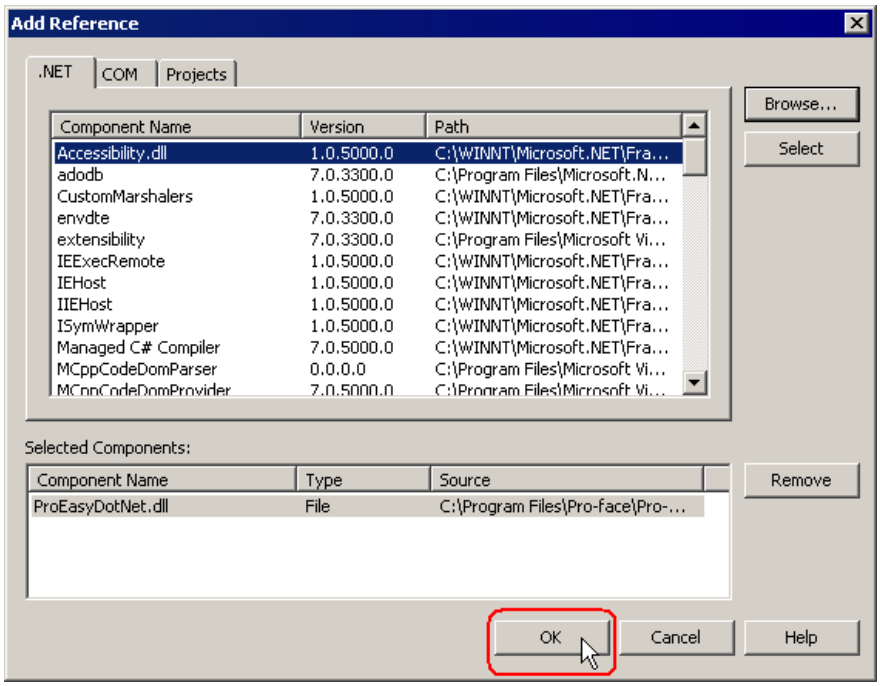
5 指定 ProEasyDotNet.dll 的安装目录，点击 [Open] 按钮。（采用标准安装时，文件保存目录为“C:\Program Files\Pro-face\Pro-Server EX\PRO-SDK\DotNet\bin\ProEazyDotNet.dll”。）

注 释

- ProEasyDotNet 对 Microsoft .NET Framework 1.1 的支持
  - Windows Vista 或以上
    C:\Pro-face\Pro-Server EX\PRO-SDK\DotNet\bin\ProEasyDotNet.dll
  - Windows 2000 / XP / Server 2003
    C:\Program Files\Pro-face\Pro-Server EX\PRO-SDK\DotNet\bin\ProEasyDotNet.dll
- ProEasyDotNet 对 Microsoft .NET Framework 2.0 的支持
  - Windows Vista 或以上
    C:\Pro-face\Pro-Server EX\PRO-SDK\DotNet20\bin\ProEasyDotNet.dll
  - Windows 2000 / XP / Server 2003
    C:\Program Files\Pro-face\Pro-Server EX\PRO-SDK\DotNet20\bin\ProEasyDotNet.dll



6 点击 [OK] 按钮。



“ProEasyDotNet.dll”将被注册。

C#.NET 的运行环境设置至此结束。

上述 1 到 6 步对读取和写入应用程序均适用。

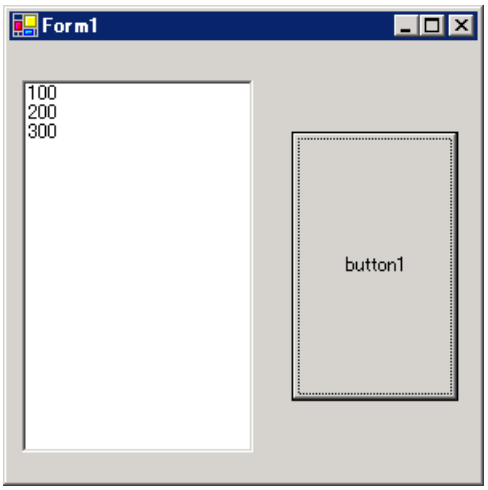
下述步骤则因应用程序是用于读取还是写入而不同，因此分别进行描述。

创建 “读取”应用程序，请参阅 7 到 19 步。

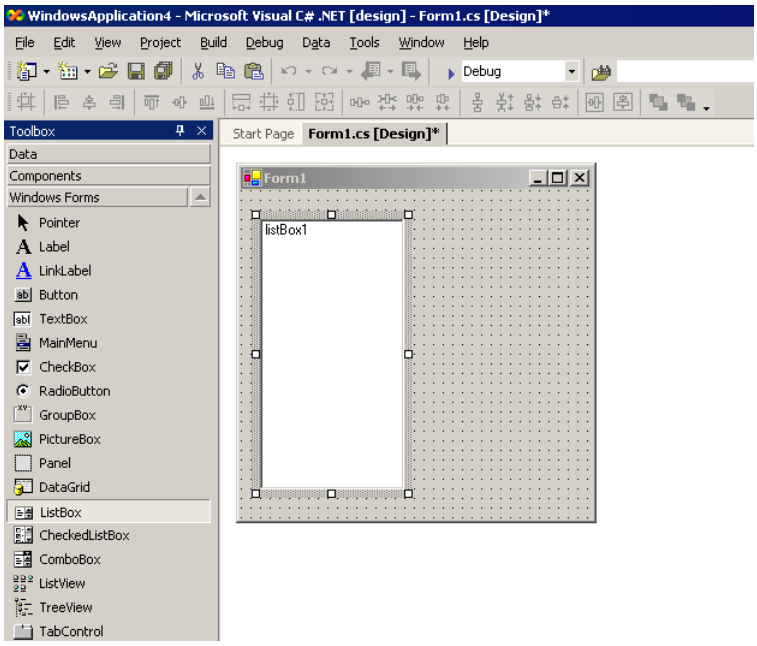
创建 “写入”应用程序，请参阅 20 到 32 步。

创建 “读取”应用程序

本节介绍下述程序的创建步骤：点击 [Button1] 读取并显示三个数据 (16 位有符号)。

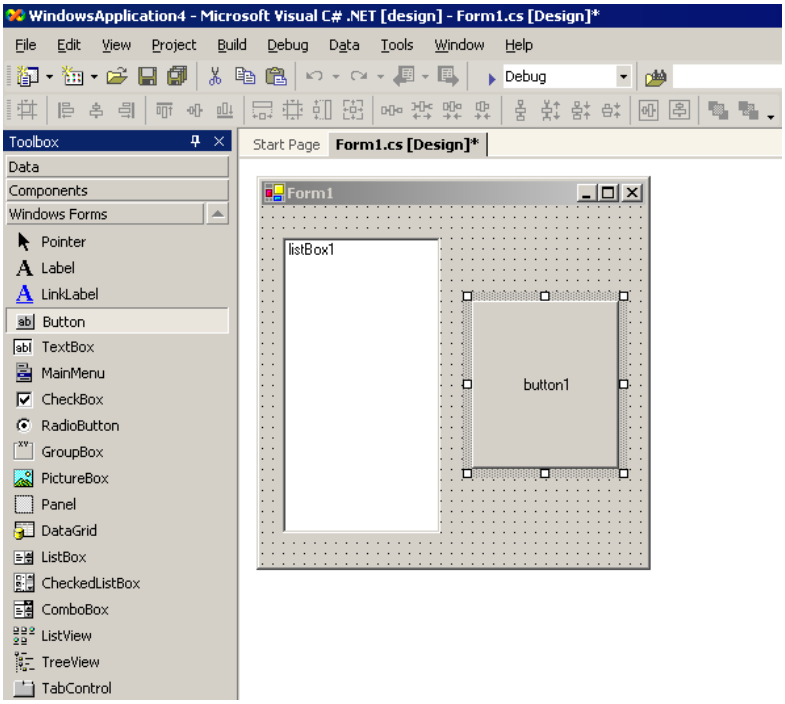


7 选择 [Toolbox] 中的 [ListBox]，将其剪切并粘贴到 [Form1]。

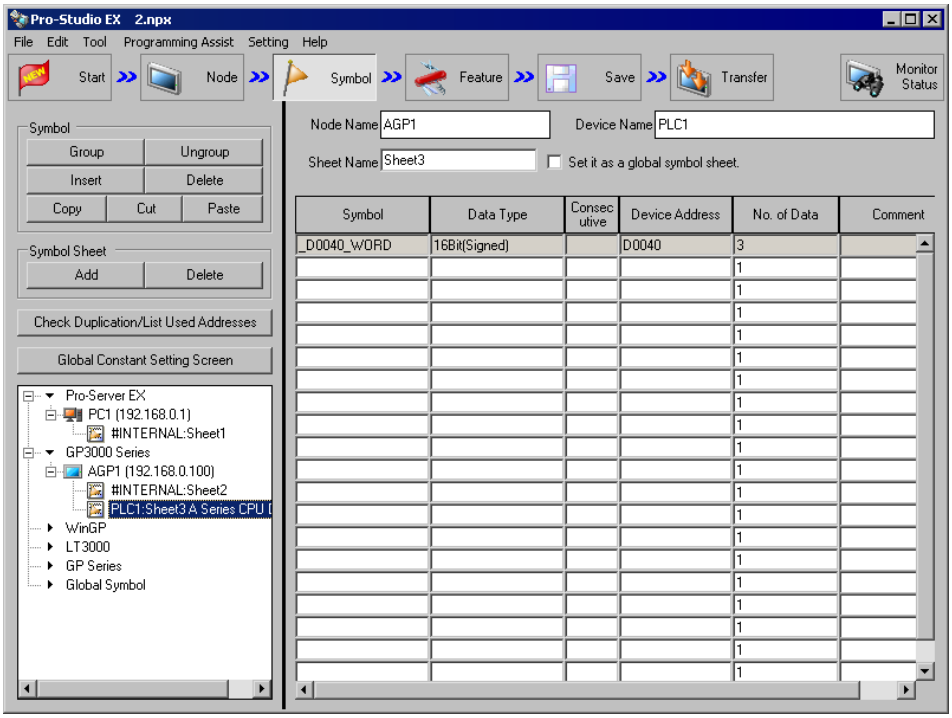


* 如果未显示 [Toolbox]，请从 [View] 菜单中选择 [Toolbox]。

8 选择 [Toolbox] 中的 [Button]，将其剪切并粘贴到 [Form1]。

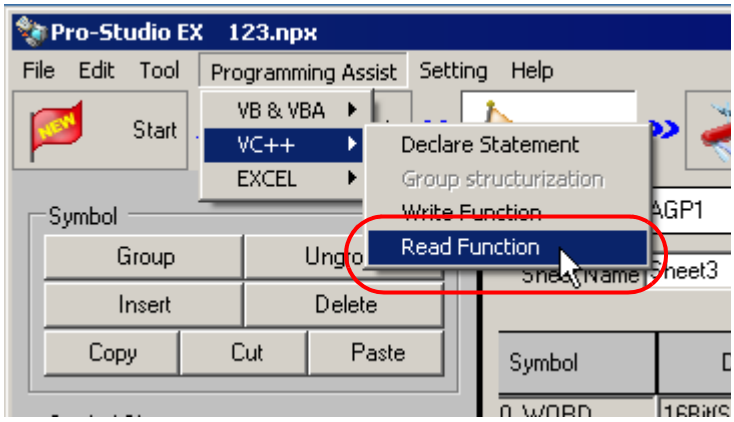


9 从 Pro-Studio EX 中注册的符号名称中选择一个目标符号。

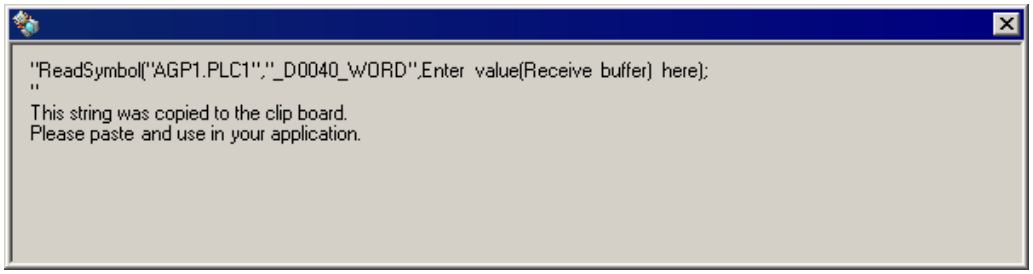


* 上例显示数据类型为 [16Bit(Signed)]，数据数量为 “3”。

10 从菜单上选择 [Programming Assist] - [VC++] - [Read Function]。



读取函数被复制到剪贴板上。



11 双击 [Form1] 上的 [Button1]，将剪贴板上的数据 (读取函数) 粘贴到 [button1_Click] 方法 (“Private void button1_Click...” 字符串) 下。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication4.Form1                    button1_Click(object sender,System.EventArgs e)
13      public class Form1 : System.Windows.Forms.Form
14      {
15          private System.Windows.Forms.ListBox listBox1;
16          private System.Windows.Forms.Button button1;
17          /// <summary>
18          /// Required designer variable.
19          /// </summary>
20          private System.ComponentModel.Container components = null;
21
22          public Form1()...
34          /// <summary>
35          /// Clean up any resources being used.
36          /// </summary>
37          protected override void Dispose( bool disposing )...
49          Windows Form Designer generated code
88
89          /// <summary>
90          /// The main entry point for the application.
91          /// </summary>
92          [STAThread]
93          static void Main()
94          {
95              Application.Run(new Form1());
96          }
97
98          private void button1_Click(object sender, System.EventArgs e)
99          {
100             ReadSymbol("AGP1.PLC1","_D0040_WORD",Enter value(Receive buffer) here);
101
102         }
103     }
104 }
105
106
107
108
109
```

12 描述 ProEasyDotNet 指令。

在源代码头部、“using...” 行的末尾输入 “using ProEasyDotNet;”。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication4.Form1                    listBox1
1   using System;
2   using System.Drawing;
3   using System.Collections;
4   using System.ComponentModel;
5   using System.Windows.Forms;
6   using System.Data;
7
8   using ProEasyDotNet;
9
10
11
12
13
14
15  namespace WindowsApplication4
16  {
17      /// <summary>
18      /// Summary description for Form1.
19      /// </summary>
20      public class Form1 : System.Windows.Forms.Form
21      {
22          private System.Windows.Forms.ListBox listBox1;
23          private System.Windows.Forms.Button button1;
24          /// <summary>
25          /// Required designer variable.
26          /// </summary>
27          private System.ComponentModel.Container components = null;
28
```

13 为读取数据的保存区域声明一个变量 “wData”。

数组类型 (本例中为 “Short”) 必须与目标符号的数据类型一致。指定与目标符号一致的数据长度 (本例中为 “3”)。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication4.Form1                    button1_Click(object sender,System.EventArgs e)
96          /// <summary>
97          /// The main entry point for the application.
98          /// </summary>
99          [STAThread]
100         static void Main()
101         {
102             Application.Run(new Form1());
103         }
104
105         private void button1_Click(object sender, System.EventArgs e)
106         {
107             short[] wData = new short[3];
108             ReadSymbol("AGP1.PLC1","_D0040_WORD",Enter value(Receive buffer) here);
109
110         }
111     }
112 }
113
114
115
116
117
118
```

14 在 “ReadSymbol” 前输入 “ProEasy.”，从显示的列表框中选择 [ReadDevice16]。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication4.Form1                    button1_Click(object sender,System.EventArgs e)
96          /// <summary>
97          /// The main entry point for the application.
98          /// </summary>
99          [STAThread]
100         static void Main()
101         {
102             Application.Run(new Form1());
103         }
104
105         private void button1_Click(object sender, System.EventArgs e)
106         {
107             short[] wData = new short[3];
108             ProEasy.ReadSymbol("AGP1.PLC1","_D0040_WORD",Enter value(Receive buffer) here);
109
110         }
111     }
112 }
113
114
115
116
117
118
119
120
121
122
```

QueuingExec
QueuingStart
QueuingStatusCard
QueuingTblCell
ReadDevice16
ReadDevice16D
ReadDevice16DM
ReadDevice16M
ReadDevice32
ReadDevice32D

15 从字符串 (读取函数) 中删除 “ReadSymbol”，该字符串是先前从剪贴板粘贴而来。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication4.Form1                    button1_Click(object sender,System.EventArgs e)
96          /// <summary>
97          /// The main entry point for the application.
98          /// </summary>
99          [STAThread]
100         static void Main()
101         {
102             Application.Run(new Form1());
103         }
104
105         private void button1_Click(object sender, System.EventArgs e)
106         {
107             short[] wData = new short[3];
108             ProEasy.ReadDevice16("AGP1.PLC1","_D0040_WORD",Enter value(Receive buffer) here);
109
110         }
111     }
112 }
113
114
115
116
```

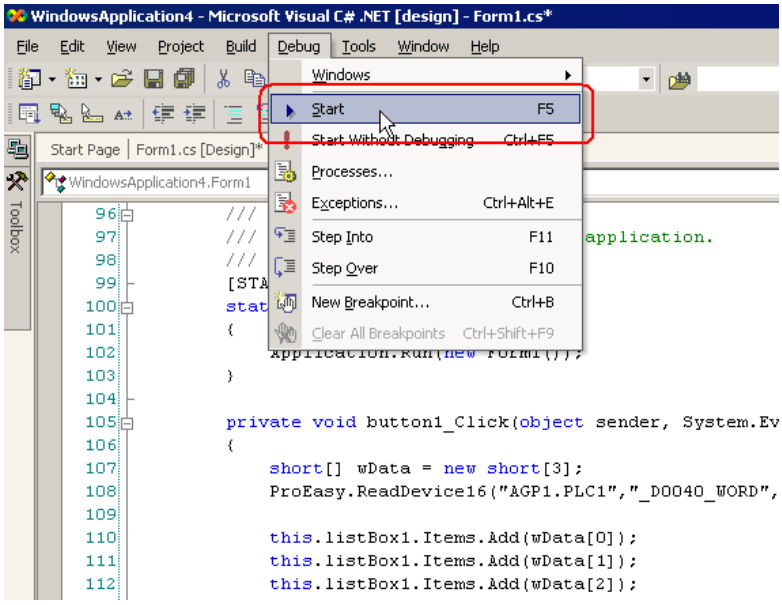
16 指定带引用修饰符 (out) 的数据保存区 “wData”作为第三个参数。在第三个参数后输入 “，” (逗号)，然后输入目标符号的长度 “3”作为第四个参数。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication4.Form1                    button1_Click(object sender,System.EventArgs e)
96          /// <summary>
97          /// The main entry point for the application.
98          /// </summary>
99          [STAThread]
100         static void Main()
101         {
102             Application.Run(new Form1());
103         }
104
105         private void button1_Click(object sender, System.EventArgs e)
106         {
107             short[] wData = new short[3];
108             ProEasy.ReadDevice16("AGP1.PLC1","_D0040_WORD",out wData,3);
109
110         }
111     }
112 }
113
114
115
116
```

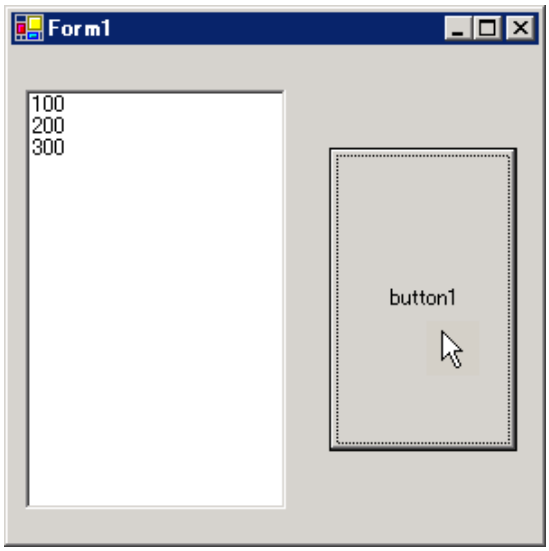
17 依次将三个读取到的数据 (wData[0], wData[1], wData[2]) 添加到 [ListBox1] 中。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication4.Form1                    button1_Click(object sender,System.EventArgs e)
96          /// <summary>
97          /// The main entry point for the application.
98          /// </summary>
99          [STAThread]
100         static void Main()
101         {
102             Application.Run(new Form1());
103         }
104
105         private void button1_Click(object sender, System.EventArgs e)
106         {
107             short[] wData = new short[3];
108             ProEasy.ReadDevice16("AGP1.PLC1","_D0040_WORD",out wData,3);
109
110             this.listBox1.Items.Add(wData[0]);
111             this.listBox1.Items.Add(wData[1]);
112             this.listBox1.Items.Add(wData[2]);
113
114         }
115     }
116 }
117
118
119
120
```

18 从 [Debug] 菜单中选择 [Start]。

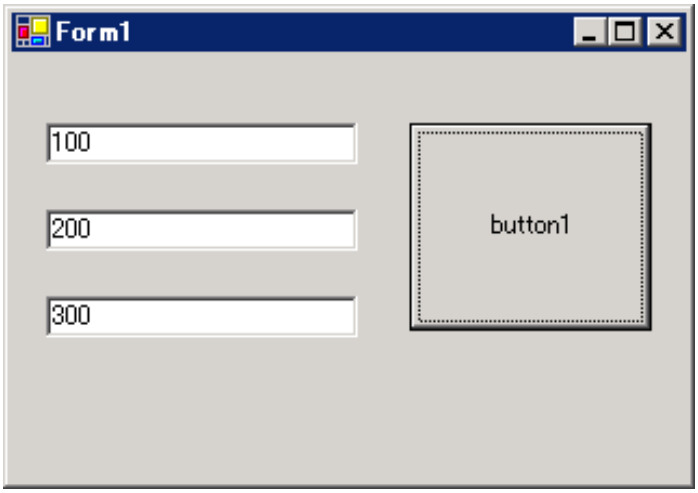


19 点击 [button1]，将在 [ListBox] 中显示目标符号数据 (三个)。

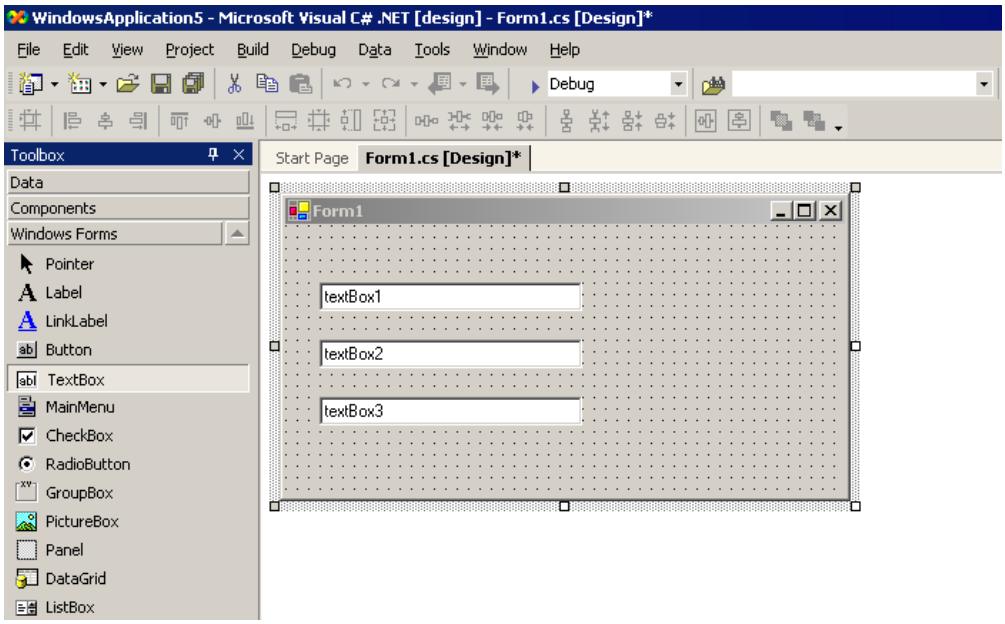


创建 “写入” 应用程序

本节介绍下述程序的创建步骤：点击 [button1] 写入三个数据 (16 位有符号)。

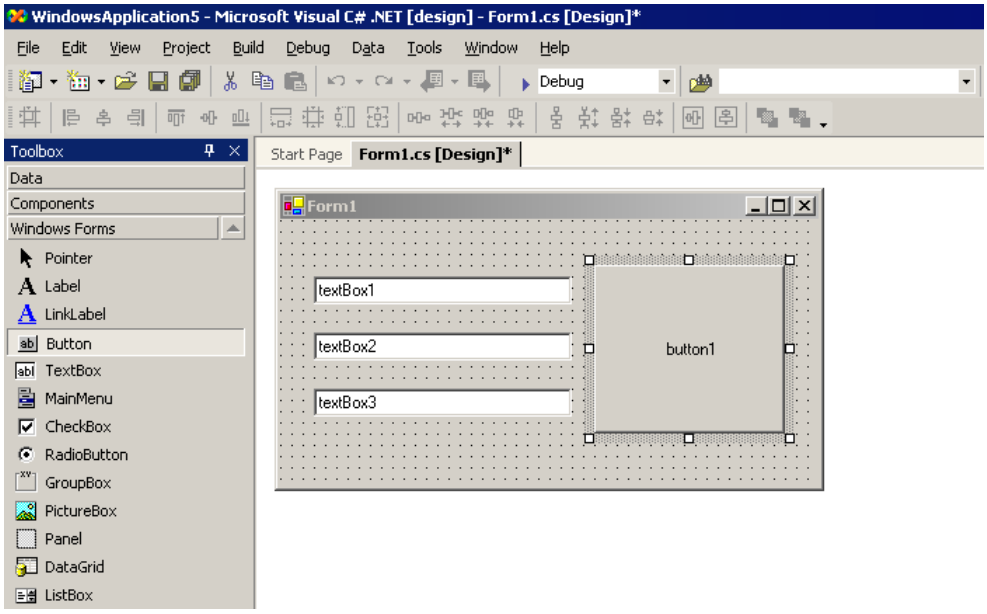


20 选择 [Toolbox] 中的 [TextBox]，将三个文本框剪切并粘贴到 [Form1]。

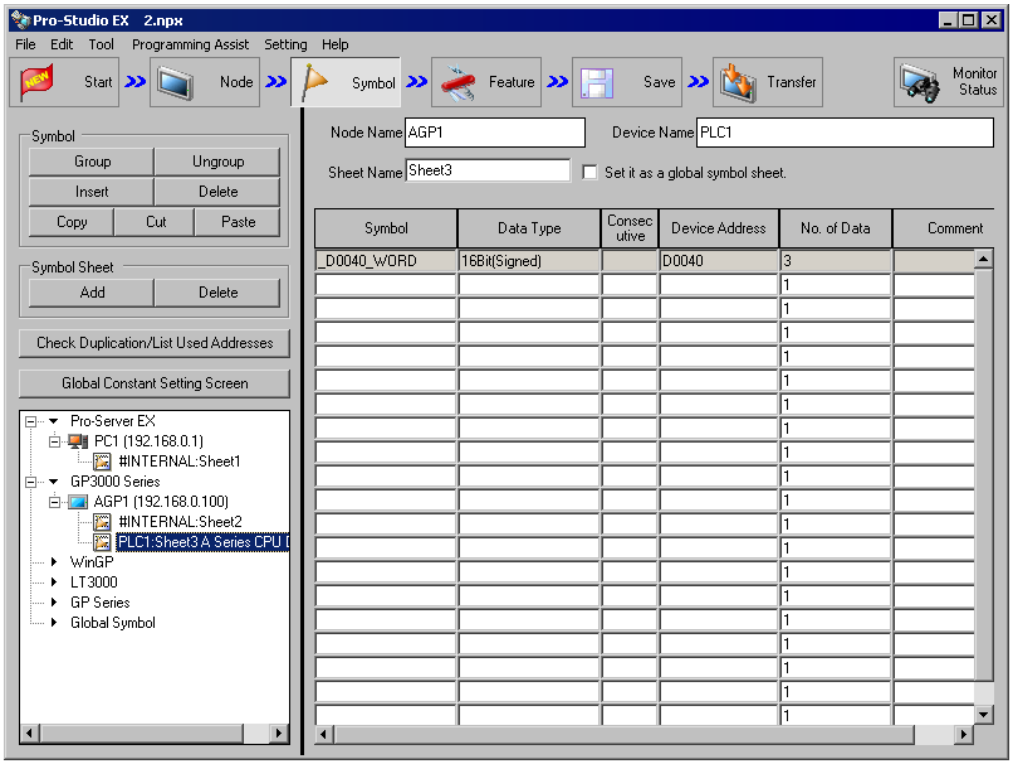


* 如果未显示 [Toolbox]，请从 [View] 菜单中选择 [Toolbox]。

21 选择 [Toolbox] 中的 [Button]，将其剪切并粘贴到 [Form1]。

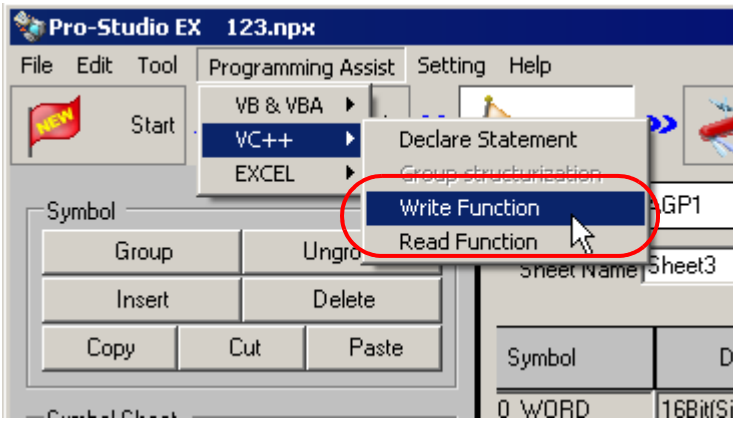


22 从 Pro-Studio EX 中注册的符号名称中选择一个目标符号。（选择首个写入区。）

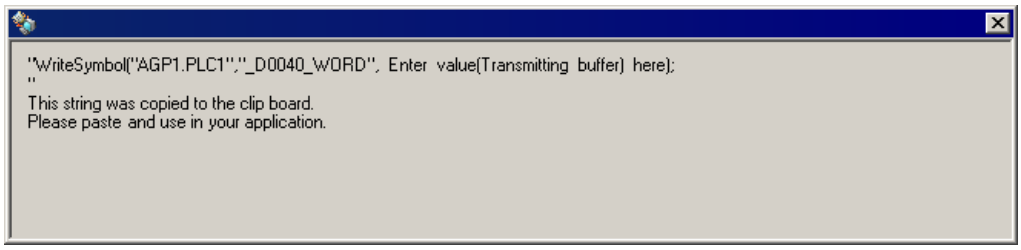


* 上例显示数据类型为 [16Bit(Signed)]，数据数量为 “3”。

23 从菜单上选择 [Programming Assist] - [VC++] - [Write Function]。



写入函数被复制到剪贴板上。



24 双击 [Form1] 上的 [button1]，将剪贴板上的数据 (写入函数) 粘贴到 [button1_Click] 方法 (“Private void button1_Click...” 字符串) 下。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication5.Form1                    button1_Click(object sender,System.EventArgs e)
12      /// </summary>
13      public class Form1 : System.Windows.Forms.Form
14      {
15          private System.Windows.Forms.TextBox textBox1;
16          private System.Windows.Forms.TextBox textBox2;
17          private System.Windows.Forms.TextBox textBox3;
18          private System.Windows.Forms.Button button1;
19          /// <summary>
20          /// Required designer variable.
21          /// </summary>
22          private System.ComponentModel.Container components = null;
23
24          public Form1()...
36          /// <summary>
37          /// Clean up any resources being used.
38          /// </summary>
39          protected override void Dispose( bool disposing )...
51          Windows Form Designer generated code
111
112         /// <summary>
113         /// The main entry point for the application.
114         /// </summary>
115         [STAThread]
116         static void Main()
117         {
118             Application.Run(new Form1());
119         }
120
121         private void button1_Click(object sender, System.EventArgs e)
122         {
123             WriteSymbol("AGP1.PLC1","_D0040_WORD", Enter value(Transmitting buffer) here);
124
125         }
126     }
127 }
128
129
130
131
132
```

25 描述 ProEasyDotNet 指令。

在源代码头部、“using...” 行的末尾输入 “using ProEasyDotNet;”。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication5.Form1                    button1_Click(object sender,System.EventArgs e)
1   using System;
2   using System.Drawing;
3   using System.Collections;
4   using System.ComponentModel;
5   using System.Windows.Forms;
6   using System.Data;
7
8   using ProEasyDotNet;
9
10  namespace WindowsApplication5
11  {
12      /// <summary>
13      /// Summary description for Form1.
14      /// </summary>
15      public class Form1 : System.Windows.Forms.Form
16      {
17          private System.Windows.Forms.TextBox textBox1;
18          private System.Windows.Forms.TextBox textBox2;
19          private System.Windows.Forms.TextBox textBox3;
20          private System.Windows.Forms.Button button1;
```

26 为写入数据的保存区域声明一个变量 “wData”。

数组类型 (本例中为 “Short”) 必须与目标符号的数据类型一致。指定与目标符号一致的数据长度 (本例中为 “3”)。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication5.Form1                    button1_Click(object sender,System.EventArgs e)
 25
 26         public Form1()...
 38         /// <summary>
 39         /// Clean up any resources being used.
 40         /// </summary>
 41         protected override void Dispose( bool disposing )...
 53         Windows Form Designer generated code
113
114         /// <summary>
115         /// The main entry point for the application.
116         /// </summary>
117         [STAThread]
118         static void Main()
119         {
120             Application.Run(new Form1());
121         }
122
123         private void button1_Click(object sender, System.EventArgs e)
124         {
125             short[] wData = new short[3];
126
127             WriteSymbol("AGP1.PLC1","_D0040_WORD", Enter value(Transmitting buffer) here);
128
129         }
130     }
131 }
132
133
134
135
136
```

27 将准备输入的数据赋值给数组中的 [textBox1]~[textBox3]。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication5.Form1                    button1_Click(object sender,System.EventArgs e)
 26         public Form1()...
 38         /// <summary>
 39         /// Clean up any resources being used.
 40         /// </summary>
 41         protected override void Dispose( bool disposing )...
 53         Windows Form Designer generated code
113
114         /// <summary>
115         /// The main entry point for the application.
116         /// </summary>
117         [STAThread]
118         static void Main()
119         {
120             Application.Run(new Form1());
121         }
122
123         private void button1_Click(object sender, System.EventArgs e)
124         {
125             short[] wData = new short[3];
126             wData[0] = short.Parse(this.textBox1.Text);
127             wData[1] = short.Parse(this.textBox2.Text);
128             wData[2] = short.Parse(this.textBox3.Text);
129
130             WriteSymbol("AGP1.PLC1","_D0040_WORD", Enter value(Transmitting buffer) here);
131
132         }
133     }
134 }
135
136
137
138
```

28 在 “WriteSymbol” 前输入 “ProEasy.”，从显示的列表框中选择 [WriteDevice16]。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication5.Form1                    button1_Click(object sender,System.EventArgs e)
26          public Form1()...
38          /// <summary>
39          /// Clean up any resources being used.
40          /// </summary>
41          protected override void Dispose( bool disposing )...
53          Windows Form Designer generated code
113
114         /// <summary>
115         /// The main entry point for the application.
116         /// </summary>
117         [STAThread]
118         static void Main()
119         {
120             Application.Run(new Form1());
121         }
122
123         private void button1_Click(object sender, System.EventArgs e)
124         {
125             short[] wData = new short[3];
126             wData[0] = short.Parse(this.textBox1.Text);
127             wData[1] = short.Parse(this.textBox2.Text);
128             wData[2] = short.Parse(this.textBox3.Text);
129
130             ProEasy.WriteSymbol("AGP1.PLC1","_D0040_WORD", Enter value(Transmitting buffer) here);
131
132         }
133     }
134 }
135
136
137
138
139
140
141
142
143
144
145
146
```

```
ReadSymbolVariantDM
ReadSymbolVariantM
ReferenceEquals
SymbolInformation
WriteDevice16
WriteDevice16D
WriteDevice16DM
WriteDevice16M
WriteDevice32
WriteDevice32D
```

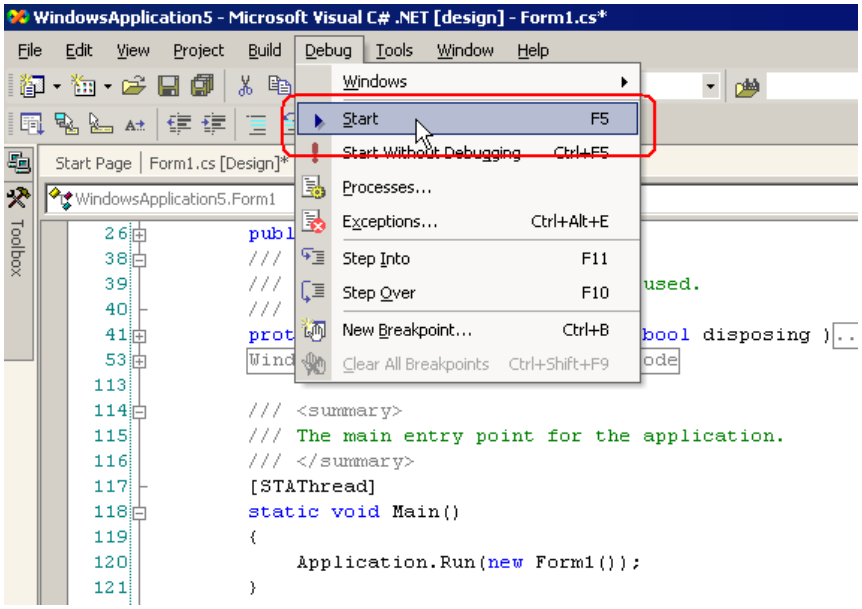
29 从字符串 (写入函数) 中删除 “WriteSymbol”，该字符串是先前从剪贴板粘贴而来。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication5.Form1                    button1_Click(object sender,System.EventArgs e)
26          public Form1()...
38          /// <summary>
39          /// Clean up any resources being used.
40          /// </summary>
41          protected override void Dispose( bool disposing )...
53          Windows Form Designer generated code
113
114         /// <summary>
115         /// The main entry point for the application.
116         /// </summary>
117         [STAThread]
118         static void Main()
119         {
120             Application.Run(new Form1());
121         }
122
123         private void button1_Click(object sender, System.EventArgs e)
124         {
125             short[] wData = new short[3];
126             wData[0] = short.Parse(this.textBox1.Text);
127             wData[1] = short.Parse(this.textBox2.Text);
128             wData[2] = short.Parse(this.textBox3.Text);
129
130             ProEasy.WriteDevice16("AGP1.PLC1","_D0040_WORD", Enter value(Transmitting buffer) here);
131
132         }
133     }
134 }
135
136
137
138
```

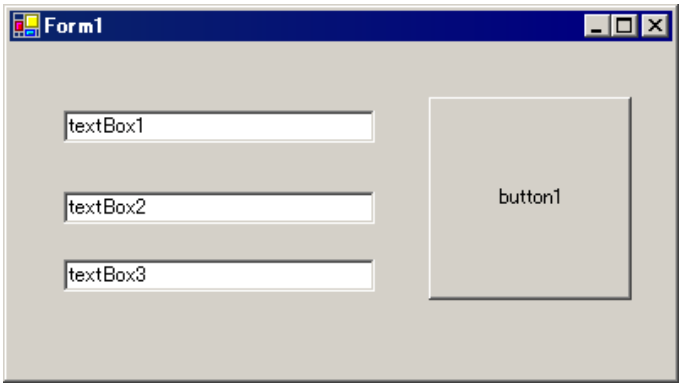
30 指定数据保存区 “wData” 作为第三个参数。在第三个参数后输入 “，” (逗号)，然后输入目标符号的长度 “3” 作为第四个参数。

```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication5.Form1                    button1_Click(object sender,System.EventArgs e)
26          public Form1()...
38          /// <summary>
39          /// Clean up any resources being used.
40          /// </summary>
41          protected override void Dispose( bool disposing )...
53          Windows Form Designer generated code
113
114         /// <summary>
115         /// The main entry point for the application.
116         /// </summary>
117         [STAThread]
118         static void Main()
119         {
120             Application.Run(new Form1());
121         }
122
123         private void button1_Click(object sender, System.EventArgs e)
124         {
125             short[] wData = new short[3];
126             wData[0] = short.Parse(this.textBox1.Text);
127             wData[1] = short.Parse(this.textBox2.Text);
128             wData[2] = short.Parse(this.textBox3.Text);
129
130             ProEasy.WriteDevice16("AGP1.PLC1","_D0040_WORD",wData,3);
131
132         }
133     }
134 }
135
136
137
138
```

31 从 [Debug] 菜单中选择 [Start]。



32 启动后，在 [TextBox] 中立刻显示字符串 “textBox*”。



在 [TextBox] 中输入写入数据 (三个) 后，点击 [button1]。之后即会将数据写入由符号指定的区域。