26 Designing Your Own Program

26.1 Using API Functions 26-2
26.2 Device Access APIs 26-19
26.3 Cache Buffer Control APIs 26-30
26.4 Queuing Access Control APIs 26-36
26.5 System APIs 26-39
26.6 SRAM Data Access APIs 26-46
26.7 CF Card APIs 26-52
26.8 Other APIs 26-61
26.9 Precautions for Using APIs 26-66
26.10 Using APIs (Examples) 26-77

26.1 Using API Functions

■ Reading and writing a Device/PLC

Single-handle functions (☞26.1.1 Single-/Multi-Handle Functions)

- Read API
  - Cache type (☞26.1.2 Cache/Direct Type)
  - Direct type (☞26.1.2 Cache/Direct Type)
- Write API
  - Direct type (☞26.1.2 Cache/Direct Type)
  - With cache refresh after writing (☞26.1.2 Cache/Direct Type)

■ PLC communication with multiple devices

Multi-handle functions (☞26.1.1 Single-/Multi-Handle Functions)

- Read API
  - Cache type (☞26.1.2 Cache/Direct Type)
  - Direct type (☞26.1.2 Cache/Direct Type)
- Write API
  - Direct type (☞26.1.2 Cache/Direct Type)
  - With cache refresh after writing (☞26.1.2 Cache/Direct Type)

■ For effective communication

- Group symbol access (☞26.1.4 Group Access)
- Queuing access (☞26.1.5 Queuing Access)

■ Other functions

- System APIs (→26.1.7 System APIs)
- SRAM Data Access APIs (→26.1.8 SRAM Data Access APIs)
- CF Card APIs (→26.1.9 CF Card APIs)
- Other APIs (→26.8 Other APIs)

26.1.1 Single-/Multi-Handle Functions

Single-Handle APIs

This API is used for sequential communications with target devices. During a call of an API, you cannot call another API.

To call an API, however, you need not perform a troublesome procedure such as 'Pro-Server EX' access handle acquisition.

Multi-Handle APIs

This API enables simultaneous use of single-handle API features for multiple devices. For differentiation from Single-Handle APIs, Multi-Handle APIs are identified with a capital "M" at the end of each API name.

For example, a Multi-Handle API that provides the same feature as a Single-Handle API "ReadDeviceVariant()" is named "ReadDeviceVariantM()".

Multi-Handle APIs can be used for multi-thread applications, or for simultaneous access to multiple Devices/ PLCs.

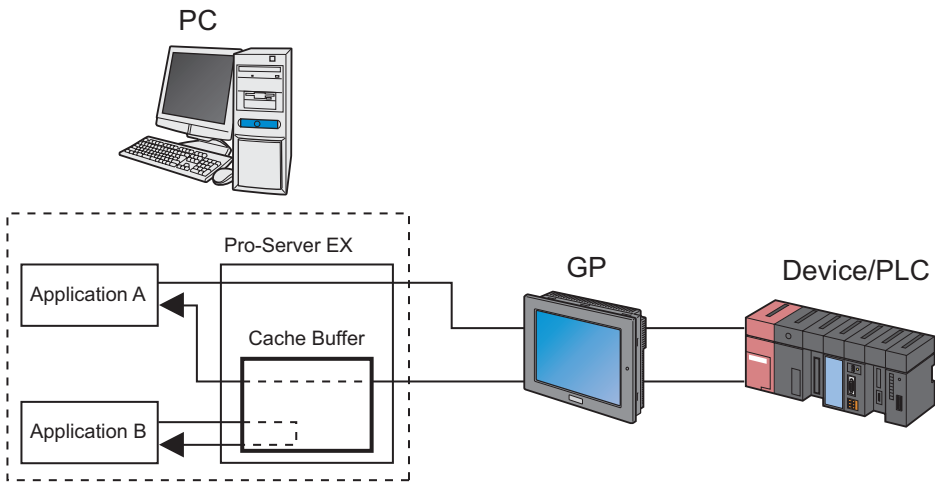
26.1.2 Cache/Direct Type

Cache Read

When multiple applications send reading requests to the same device/PLC, it takes time if 'Pro-Server EX' accesses the Device/PLC to meet individual applications' reading request one by one.

With the Cache Read feature, however, when two applications A and B send reading requests to the same Device/PLC, 'Pro-Server EX' reads data from the Device/PLC according to the request of Application A first, stores the read data into the internal cache buffer, and sends the data to Application A in response to the reading request. Then, according to the request of Application B, 'Pro-Server EX' sends the data stored in the cache buffer to Application B, since the response data are already stored together with the data for Application A.

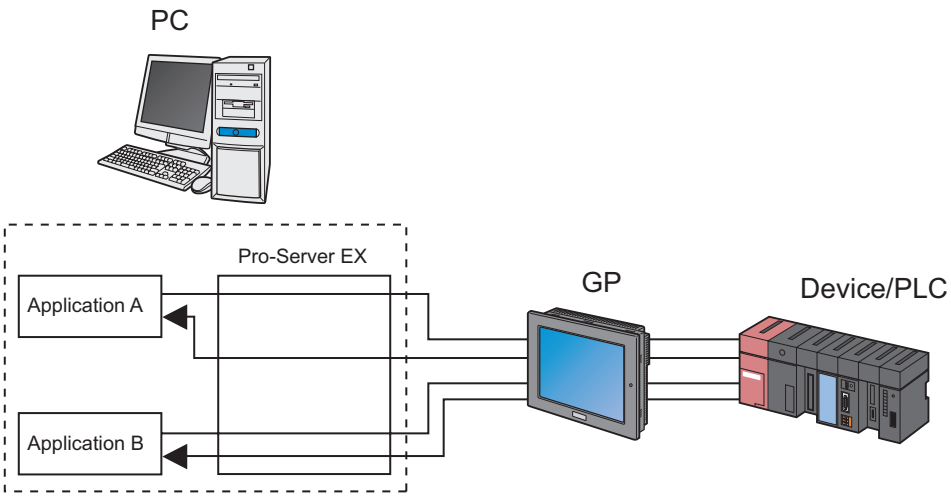
'Pro-Server EX' also provides cache buffer control APIs. Refer to "26.3 Cache Buffer Control APIs" for more details.



Direct Read

This feature always reads latest data from a Device/PLC, regardless of cache status.

Direct Read APIs are identified with a capital "D" or "DM" at the end of each API name.



Direct Write

This API writes values. Direct Write APIs are identified with a capital "D" or "DM" at the end of each API name.

Write with Cache Refresh

When caching data from a device, 'Pro-Server EX' rereads the relevant device data after writing values, to refresh the cache data.

The processing speed of this API is lower than that of Direct Write APIs. When 'Pro-Server EX' has cache-read device data, use Write with Cache Refresh.

26.1.3 Cache Buffer Control APIs

Cache Buffer Control APIs allow you to know whether cache data for a target device has been updated or not.

NOTE
- Cache Buffer Control APIs are not intended to rewrite a network project file, but used to add data to or change data in the internal memory of 'Pro-Server EX'.

■ Cache Buffer

When caching device data, 'Pro-Server EX' manages multiple devices as a whole. The unit of the management is called "cache buffer".

(1) One cache buffer is comprised of multiple records.
(2) One record can be specified by direct specification of addresses of consecutive multiple devices, by symbol specification, or by group symbol specification.
(3) You can assign a unique name to each cache buffer.

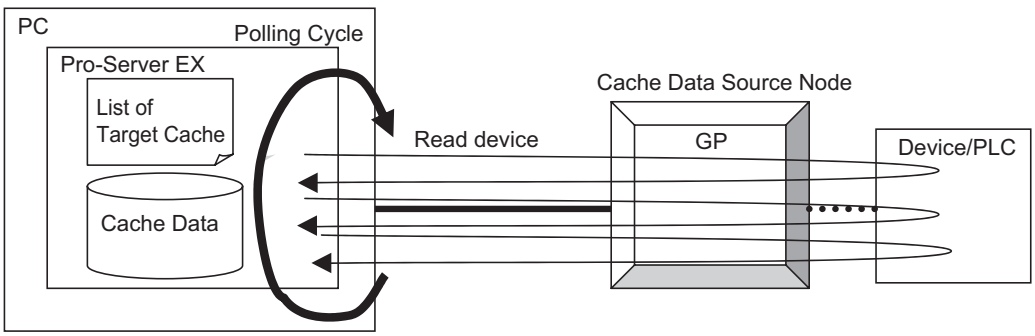
NOTE
- For cache buffer registration, the following two methods are available:
  (1) Registration using 'Pro-Studio EX' (Create a cache buffer in "Device Cache" on the feature screen, and register it in a network project file.)
  (2) Registration using API

■ Cache buffer updating procedure

To update a cache buffer, "Polling" and "Constant monitoring" methods are available.

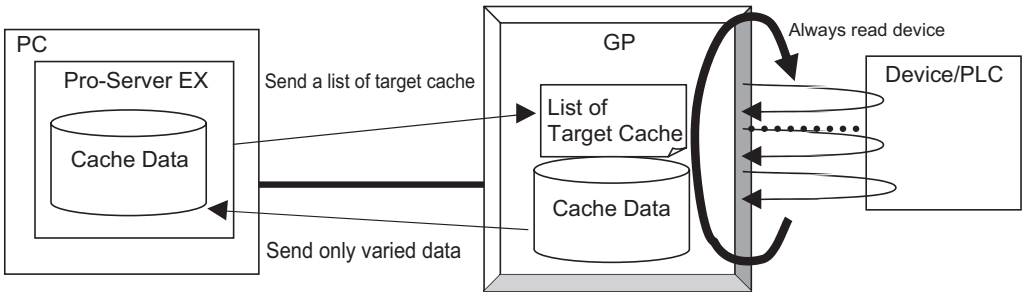
◆ The principle of polling method

According to a list of target devices in the cache buffer, 'Pro-Server EX' reads device data to update the cache buffer when the cycle specified in cache buffer registration is reached.



◆ The principle of constant monitoring method

At the start of cache buffer updating, 'Pro-Server EX' sends a list of target devices to a data source node. According to the list, the data source node constantly reads device data (as fast as possible), and sends only changed data to 'Pro-Server EX'.
'Pro-Server EX' receives the data, and handles it as cache data.



NOTE
- When the cache data source node is in the GP Series, the constant monitoring method cannot be used.

■ Selecting constant monitoring method or polling method

If a large volume of device data are monitored with the constant monitoring method, then 'Pro-Server EX' is engaged in monitoring, resulting in deterioration of the whole system performance.

To prevent this, it is recommended to select the constant monitoring method only for highly-urgent items, and to use the polling method for other items.

With the polling method, the cache buffer may not be updated according to the update cycle, depending on your PC or network conditions, types of Device/PLCs, and performance of your system. In this case, use Direct Read APIs.

As standard data volume acceptable with each method, the constant monitoring method can handle up to tens of bytes to hundreds of bytes, and the polling method can handle up to several kilobytes. For a larger data volume, use Direct Read APIs.

Note that the allowable number of bytes varies depending on performance of your system.

■ Starting and Stopping Caching

'Pro-Server EX' caching start/stop timing is described below.

(1) Caching starts or stops by cache buffer.

(2) To register a cache buffer in a network project file with 'Pro-Studio EX', the following three types of registration methods can be selected for each cache buffer. The caching start timing for each method is as follows.

1) At start of 'Pro-Server EX'

After 'Pro-Server EX' starts and a network project is loaded, 'Pro-Server EX' starts caching.

When a network project is reloaded, 'Pro-Server EX' also starts caching.

2) Starting caching automatically when a pre-registered device is read

When a Device Read API is issued for a cache device registered in the cache buffer, 'Pro-Server EX' starts caching.

Even if reading is executed for some of the devices registered in the cache buffer, 'Pro-Server EX' starts caching for all registered devices.

Caching can be started by all the reading methods as well as Device Read APIs. (For example, when a device is specified as a data source for a data transfer function, or when a device is subjected to start condition check, caching starts.)

However, only when caching is started with the method 2), 'Pro-Server EX' stops caching if there is no access to the target device in the cache buffer for a specified period.

3) Starting caching with a program using Cache Buffer Start API (PS_StartCache)

(3) In the following conditions, 'Pro-Server EX' stops caching.

1) When 'Pro-Server EX' is closed, the cache buffer stops, and discards cache data.

2) Immediately before a network project is reloaded, the cache buffer stops, and discards cache data.

3) When the function of "Automatically start when a registered device is read" is enabled, and the cache buffer is not accessed within a specified stop time after start of caching, the cache buffer stops. (Cache data will not be discarded.)

4) When the cache buffer is stopped with a program by using the Cache Stop API (PS_StopCache).

26.1.4 Group Access

Some APIs use a group symbol to specify a device address.

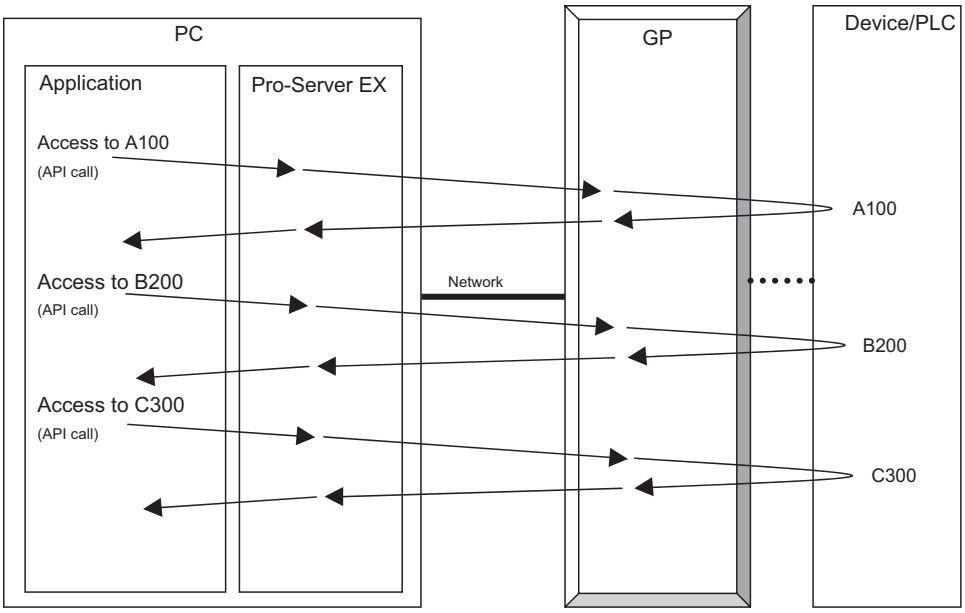
With a group symbol, 'Pro-Server EX' can efficiently access multiple devices with a single call of an API.

NOTE

- When 'Pro-Server EX' accesses devices by using a group symbol comprised of multiple devices, each access speed becomes high, and 'Pro-Server EX' and GP internally optimize the processing. Therefore, you cannot specify the device access order. (The registration order of symbols in group symbol registration does not mean the access order.)
  If an access error occurs with any one of the multiple devices, the processing will stop. 'Pro-Server EX' recognizes it as the whole group access error, and will not execute access to the remaining devices.
- The maximum group symbol data size available with a single call of an API is 1 Mbyte.

◆ When calling API individually for each device:

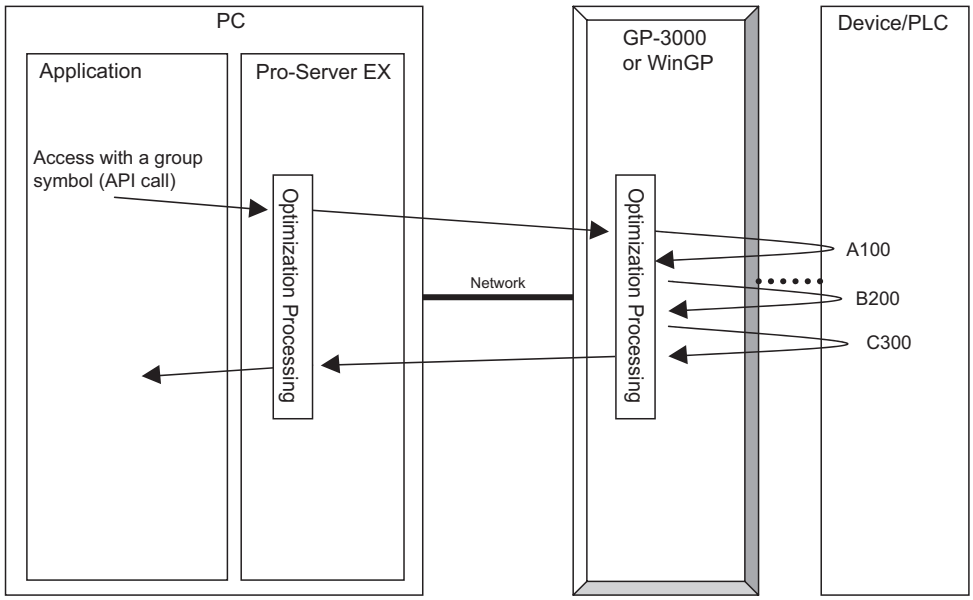
Every time the API is called, 'Pro-Server EX' communicates with the device.


PC
Application
Pro-Server EX
Access to A100
(API call)
Access to B200
(API call)
Access to C300
(API call)
Network
GP
Device/PLC
A100
B200
C300

◆ When accessing group symbols

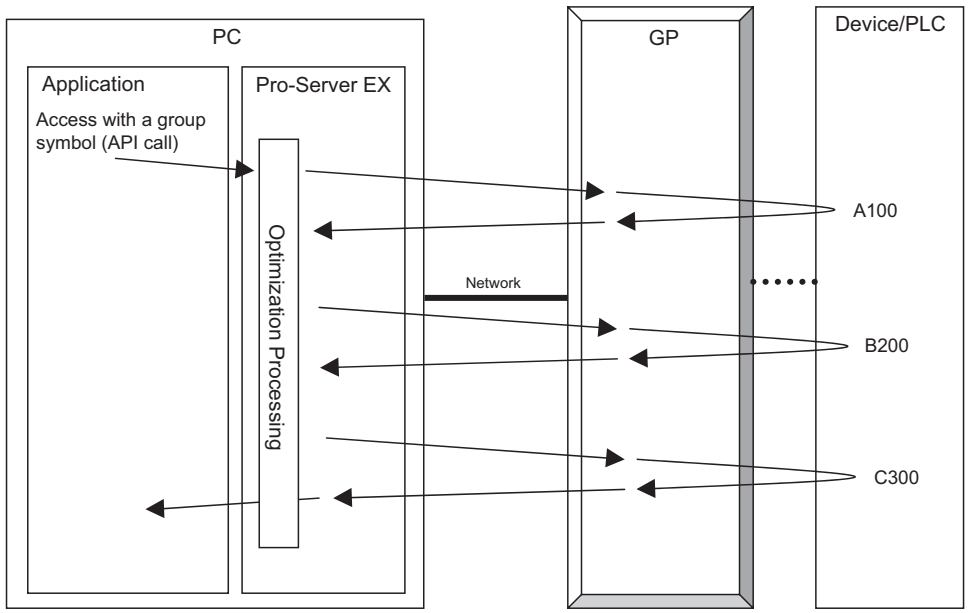
The processing varies depending on whether the target node is in the GP3000 Series, WinGP or GP Series.

- For GP3000 Series node or WinGP node
  'Pro-Server EX' sends a request for the GP3000 Series node or WinGP node only once. The GP3000 Series node or WinGP node internally divides the request to access each device separately. Thus, 'Pro-Server EX' can efficiently communicate with the devices on the network.



- For GP Series node

  The API is called only once, and 'Pro-Server EX' internally divides the request to access each GP Series node separately. However, if the group has several consecutive symbols, 'Pro-Server EX' accesses these symbols at once.



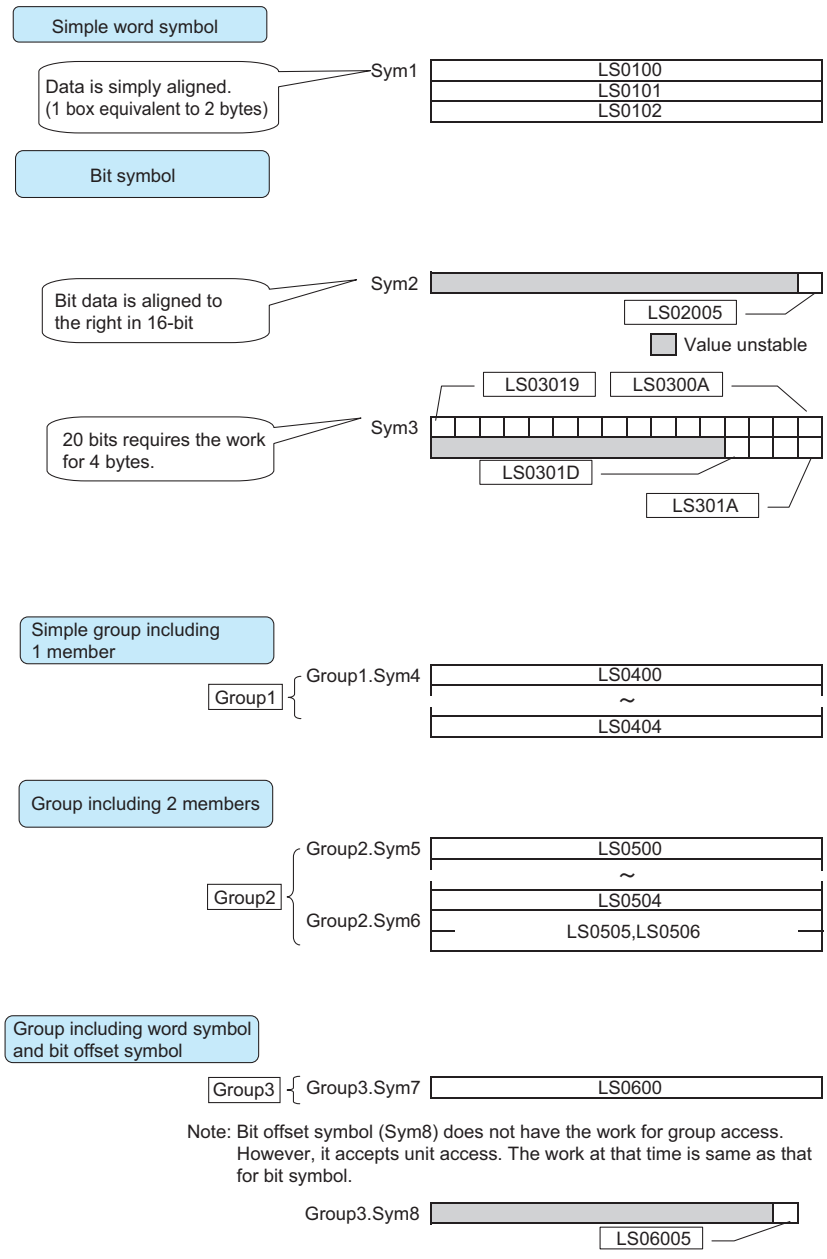
■ Data structure for group symbol access

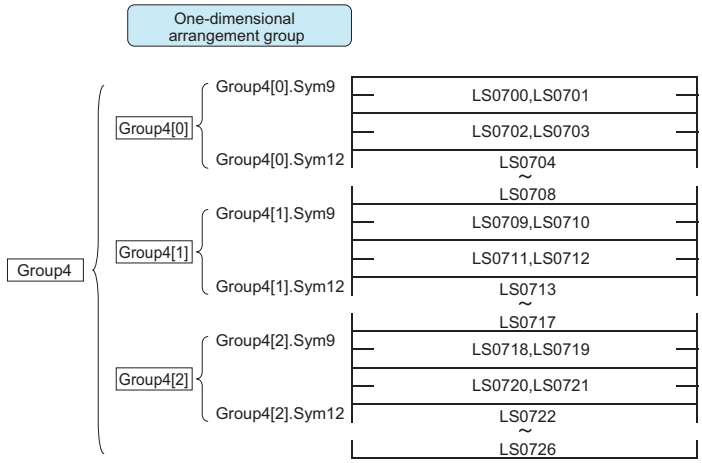
When 'Pro-Server EX' accesses devices via a group symbol, the data buffer structure varies depending on the symbol type or size of the group. The data buffer structure by group symbol type is as follows:

Group symbol data type	Secured data size
Bit Data	• For bit symbol Data buffer is secured in multiples of 16 bits. • For bit offset symbol No data buffer is secured.
16-bit(Signed)Data	Data buffer of 2 bytes/device is secured. Binary value is used.
16-bit(Unsigned)Data	
16-bit(HEX)Data	
16-bit(BCD)Data	Data buffer of 2 bytes/device is secured. During access to a device, 'Pro-Server EX' executes BCD-Binary conversion.
32-bit(Signed)Data	Data buffer of 4 bytes/device is secured. Binary value is used.
32-bit(Unsigned)Data	
32-bit(HEX)Data	
32-bit(BCD)Data	Data buffer of 4 bytes/device is secured. During access to a device, 'Pro-Server EX' executes BCD-Binary conversion.

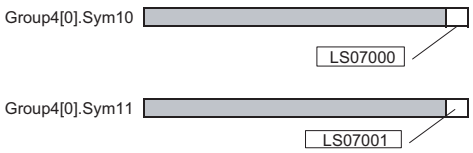
Group symbol data type	Secured data size
Single-precision floating point	Data buffer of 4 bytes/device is secured. The value is handled as a single-precision floating point value.
Double-precision floating point	Data buffer of 8 bytes/device is secured. The value is handled as a single-precision floating point value.
Character string data	Data buffer of 1 byte/character is secured. The data is handled as a NULL-terminated character string.

Examples of the data buffer structures are shown below.

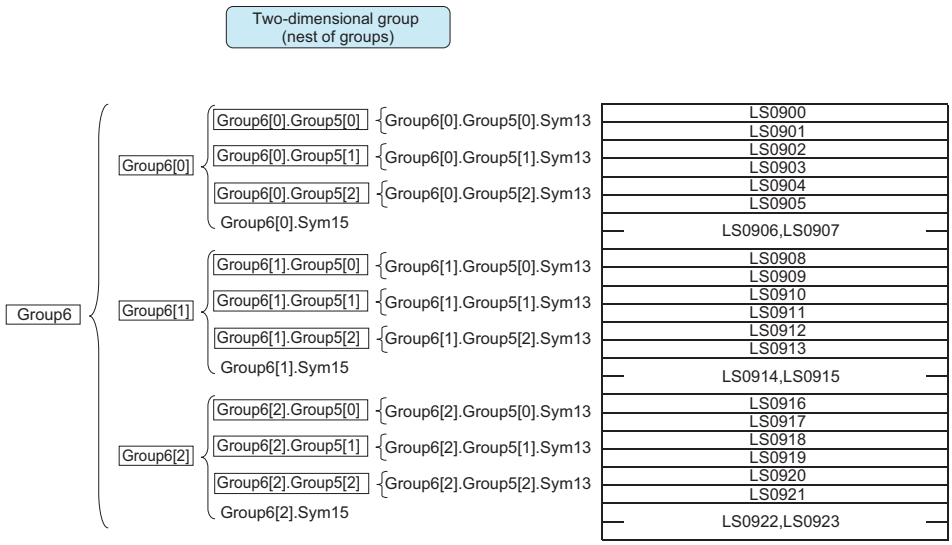




Note: Bit offset symbols (Sym10, Sym11) do not have the work for group access. However, they accept unit access. The works at that time are same as that for bit symbol.

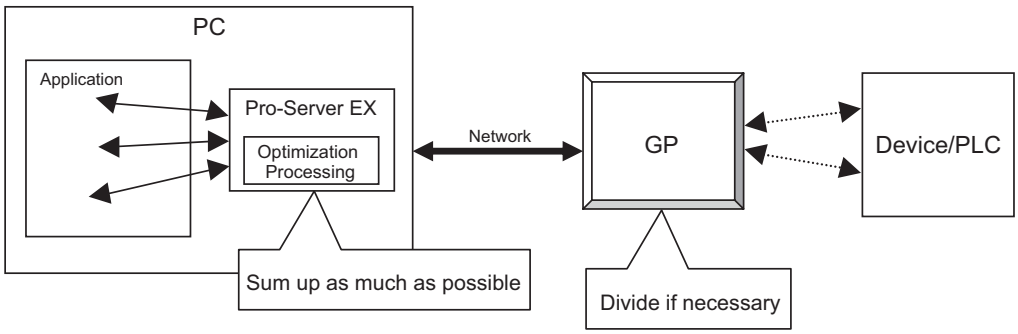


Device addresses for Group4[1].Sym10 and Group4[1].Sym11 are LS07090 and LS07091 respectively.
Device addresses for Group4[2].Sym10 and Group4[2].Sym11 are LS0718 and LS07181 respectively.



26.1.5 Queuing Access

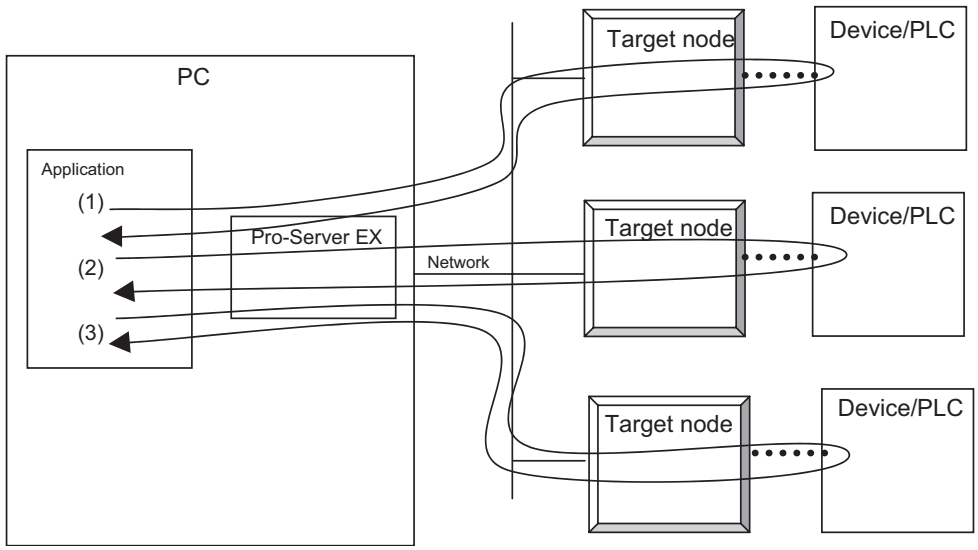
'Pro-Server EX' stores a device access request every time an API is called, and then optimizes the stored requests to access individual devices at once.



The principle of queuing access

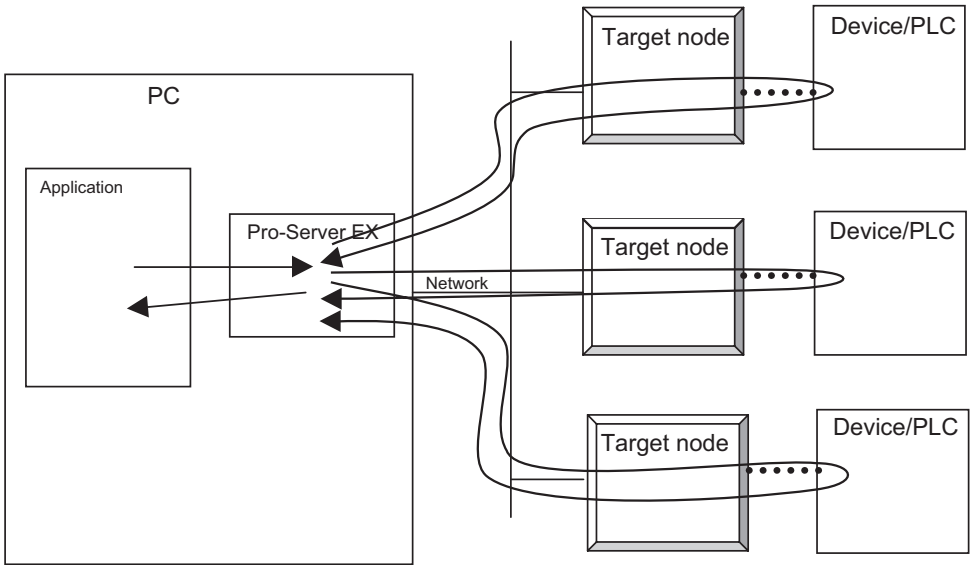
◆ Simple API access

'Pro-Server EX' executes sequential processing.



◆ Queuing access

'Pro-Server EX' executes parallel processing for individual nodes.



Procedures for use

(1) Declare start of queuing access. (Call BeginQueuingRead() or BeginQueungWrite().)

(2) Call a Device Read or Device Write API.
(For example, call ReadDevice16() or WriteDevice16().)
If the argument is normal, the API is returned soon, and 'Pro-Server EX' stores the device access request only. This step is called "Access request registration".

(3) To execute the stored device access request actually, call ExecuteQueuingAccess(). In this step, 'Pro-Server EX' optimizes the device access request, and tries to communicate with the devices efficiently.
If 'Pro-Server EX' successfully accesses all specified devices, ExecuteQueuingAccess() returns a success code. If 'Pro-Server EX' fails to access any device, on the other hand, ExecuteQueuingAccess() returns an access error code.
If you wish to know whether each device access request has been successfully executed or not, call IsQueuingAcceessSucceeded() to check the result.

NOTE

- When using queuing access, you cannot register read access and write access simultaneously. For example, after declaration of start of queuing access for read access, write access cannot be registered. Also, after declaration of start of queuing access for write access, read access cannot be registered.
  However, since queuing access is registered for each Pro-Server handle, you can register write access and read access separately for different Pro-Server handles.
- Once an access request is registered, you need not re-register it when you try to access the same device with the same method.
  Since 'Pro-Server EX' stores an access request per Pro-Server handle, it will be executed repeatedly based on the stored data, every time ExecuteQueuingAccess() is called.
  Access request registration memory will be cleared in the following cases:
  (1) When a stored Pro-Server handle is discarded.
  (2) When new queuing access registration is started.
  (3) When existing queuing access registration is cancelled (CancelQueuingAccess() is called). If a function other than Converting error code into character string(EasyLoadErrorMessage etc.) is executed after execution of ExecuteQueuingAccess(), 'Pro-Server EX' cancels existing queuing data, and starts new queuing access registration.
- During "Access request registration", 'Pro-Server EX' stores the access data buffer address (address only, excluding data).
  Therefore, after "Access request registration", the data buffer must remain in the address specified in "Access request registration", until ExecuteQueuingAccess() returns a value after it is called. Otherwise, 'Pro-Server EX' will access an invalid address, causing a fatal error.
  Also, when queuing qccess is used again, the data buffer must remain in the address specified in "access request registration".

26.1.6 Bit Data Access

To access bit devices, 'Pro-Server EX' provides the following three types of bit data handling methods:

(1) Handling bit data in multiples of 16 bits: Bit devices are handled as bit strings in multiples of 16 bits.
A specified quantity of bit data are stored and used from bit D0 (right end).
Even if only one device is specified, a 16-bit data buffer is required. Data buffers are required in multiples of 16 bits, depending on the specified number of devices.

(Example) Data buffer storing order for 20 bit devices

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

< Applicable API >
When data type "1" (EASY_AppKind_Bit) is specified for ReadDeviceBit/WriteDeviceBit(), ReadDevice/ WriteDevice() or ReadDeviceVariant/WriteDeviceVariant();

When a bit symbol, or a group including a bit symbol is specified for ReadSymbol/WriteSymbol()

(2) Handling bit data as Variant BOOL data: One bit is handled as Variant BOOL data.
The data buffer handles one piece of Variant BOOL data for one bit. BOOL data alignments as many as the specified number of devices are provided.

< Applicable API >
When data type "0x201" (EASY_AppKind_BOOL) is specified for ReadDeviceVariant/WriteDeviceVariant();
When a bit symbol, or a group including a bit symbol is specified for ReadSymbolVariant/WriteSymbolVariant()

(3) Handling bit offset symbol for group symbol access
If you access a device by directly specifying a bit offset symbol, the data buffer handles "Strings in multiples of 16 bits", or "Variant BOOL data", as described in the above section.
However, when you access a device by using a group symbol that includes a bit offset symbol, a data area for the bit offset symbol is not secured in the data buffer.
A bit offset symbol cannot exist by itself without a word symbol, or a parent symbol. The data area is secured for this parent symbol, and you can use a part of that area for the bit offset symbol.
Refer to "26.1.4 Group Access" for more details.

26.1.7 System APIs

System APIs are intended for system control, such as starting or closing 'Pro-Server EX', loading network project files and so on.

The system APIs are classified into the following categories:

Single-Handle APIs

You can use the 'Pro-Server EX' features without specifying a Pro-Server handle.

With this method, multiple APIs cannot be simultaneously used. (If you try to use multiple APIs simultaneously, the double-call error occurs.)

Multi-Handle APIs

You can use the 'Pro-Server EX' features by specifying a Pro-Server handle.

You can use multiple APIs simultaneously by specifying different Pro-Server handles.

26.1.8 SRAM Data Access APIs

The SRAM incorporated in the GP Series stores various data depending on the GP setup and operating conditions.

The following APIs are intended to access data stored in the SRAM.

All SRAM Data Access APIs support both Single-Handle and Multi-Handle functions.

This section describes Single-Handle APIs. Multi-Handle APIs are identified with "M" at the end of each API name, and a Pro-Server handle is added to the first argument.

26.1.9 CF Card APIs

The CF card incorporated in the GP Series stores various data depending on the GP setup and operating conditions, like SRAM.

The following APIs are intended to access data stored in the CF card.

26.2 Device Access APIs

■ Single-Handle Cache Read APIs

Function	Bit data
INT WINAPI ReadDeviceBit(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	16-bit data
INT WINAPI ReadDevice16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	32-bit data
INT WINAPI ReadDevice32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
Function	16-bit BCD data
INT WINAPI ReadDeviceBCD16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	32-bit BCD data
INT WINAPI ReadDeviceBCD32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
Function	Single-precision floating point data
INT WINAPI ReadDeviceFloat(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
Function	Double-precision floating point data
INT WINAPI ReadDeviceDouble(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
Function	Character string data
INT WINAPI ReadDeviceStr(LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
Function	General-use data
INT WINAPI ReadDevice(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
Function	General-use data (Variant-type)
INT WINAPI ReadDeviceVariant(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
Function	Group symbol
INT WINAPI ReadSymbol(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
Function	Group symbol (Variant-type)
INT WINAPI ReadSymbolVariant(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

* For each parameter, please refer to "■ Parameters of read/write functions".

■ Single-Handle Direct Read APIs

Function	Bit data
INT WINAPI ReadDeviceBitD(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	16-bit data
INT WINAPI ReadDevice16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	

Function	Bit data
Function	32-bit data
INT WINAPI ReadDevice32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
Function	16-bit BCD data
INT WINAPI ReadDeviceBCD16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	32-bit BCD data
INT WINAPI ReadDeviceBCD32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
Function	Single-precision floating point data
INT WINAPI ReadDeviceFloatD(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
Function	Double-precision floating point data
INT WINAPI ReadDeviceDoubleD(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
Function	Character string data
INT WINAPI ReadDeviceStrD(LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
Function	General-use data
INT WINAPI ReadDeviceD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
Function	General-use data (Variant-type)
INT WINAPI ReadDeviceVariantD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
Function	Group symbol
INT WINAPI ReadSymbolD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
Function	Group symbol (Variant-type)
INT WINAPI ReadSymbolVariantD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

* For each parameter, please refer to "■ Parameters of read/write functions".

■ Single-Handle Direct Write APIs

Function	Bit data
INT WINAPI WriteDeviceBitD(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	16-bit data
INT WINAPI WriteDevice16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	32-bit data
INT WINAPI WriteDevice32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
Function	16-bit BCD data
INT WINAPI WriteDeviceBCD16D(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	32-bit BCD data

Function	Bit data
INT WINAPI WriteDeviceBCD32D(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
Function	Single-precision floating point data
INT WINAPI WriteDeviceFloatD(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
Function	Double-precision floating point data
INT WINAPI WriteDeviceDoubleD(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
Function	Character string data
INT WINAPI WriteDeviceStrD(LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
Function	General-use data
INT WINAPI WriteDeviceD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
Function	General-use data (Variant-type)
INT WINAPI WriteDeviceVariantD(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
Function	Group symbol
INT WINAPI WriteSymbolD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
Function	Group symbol (Variant-type)
INT WINAPI WriteSymbolVariantD(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

* For each parameter, please refer to "■ Parameters of read/write functions".

■ Single-Handle Write APIs with Cache Refresh after Writing

Function	Bit data
INT WINAPI WriteDeviceBit(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	16-bit data
INT WINAPI WriteDevice16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	32-bit data
INT WINAPI WriteDevice32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
Function	16-bit BCD data
INT WINAPI WriteDeviceBCD16(LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	32-bit BCD data
INT WINAPI WriteDeviceBCD32(LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
Function	Single-precision floating point data
INT WINAPI WriteDeviceFloat(LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
Function	Double-precision floating point data
INT WINAPI WriteDeviceDouble(LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
Function	Character string data

Function	Bit data
INT WINAPI WriteDeviceStr(LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
Function	General-use data
INT WINAPI WriteDevice(LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
Function	General-use data (Variant-type)
INT WINAPI WriteDeviceVariant(LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
Function	Group symbol
INT WINAPI WriteSymbol(LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
Function	Group symbol (Variant-type)
INT WINAPI WriteSymbolVariant(LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

* For each parameter, please refer to "■ Parameters of read/write functions".

■ Multi-Handle Cache Read APIs

Function	Bit data
INT WINAPI ReadDeviceBitM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	16-bit data
INT WINAPI ReadDevice16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	32-bit data
INT WINAPI ReadDevice32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
Function	16-bit BCD data
INT WINAPI ReadDeviceBCD16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	32-bit BCD data
INT WINAPI ReadDeviceBCD32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
Function	Single-precision floating point data
INT WINAPI ReadDeviceFloatM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
Function	Double-precision floating point data
INT WINAPI ReadDeviceDoubleM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
Function	Character string data
INT WINAPI ReadDeviceStrM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
Function	General-use data

Function	Bit data
INT WINAPI ReadDeviceM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
Function	General-use data (Variant-type)
INT WINAPI ReadDeviceVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
Function	Group symbol
INT WINAPI ReadSymbolM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
Function	Group symbol (Variant-type)
INT WINAPI ReadSymbolVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

* For each parameter, please refer to "■ Parameters of read/write functions".

■ Multi-Handle Direct Read APIs

Function	Bit data
INT WINAPI ReadDeviceBitDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	16-bit data
INT WINAPI ReadDevice16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	32-bit data
INT WINAPI ReadDevice32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
Function	16-bit BCD data
INT WINAPI ReadDeviceBCD16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* owData,WORD wCount);	
Function	32-bit BCD data
INT WINAPI ReadDeviceBCD32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* odwData,WORD wCount);	
Function	Single-precision floating point data
INT WINAPI ReadDeviceFloatDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* oflData,WORD wCount);	
Function	Double-precision floating point data
INT WINAPI ReadDeviceDoubleDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* odbData,WORD wCount);	
Function	Character string data
INT WINAPI ReadDeviceStrDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPSTR psData,WORD wCount);	
Function	General-use data
INT WINAPI ReadDeviceDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
Function	General-use data (Variant-type)

Function	Bit data
INT WINAPI ReadDeviceVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
Function	Group symbol
INT WINAPI ReadSymbolDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID oReadBufferData);	
Function	Group symbol (Variant-type)
INT WINAPI ReadSymbolVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

* For each parameter, please refer to "■ Parameters of read/write functions".

■ Multi-Handle Direct Write APIs

Function	Bit data
INT WINAPI WriteDeviceBitDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	16-bit data
INT WINAPI WriteDevice16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	32-bit data
INT WINAPI WriteDevice32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
Function	16-bit BCD data
INT WINAPI WriteDeviceBCD16DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	32-bit BCD data
INT WINAPI WriteDeviceBCD32DM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
Function	Single-precision floating point data
INT WINAPI WriteDeviceFloatDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
Function	Double-precision floating point data
INT WINAPI WriteDeviceDoubleDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
Function	Character string data
INT WINAPI WriteDeviceStrDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
Function	General-use data
INT WINAPI WriteDeviceDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
Function	General-use data (Variant-type)
INT WINAPI WriteDeviceVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	

Function	Bit data
Function	Group symbol
INT WINAPI WriteSymbolDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
Function	Group symbol (Variant-type)
INT WINAPI WriteSymbolVariantDM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

* For each parameter, please refer to "■ Parameters of read/write functions".

■ Multi-Handle Write APIs with Cache Refresh after Writing

Function	Bit data
INT WINAPI WriteDeviceBitM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	16-bit data
INT WINAPI WriteDevice16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	32-bit data
INT WINAPI WriteDevice32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
Function	16-bit BCD data
INT WINAPI WriteDeviceBCD16M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,WORD* pwData,WORD wCount);	
Function	32-bit BCD data
INT WINAPI WriteDeviceBCD32M(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DWORD* pdwData,WORD wCount);	
Function	Single-precision floating point data
INT WINAPI WriteDeviceFloatM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,FLOAT* pflData,WORD wCount);	
Function	Double-precision floating point data
INT WINAPI WriteDeviceDoubleM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,DOUBLE* pdbData,WORD wCount);	
Function	Character string data
INT WINAPI WriteDeviceStrM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPCSTR psData,WORD wCount);	
Function	General-use data
INT WINAPI WriteDeviceM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVOID pData,WORD wCount,WORD wAppKind);	
Function	General-use data (Variant-type)
INT WINAPI WriteDeviceVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sDeviceName,LPVARIANT pData,WORD wCount,WORD wAppKind);	
Function	Group symbol

Function	Bit data
INT WINAPI WriteSymbolM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVOID pWriteBufferData);	
Function	Group symbol (Variant-type)
INT WINAPI WriteSymbolVariantM(HANDLE hProServer,LPCSTR sNodeName,LPCSTR sSymbolName,LPVARIANT pData);	

* For each parameter, please refer to "■ Parameters of read/write functions".

■ Parameters of read/write functions

< Argument >

bsNodeName : Pointer to node name (character string)

Specify the entry node name or the IP address registered in 'Pro-Studio EX' directly.

Ex. 1) When specifying node name: "AGP"

Ex. 2) When specifying IP address directly: "192.9.201.1"

bsDeviceName : Pointer to the symbol (character string) subjected to Read/Write function

Specify the symbol name or the device address registered in 'Pro-Studio EX' directly.

Ex. 1) When specifying symbol name: "SWITCH1"

Ex. 2) When specifying device address directly: "M100"

Function	Symbol data type							
	Bit	16 bits		32 bits		Float	Double	String
		S/U/HEX	BCD	S/U/HEX	BCD			
XXXDeviceBit	0	-	-	-	-	-	-	-
XXXDevice16	-	0	-	-	-	-	-	-
XXXDevice32	-	-	-	0	-	-	-	-
XXXDeviceBCD16	-	-	0	-	-	-	-	-
XXXDeviceBCD32	-	-	-	-	0	-	-	-
XXXDeviceFloat	-	-	-	-	-	0	-	-
XXXDeviceDouble	-	-	-	-	-	-	0	-
XXXDeviceStr	-	-	-	-	-	-	-	0
XXXDevice	0	0	0	0	0	0	0	0

pxxData : Pointer to read/write target data

Accessible data types and corresponding argument types are listed below.

Accessible data type	Argument type
Bit data	WORD * pwData
16-bit data	WORD * pwData
32-bit data	DWORD * pdwData
16-bit BCD data	WORD * pwData
32-bit BCD data	DWORD * pdwData
Single-precision floating point data	FLOAT * pflData
Double-precision floating point data	DOUBLE * pdbData
Character string data	LPTSTR psData
General-use data	LPVOID pData
General-use data (for VB)	LPVARIANT pData

wCount : Quantity of read/write target data

With the Read/WriteDeviceStr function, character string data is counted as the number of bytes. For a device symbol with 16-bit width, specify multiples of two characters; for a device symbol with 32-bit width, specify multiples of four characters.
The maximum data quantities subjected to read/write functions are as follows:

Accessible data type	Read	Write
Bit data	255	255
16-bit data	1020	1020
32-bit data	510	510
16-bit BCD data	1020	1020
32-bit BCD data	510	510
Single-precision floating point data	510	510
Double-precision floating point data	255	255
Character string data	255 characters (single-byte)	255 characters (single-byte)

wAppKind : Data type specification

Value	Data type	Value	Data type
1	Bit	7	Unsigned 32 bits
2	Signed 16 bits	8	HEX 32 bits
3	Unsigned 16 bits	9	BCD 32 bits
4	HEX 16 bits	10	Float
5	BCD 16 bits	11	Double
6	Signed 32 bits	12	String

With the Read/Write Device function, the data type is specified by parameter. Therefore, the data type can be dynamically changed.

< Return value >

Normal end: 0

Abnormal end: Error code

< Special Note >

When using the Read/WriteDeviceBit function:

pwData stores a quantity of data specified with wCount, consecutively from the D0 bit.

Example: When wCount is "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

When handing several consecutive bit strings, it is more effective to use the read/write functions that handle data in multiples of 16 bits or 32 bits (Read/WriteDevice16 or Read/WriteDevice32), rather than Read/WriteDeviceBit.

The bit indicated with "*" (asterisk) stores an undefined value. Mask these areas in your application program.

When using the Read/WriteDeviceBCD16 or Read/WriteDeviceBCD32 functions:

If the target device/PLC handles BCD data, you can use these functions. However, the data passed with these functions (contents of pxxData) are handled as binary data, not BCD data. ('Pro-Server EX' internally executes BCD conversion.) A negative value cannot be handled.

Function	Decimal expression	Hexadecimal expression
Read/WriteDeviceBCD16	0 to 9999	0000 to 270F
Read/WriteDeviceBCD32	0 to 99999999	00000000 to 05F5E0FF

When using the string data functions:

To receive character string data for variables, secure sufficient data storing area.

26.3 Cache Buffer Control APIs

Function	Creating cache buffer

To increase the device read processing speed, 'Pro-Server EX' incorporates the device data caching function (with copy function). This API is used to create a cache buffer.
This API only defines a cache buffer. To define which device to cache, use PS_EntryCacheRecord().

Single
INT WINAPI PS_CreateCache(LPCSTR sCacheName, DWORD dwPollingTime);
Multi
INT WINAPI PS_CreateCacheM(HANDLE hProServer, LPCSTR sCacheName, DWORD dwPollingTime);

Argument
sCacheName: (In) Cache buffer name
dwPollingTime: (In) To select the constant monitoring method, specify "0".
The cache buffer is updated as fast as possible.
If you specify any value other than "0", the polling method is selected.
Specify the polling cycle (cache updating cycle) by the millisecond.

Return value
Normal end: 0
Abnormal end: Error code

Special Note
- Up to 1000 cache buffers can be created for a single 'Pro-Server EX' program.
- You can directly use the cache buffer which has been registered when creating a network project file with 'Pro-Studio EX'. It is unnecessary to re-create it with this API.

Function	Registering record into cache buffer

Registers a caching device (cache source device) into the cache buffer created with PS_CreateCache().

For a GP Series node or Pro-Server EX node, 'Pro-Server EX' does not support the constant monitoring method to update a cache buffer.
Therefore, if you specify a GP Series node or Pro-Server EX node with this API for a cache buffer subjected to the constant monitoring method (if dwPollingTime is set to "0" when a cache buffer is created with PS_CreateCache()), an error occurs.

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

Argument

sCacheName: (In) Cache buffer name

Register a cache source device into the cache buffer specified with this name.

sNodeName: (In) Entry node name with cache source Device/PLC name

sDevice:(In) Cache source device

To specify a cache source device, you can directly specify the device address, or specify a symbol or group registered with 'Pro-Studio EX'. If you specify a group, multiple symbols can be registered at once.

wAppKind: (In) Source device data type

Available data types vary depending on the cache source device designation method.

a) When device address of cache source device is directly specified:

Specify a data type (1 to 12) available with 'Pro-Server EX'. "0" cannot be specified.

Value	Data type	Value	Data type
1	Bit	7	32 bits, Unsigned decimal
2	16 bits, Signed decimal	8	32 bits, Hexadecimal
3	16 bits, Unsigned decimal	9	32 bits, BCD
4	16 bits, Hexadecimal	10	Single-precision floating point
5	16 bits, BCD	11	Double-precision floating point
6	32 bits, Signed decimal	12	Character string

b) When symbol is specified for cache source device:

Specify a data type (0 to 12) available with 'Pro-Server EX'. If you specify "0", the symbol type specified in symbol definition is used.

c) When group is specified for cache source device:

Fixed to "0".

The symbol type is registered for all symbols in the specified group.

wCount: (In) Device data quantity subjected to caching

Available values vary depending on the cache source device specification method.

a) When device address of cache source device is directly specified:

Data quantity (1 to 1020) according to the device type can be used. (The maximum value varies depending on the device type.)

b) When symbol is specified for cache source device:

If you specify "0", the quantity specified in symbol definition is used.

If you specify any value other than 0, data quantity (1 to 1020) according to the device type can be used. (The maximum value varies depending on the device type.)

c) When group is specified for cache source device:

Fixed to "0".

All symbols in the specified group are subjected to caching.

Return value

Normal end: 0

Abnormal end: Error code

Special Note

Function	Starting caching

Starts caching.

Single
INT WINAPI PS_StartCache(LPCSTR sCacheName);
Multi
INT WINAPI PS_StartCacheM(HANDLE hProServer, LPCSTR sCacheName);

Argument	**Return value**
sCacheName: (In) Name of cache buffer to start A cache buffer name registered with 'Pro-Studio EX' can be also specified.	Normal end: 0 Abnormal end: Error code

Special Note

Function	Stopping caching

Temporarily stops caching.
Caching stops, but definition of the cache buffer is retained.
To restart caching, call PS_StartCache().

Single
INT WINAPI PS_StopCache(LPCSTR sCacheName);
Multi
INT WINAPI PS_StopCacheM(HANDLE hProServer, LPCSTR sCacheName);

Argument	**Return value**
sCacheName: (In) Name of cache buffer to stop A cache buffer name registered with 'Pro-Studio EX' can be also specified.	Normal end: 0 Abnormal end: Error code

Special Note

Function	Checking caching status

Checks caching status.

Single
INT WINAPI PS_GetCacheStatus(LPCSTR sCacheName);
Multi
INT WINAPI PS_GetCacheStatusM(HANDLE hProServer, LPCSTR sCacheName);

Argument	**Return value**
sCacheName: (In) Name of cache buffer to be checked A cache buffer name registered with 'Pro-Studio EX' can be also specified.	0 : The cache buffer has been created, but not started yet. 1: Caching in progress 2: Caching under suspension XX: Error code

Special Note

Function	Discarding cache buffer

Stops caching, and discards the cache buffer.

Single
INT WINAPI PS_DestroyCache(LPCSTR sCacheName);
Multi
INT WINAPI PS_DestroyCacheM(HANDLE hProServer, LPCSTR sCacheName);

Argument	**Return value**
sCacheName: (In) Name of cache buffer to be discarded A cache buffer name registered with 'Pro-Studio EX' can be also specified.	Normal end: 0 Abnormal end: Error code

Function	Setting cache update notification function

Sets the function to notify cache buffer update status to a specified window.

When a device is cache-read from an application, there will be no change without updating the cache data even if the device is frequently cache-read.
'Pro-Server EX' can send a message to a specified window, when cache data is updated (when at least one target device has a change with the constant monitoring method, or when one polling cycle is completed with the polling method).
If your system is built so as to execute cache-reading of a device after receiving this message, the system efficiency can be improved.
This API allows you to set "Target cache buffer name", "Window to receive the message", and "Contents of the message" in 'Pro-Server EX'.
After these settings are normally completed, the API returns the ID that identifies the currently-set notification function.

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

Argument	**Return value**
sCacheName: (In) Cache buffer name A cache buffer name registered with 'Pro-Studio EX' can be also specified. hWnd: (In) Handle for the window to receive the message message: (In) Message ID to be sent to the window wParam: (In) WPARAM value to be sent to the window together with message ID LParam: (In) LPARAM value to be sent to the window together with message ID ohCacheNotifyID: (Out) Returns the ID that identifies the currently set notification function.	Normal end: 0 Abnormal end: Error code

Special Note
If the returned handle is not necessary, discard it with PS_KillNotifyFromCache().
After the cache buffer is updated, call PostMessage() to send the message (specified with the second argument), wParam value (specified with the third argument), and LParam value (specified with the fourth argument) to the target window (hWnd).
For details of PostMessage(), refer to the Windows API Manual.

Function	Accepting next cache update notification

Accepts the next cache update notification.

'Pro-Server EX' provides the function to send a message to a specified window when a cache buffer is updated. However, once this notification function is executed, 'Pro-Server EX' will not send a message until this API is called again, even if the cache buffer is updated next. This is because in case it has taken a long time in processing with the notification routine, a multiple-call error can occur with the relevant routine when 'Pro-Server EX' sends the next cache update message. (If the notification routine receives the next message before completion of the processing, a multiple-call error occurs with the routine.)
To prevent this error, this API explicitly informs 'Pro-Server EX' that it can send the next message.
By calling this API at the end of the processing of the notification routine, you can build a system that enables continuous processing every time a cache buffer is updated.

Single
INT WINAPI PS_AcceptNextNotifyFromCache(HANDLE hCacheNotifyID);
Multi
INT WINAPI PS_AcceptNextNotifyFromCacheM(HANDLE hProServer, HANDLE hCacheNotifyID);

Argument	**Return value**
hCacheNotifyID: (In) ID of next message acceptance notification function ID obtained with PS_SetNotifyFromCache()	Normal end: 0 Abnormal end: Error code

Special Note

Function	Canceling cache update notification

Cancels the function for sending a cache buffer update message to a specified window.

After cancellation, 'Pro-Server EX' will not send a cache buffer update message to the relevant window, even if the cache buffer related with hCacheNotifyID is updated.

Single
INT WINAPI PS_KillNotifyFromCache(HANDLE hCacheNotifyID);
Multi
INT WINAPI PS_KillNotifyFromCacheM(HANDLE hProServer, HANDLE hCacheNotifyID);

Argument	**Return value**
hCacheNotifyID: (In) ID of the notification function to be canceled ID obtained with PS_SetNotifyFromCache()	Normal end: 0 Abnormal end: Error code

Special Note
This API will not fetch and discard a message sent from 'Pro-Server EX', even if the message remains in the window. Therefore, if 'Pro-Server EX' has sent a message to a window and the application has not fetched the message from the window before this API is called, the application can fetch the message from the window even after this API is called. (Depending on the timing, the notification routine may be called even after this API is called.)

Function	Acquiring cache buffer update count

Returns a cache buffer update count.

By monitoring the update count on the program, you can check if a cache buffer has been updated or not.
Using this function, you can omit unnecessary calls of device cache read APIs. (Even if a device cache read API is called for a device with no change, the value will not be changed.)

Single
INT WINAPI PS_GetUpdateCounter(LPCSTR sCacheName, DWORD* odwCount);
Multi
INT WINAPI PS_GetUpdateCounterM(HANDLE hProServer, LPCSTR sCacheName, DWORD* odwCount);

Argument	**Return value**
sCacheName: (In) Name of cache buffer to be monitored A cache buffer name registered with 'Pro-Studio EX' can be also specified. odwCount: (Out) Cache buffer update count Counts the number of updates from 0 to 4294967295 endlessly. (After the count reaches 4294967295, it returns to"0".)	Normal end: 0 Abnormal end: Error code

Special Note

26.4 Queuing Access Control APIs

Function	Starting the queuing of device read request

After this API is called, 'Pro-Server EX' queues device read requests until ExecuteQueuingAccess() is called.
Queuing is executed for each Pro-Server handle.

Single
INT WINAPI BeginQueuingRead();
Multi
INT WINAPI BeginQueuingReadM(HANDLE hProServer);

Argument	**Return value** Normal end: 0 Abnormal end: Error code

Special Note
- Do not call a Device Write API until you call ExecuteQueuingAccess() after BeginQueuingRead(). After BeginQueuingRead() is called, 'Pro-Server EX' queues cache read or direct read requests. However, cache read and direct read requests cannot be queued together.
- To discard a request in queue, call CancelQueuingAccess().
- Queuing is available up to 1500 requests and a data size of 1 Mbyte.

Function	Starting the queuing of device write request

After this API is called, 'Pro-Server EX' queues device write requests until ExecuteQueuingAccess() is called.
Queuing is executed for each Pro-Server handle.

Single
INT WINAPI BeginQueuingWrite();
Multi
INT WINAPI BeginQueuingWriteM(HANDLE hProServer);

Argument	**Return value** Normal end: 0 Abnormal end: Error code

Special Note
- Do not call a Device Read API until you call ExecuteQueuingAccess() after BeginQueuingWrite(). After BeginQueuingWrite() is called, 'Pro-Server EX' queues cache write or direct write requests. However, cache write and direct write requests cannot be queued together.
- To discard a request in queue, call CancelQueuingAccess().
- Queuing is available up to 1500 requests and a data size of 1 Mbyte.

Special Note

Function	Executing device read/write request in queue

Accesses device data according to the device read/write request in queue.

Single
INT WINAPI ExecuteQueuingAccess();
Multi
INT WINAPI ExecuteQueuingAccessM(HANDLE hProServer);

Argument	**Return value** Normal end: 0 Abnormal end: Error code

Special Note

- If 'Pro-Server EX' successfully accesses all specified devices, ExecuteQueuingAccess() returns a success code. If 'Pro-Server EX' fails to access any device, on the other hand, ExecuteQueuingAccess() returns an access error code. If you wish to know whether each device access request has been successfully executed or not, call IsQueuingAcceessSucceeded() to check the result.
- You cannot register ACTIONs in queuing access.

Function	Discarding device read/write request in queue

Discards the device read/write request in queue.

Single
INT WINAPI CancelQueuingAccess();
Multi
INT WINAPI CancelQueuingAccessM(HANDLE hProServer);

Argument	**Return value** Normal end: 0 Abnormal end: Error code

Special Note

After BeginQueuingWrite() or BeginQueuingRead()is called, 'Pro-Server EX' queues device access requests until ExecuteQueuingAccess() is called.
If a request in queue becomes unnecessary for any reason, call this API. 'Pro-Server EX' discards the request in queue, and quits queuing.

Function	Checking the success or failure of device read/write request in queue

Checks whether or not each device access request has been successfully executed, after ExecuteQueuingAccess() is called.

Single
INT WINAPI IsQueuingAccessSucceeded(INT iIndex);
Multi
INT WINAPI IsQueuingAccessSucceededM(HANDLE hProServer,INT iIndex);

Argument
iIndex: (In) Number of request to be checked

After BeginQueuingWrite() or BeginQueuingRead() is called, Device Access APIs are called several times to queue device access requests until ExecuteQueuingAccess() is called. Note that you cannot know an actual device access result until execution of ExecuteQueuingAccess().
If you wish to know a result of each device access request, execute ExecuteQueuingAccess() first, and then specify the number (from 0) of the request for the target device.

Return value
XX: Error code
0: Indicates that the device access request of the specified number has been successfully executed.

Special Note
(Example)

```
BeginQueuingWrite();
        WriteDevice16("Node1","LS100",Data,10);
        WriteDevice16("Node1","LS200",Data,10);
        WriteDevice16("Node1","LS300",Data,10);
ExecuteQueuingAccess()
```

To check if the "Node1" access to "LS200" has been successfully executed, use IsQueuingAccessSucceeded(1).
If the return value is "0", this access has been successfully executed.

26.5 System APIs

Function	Creating Pro-Server handle
Obtains a Pro-Server handle for use of a Multi-Handle function. HANDLE WINAPI CreateProServerHandle();	
Argument	**Return value** Normal end: Other than 0 (Handle code) Abnormal end: 0
Special Note	

Function	Releasing Pro-Server handle
Releases an obtained Pro-Server handle. INT WINAPI DeleteProServerHandle(HANDLE hProServer);	
Argument hProServer: (In) Pro-Server handle to be released	**Return value** Normal end: 0 Abnormal end: Error code
Special Note	

Function	Loading network project file
Loads the network project file specified with the argument. Single INT WINAPI EasyLoadNetworkProject(LPCSTR sDBName,DWORD dwSetOrAdd = TRUE); Multi INT WINAPI EasyLoadNetworkProjectM(HANDLE hProServer,LPCSTR sDBName,DWORD dwSetOrAdd = TRUE);	
Argument sDBName: Specify the full path of a network project file to be loaded. dwSetOrAdd: Reserve (Fixed to "1") hProServer: Pro-Server handle	**Return value** Normal end: 0 Abnormal end: Error code
Special Note	

Function	Converting error code into character string

Converts an error code returned by each API of 'Pro-Server EX' into an error message.
EasyLoadErrorMessage() returns a multibyte character string (ASCII) as a message. EasyLoadErrorMessageW() returns a wide character string (UNICODE) as a message.

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

Argument
iErrorCode: (In) Error code returned by 'Pro-Server EX' function
osErrorMessage: (Out) Pointer to the converted character string (multibyte character string) storing area. (To call this API, secure a storing area with at least 512 bytes.)
osErrorMessage: (Out) Pointer to the converted character string (multibyte character string) storing area. (To call this API, secure a storing area with at least 1024 bytes.)

Return value
Normal end: Other than 0
Failure in character string conversion (ex. Undefined code): 0

Special Note
- This API is intended to ensure compatibility with older versions of 'Pro-Server'.
- Using EasyLoadErrorMessageEx() enables conversion into a more detailed error message. We recommend you to use EasyLoadErrorMessageEx().

Function	Converting error code into character string (with status information)

Converts an error code returned by each API of 'Pro-Server EX' into an error message.
'Pro-Server EX' then returns the error message together with the error occurrence condition and other information, if possible.
EasyLoadErrorMessage() always returns the same error message relative to a specified error code. On the other hand, EasyLoadErrorMessageEx() returns more detailed error information including a name of communication target device, error occurrence place and so on, depending on the error occurrence condition. Thus, EasyLoadErrorMessageEx() may return a different error message relative to the same error code, depending on the situation.
EasyLoadErrorMessageEx() and EasyLoadErrorMessageExM() return a multibyte character string (ASCII) as a message.
EasyLoadErrorMessageExW() and EasyLoadErrorMessageExWM() return a wide character string (UNICODE) as a message.

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

Argument
iErrorCode: (In) Error code returned by 'Pro-Server EX' function
osErrorMessage: (Out) Pointer to the converted character string (multibyte character string) storing area.(To call this API, secure a storing area with at least 1024 bytes.)
owsErrorMessage: (Out) Pointer to the converted character string (wide character string) storing area. (To call this API, secure a storing area with at least 2048 bytes.)

Return value
Normal end: Other than 0
Failure in character string conversion (ex. Undefined code): 0

Special Note
- EasyLoadErrorMessage() is used to convert an error code into a message, assuming a case where an API of 'Pro-Server EX' is called and then the API returns an error code.
- 'Pro-Server EX' can store only one piece of error status information per handle. Therefore, if you call another API between the API that causes an error and EasyLoadErrorMessage(),EasyLoadErrorMessage() will not return error status information because stored error status information is rewritten. For this reason, when using EasyLoadErrorMessageM(), you must specify the same Pro-Server handle as the handle used when the relevant API was called.

Function	Initializing Pro-Server API
Initializes a Pro-Server EX API, and declares use of the API internally. If you execute EasyInit() without starting 'Pro-Server EX', 'Pro-Server EX' will automatically start. INT WINAPI EasyInit();	
Argument	**Return value** Normal end: 0 Abnormal end: Error code
Special Note	

Function	Ending Pro-Server API
INT WINAPI EasyTerm();	
Argument	**Return value**
Special Note This API is intended to ensure compatibility with older versions of 'Pro-Server'. With 'Pro-Server EX', you need not call this API. (Even if you call this API, it will not be executed.)	

Function	Closing Pro-Server EX
Closes 'Pro-Server EX'. After calling this API, do not call any API of 'Pro-Server EX'. Before calling this API, be sure to discard Pro-Server handles etc. INT WINAPI EasyTermServer();	
Argument	**Return value** Normal end: 0 Abnormal end: Error code
Special Note	

Function	Pro-Server EX closing notice

This API allows you to know the 'Pro-Server EX' closing status.
When 'Pro-Server EX' starts closing processing, it sends a specified message to the window registered with this API by using PostMessage() of Windows API.
For details of PostMessage(), refer to Windows APIs.
When the application receives the message from the window, it recognizes that 'Pro-Server EX' will be immediately closed.

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

Argument
hReceivedWnd: (In) Window that receives a closing message.
uMessage: (In) Message ID to be sent as a closing message.
This ID will be sent to the window specified with hReceivedWnd when Pro-Server EX is being closed.
WParam: (In) WPARAM to be sent together with the message (Value of WPARAM in PostMessage())
Lparam: (In) LPARAM to be sent together with the message (Value of LPARAM in PostMessage())

Return value
Normal end: 0
Abnormal end: Error code

Special Note
This API is useful to build an application that closes at the same time when 'Pro-Server EX' is closed.
For example, if you specify the application main window for hReceivedWnd, and WM_QUIT for uMessage to call this API, 'Pro-Server EX' sends WM_QUIT to the application main window when 'Pro-Server EX' is being closed.
Generally, an application uses WM_QUIT as an application closing signal. Therefore, you can build an application that closes at the same time when 'Pro-Server EX' is closed.

Function	Inhibiting message processing

Most of the Pro-Server EX APIs (functions) process Windows messages during the processing of a function if the processing time would be long. This API can specify whether to execute or inhibit the Windows message processing.
When Windows message processing is inhibited, the relevant Windows message is stored in the message queue, and will not be processed during execution of a function.
As a result, you will not call a function over again by clicking the icon during execution of the function.
In this case, however, the processing of all the Windows messages as well as an "icon click" message, will be inhibited, and the processing of important messages for timer and window re-drawing is also disabled.
You can specify whether to execute or inhibit the processing of Windows messages for each Pro-Server EX handle.
With the default setting, message processing has been set to "Execute".

Single
INT EasySetWaitType(DWORD dwMode);
Multi
INT EasySetWaitTypeM(HANDLE hProServer,DWORD dwMode);

Argument
hProServerHandle: (In) Pro-Server handle subjected to processing mode change
dwMode: (In) To execute message processing, specify "1".
To inhibit message processing, specify "2".

Return value
Normal end: 0
Abnormal end: Error code

Special Note

Function	Acquiring message processing mode

Acquires the current message processing mode during a call of a Pro-Server EX API.
The Multi-Handle API returns the current message processing mode for each handle.

Single
INT EasyGetWaitType();
Multi
INT EasyGetWaitTypeM(HANDLE hProServerHandle);

Argument
HANDLE hProServerHandle: (In) Handle subjected to status acquisition

Return value
1: Executes message processing.
2: Inhibits message processing.

Special Note

Function	Adding log into log viewer

If a specific event ('Pro-Server EX' start/closing, error, etc.) occurs with internal processing, 'Pro-Server EX' can record the event.
You can see the recorded information through the log viewer. (See "27.5 Monitoring System Event Logs")
With this API, 'Pro-Server EX' records a specific message by using this function. This API is available for application debugging.

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

Argument

bLevel: (In) Event type

Recording all messages may result in performance deterioration. To prevent this, 'Pro-Server EX' provides a filtering function for recording messages by event type. Specify the event type that the current recording message belongs to.
The event types are listed below.

Definition	Hexadecimal value	Event type
EASY_LogLevel_SysMessage	0x01	System message
EASY_LogLevel_SysError	0x02	System error message
EASY_LogLevel_AppError	0x04	User program error message
EASY_LogLevel_AppStart	0x08	User program starting message
EASY_LogLevel_AppEnd	0x10	User program closing message
EASY_LogLevel_AppWarning	0x20	User program warning message
EASY_LogLevel_AppMessage1	0x40	User program detail message 1
EASY_LogLevel_AppMessage2	0x80	User program detail message 2

sPrompt: (In) Character string indicating event occurrence position (NULL-terminated)
sMessage: (In) Character string of the message to be recorded (NULL-terminated)

The actually recorded message is a simple combination of two character stings (sPrompt and sMessage).

Return value

Normal end: 0
Abnormal end: Error code

Special Note

Function	Clearing log from log viewer
Clears the information recorded by EasyOutputLog(). This API is available for application debugging. INT WINAPI EasyOutputLogClear();	
Argument HANDLE hProServerHandle: (In) Handle subjected to status acquisition	**Return value** Normal end: 0 Abnormal end: Error code
Special Note	

26.6 SRAM Data Access APIs

Function	Reading SRAM backup data
Reads the following data stored in the SRAM of a GP Series node, and saves the data into a file on the PC. Filing data are saved in binary format, and other types of data are saved in CSV format. INT WINAPI EasyBackupDataRead(LPCSTR sSaveFileName,LPCSTR sNodeName,INT iBackupDataType,INT iSaveMode);	

Argument
sSaveFileName: (In) File path of the file to save read data. (String pointer)
sNodeName: (In) Name of read data source node (String pointer)
Pro-Server EX nodes cannot be specified.
iSaveMode: (In) Saving mode
0: New (If a file with the same name already exists, 'Pro-Server EX' deletes the file, and overwrites it.)
1: Add (The read data is added to the end of an existing file. If there is no file to save the data, 'Pro-Server EX' creates a new file.)
Others: Reserve
iBackupDataType: (In) Type of data to be read

Value	Data source node in GP Series	Data source node in GP3000 Series or WinGP
0x0001	Filing data	Filing data
0x0002	Logging data	Sampling data of sampling group No. 1
0x0003	Line graph data	Data of all sampling groups other than sampling group No. 1
0x0004	Sampling data	
0x0005	Alarm block 1	Alarm block 1
0x0006	Alarm history or Alarm block 2	Alarm block 2
0x0007	Alarm log or Alarm block 3	Alarm block 3
0x0008	Alarm block 4	Alarm block 4
0x0009	Alarm block 5	Alarm block 5
0x000A	Alarm block 6	Alarm block 6
0x000B	Alarm block 7	Alarm block 7
0x000C	Alarm block 8	Alarm block 8
Others	(Reserve)	(Reserve)

When the data source node is in the GP3000 Series/WinGP and the data type is Alarm block 1 to 8, one alarm block stores up to three types of data (active data, history data and log data) depending on the settings of 'GP-Pro EX'. However, this API checks if the alarm block contains valid data or not according to the following order of precedence, and reads valid data if any.
(1) Alarm history
(2) Alarm log
(3) Alarm active
If there is no valid data, an error occurs.

Return value
Normal end: 0
Abnormal end: Error code

Special Note

Function	Reading extended SRAM backup data

Reads the following data stored in the SRAM of a GP Series node, and saves the data into a file on the PC.
Filing data are saved in binary format, and other types of data are saved in CSV format.
Unlike EasyBackupDataRead(), this API enables access to extended data for the GP3000 Series and WinGP.

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

Argument

sSaveFileName: (In) File path of the file to save read data. (String pointer)
sNodeName: (In) Name of read data source node (String pointer)
Pro-Server EX nodes cannot be specified.
iSaveMode: (In) Saving mode
0: New (If a file with the same name already exists, 'Pro-Server EX' deletes the file, and overwrites it.)
1: Add (The read data is added to the end of an existing file. If there is no file to save the data, 'Pro-Server EX' creates a new file.))
Others: Reserve
iBackupDataType: (In) Type of data to be read

Value	Data source node in GP Series	Data source node in GP3000 Series or WinGP
0x0001	Filing data	Filing data
0x0002	Logging data	Sampling data of sampling group No. 1
0x0003	Line graph data	Data of all sampling groups other than sampling group No. 1
0x0004	Sampling data	
0x0005	Alarm block 1	Alarm block 1
		Specify iNumber for alarm type.
0x0006	Alarm history or Alarm block 2	Alarm block 2 Specify iNumber for alarm type.
0x0007	Alarm log or Alarm block 3	Alarm block 3 Specify iNumber for alarm type.
0x0008	Alarm block 4	Alarm block 4 Specify iNumber for alarm type.
0x0009	Alarm block 5	Alarm block 5 Specify iNumber for alarm type.
0x000A	Alarm block 6	Alarm block 6 Specify iNumber for alarm type.
0x000B	Alarm block 7	Alarm block 7 Specify iNumber for alarm type.
0x000C	Alarm block 8	Alarm block 8 Specify iNumber for alarm type.
0x8002	(Reserve)	Sampling group of a specific group number Specify iNumber for group number.

Return value

Normal end: 0
Abnormal end: Error code

iNumber: (In) This argument is valid when sSaveFileName specifies a file of the GP3000 Series and WinGP.
For a file of the GP Series, this argument is ignored.
In addition, the meaning of this argument varies depending on the value of iBackupDataType.

<table>
<tr><th>Value of iBackupDataType</th><th>Description</th></tr>
<tr><td>0x0005 to 0x000C</td><td>Three types of alarm data (active, history and log) are available. Specify a target alarm type.
<table>
<tr><th>Value of iNumber</th><th>Description</th></tr>
<tr><td>0</td><td>'Pro-Server EX' checks if the alarm block contains valid data or not according to the following order of precedence, and reads valid data if any.
(1) Alarm history
(2) Alarm log
(3) Alarm active
If there is no valid data, an error occurs.</td></tr>
<tr><td>1</td><td>Reads alarm active data.</td></tr>
<tr><td>2</td><td>Reads alarm history data.</td></tr>
<tr><td>3</td><td>Reads alarm log data.</td></tr>
</table>
If the target data type does not exist in the alarm block specified withiBackupDataType, an error occurs.</td></tr>
<tr><td>0x8002</td><td>Group number of sampling group to be read
Any value from 1 to 64</td></tr>
<tr><td>Others</td><td>(Reserve)</td></tr>
</table>

iStringTable: (In) Reserve
Always specify "0".

Function	Writing SRAM backup data

Writes specified filing data in binary format into the SRAM of a GP Series node.

INT WINAPI EasyBackupDataWrite(LPCSTR sSourceFileName,LPCSTR sNodeName,INT iBackupDataType);

Argument

sSourceFileName: (In) File path of binary-formatted filing data to be written (String pointer)

sNodeName: (In) Name of entry node to write data (String pointer)
Pro-Server EX nodes, GP3000 Series nodes or WinGP nodes cannot be specified.

BackupDataType: (In) Fixed to "1". ("1" indicates filing data.)

Return value

Normal end: 0

Abnormal end: Error code

Special Note

26.7 CF Card APIs

Function	Reading CF card status

Acquires connection status of the CF card in a connected GP.

Single
INT WINAPI EasyIsCFCard(LPCSTR sNodeName) ;
Multi
INT WINAPI EasyIsCFCardM(HANDLE hProServer,LPCSTR sNodeName) ;

Argument
hProServer: Pro-Server handle
sNodeName: Name of GP node to read status
(This node name must be pre-registered in a network project.)

Return value

Function return value	For GP Series node	For GP3000 Series node/WinGP node
0x00000000	Normal	Normal
0x10000001	No CF card	No CF card, or CF card slot cover is opened (regardless of presence/absence of CF card)
0x10000002	Detection of device incompatible with CF card driver	
0x10000004	Detection of CF card error	Detection of CF card error
0x10000008	CF card not initialized	
Others	Error without relation to CF card	

Special Note

Function	Reading file list from CF card (Optional folder name)

Outputs a list of files from the CF card inserted in a GP node into a file specified with the parameter. You can specify an optional file to save the file list.

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

Argument
sNodeName: Name of GP node to output file list
sDirectory: Name of folder to receive file list (All capitals)
oiCount: Number of output files
sSaveFileName: Name of file to save output directory information. The specified file stores binary data of the alignment type specified with stEasyDirInfo, in the quantity specified with the return value of pioCount. All characters of the file name and extension are saved in capitals.

```
struct stEasyDirInfo {
BYTE bFileName[8+1];// File name (Terminated with "0")
BYTE bExt[3+1];// File extension (Terminated with "0")
BYTE bDummy[3];// Dummy
DWORD dwFileSize;// File size
BYTE bFileTimeStamp[8+1];// File timestamp (Terminated with "0")
BYTE bDummy2[3];// Dummy 2
} ;
```

Return value
Normal end: 0
Abnormal end: Error code

Special Note
As a supplement of "bFileTimeStamp" (8 bytes), high-order four bytes indicate time in the MS-DOS format, and low-order four bytes indicate date in the MS-DOS format (hexadecimal string).
The MS-DOS time/date format is as follows:
(Example: 20C42C22 is expressed as 2002/1/2 4:6:8. "2C22" is hexadecimal notation of date, and "20C4" is hexadecimal notation of time.)

Bit	Description
0 to 4	Day (1 to 31)
5 to 8	Month (1 = January, 2 = February , 12 = December)
9 to 15	Year: Expressed with the number of elapsed years from 1980. The actual year is the sum of 1980 and a value of these bits.

Specify time in the MS-DOS format. Time is packed in 16 bits in the following format:

Bit	Description
0 to 4	Number of seconds divided by two (0 to 29)
5 to 10	Minute (0 to 59)
11 to 15	Hour (0 to 23, on 24-hour basis)

Function	Reading file list from CF card (Type specification)

Outputs a list of files from the CF card inserted in a GP node into a file specified with the parameter. Only the file list in the directory specified with "sDirectory" can be output.

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

Argument
sNodeName: Name of GP node to output file list
sDirector: Name of directory to output list (All capitals) This API supports only the following directories:
LOG (Logging data)
TREND (Trend data)
ALARM (Alarm data)
CAPTURE (Capture data)
FILE (Filing data)
oiCount: Number of output files
sSaveFileName: Name of file to save output directory information. The specified file stores binary data of the alignment type specified with stEasyDirInfo, in the quantity specified with the return value of pioCount. All characters of the file name and extension are saved in capitals.

```
struct stEasyDirInfo {
BYTE bFileName[8+1];// File name (Terminated with "0")
BYTE bExt[3+1];// File extension (Terminated with "0")
BYTE bDummy[3];// Dummy
DWORD dwFileSize;// File size
BYTE bFileTimeStamp[8+1];// File timestamp (Terminated with "0")
BYTE bDummy2[3];// Dummy 2
} ;
```

Return value
Normal end: 0
Abnormal end: Error code

Special Note

Function	Reading file from CF card (Optional file name specification)

Reads a specified file from the CF card. You can specify an optional file to read.

INT WINAPI EasyFileReadInCfCard(LPCSTR sNodeName, LPCSTR sFolderName, LPCSTR sFileName, LPCSTR pWriteFileName, DWORD* odwFileSize);

Argument
sNodeName: Name of GP node to output file list
sFolderName: Name of folder containing source file to be read from CF card (Up to 32 single-byte characters)
sFileName: Name of source file to be read from CF card (Up to 8.3 format character string)
pWriteFileName
odwFileSize: Size of read CF file

Return value
Normal end: 0
Abnormal end: Error code

Special Note

Function	Reading file from CF card (Type specification)

Reads a specified file from the CF card. Only the file type specified with "pReadFileType" can be read.

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

Argument
sNodeName: Name of GP node to output file list
pReadFileType: Type of source file to be read from CF card (See <Special Note>)
wReadFileNo: File number of source file to be read from CF card
sWriteFileName

odwFileSize: Size of read CF file

Return value
Normal end: 0
Abnormal end: Error code

Special Note
This API supports the following file types. Only the files saved in a specified CF card folder can be read.

■File types supported for GP Series node

Data type	File type	Target folder
Filing data	ZF	FILE
CSV data	ZR	FILE
Image screen	ZI	DATA
Sound data	ZO	DATA
Line graph data	ZT	TREND
Sampling	ZS	TREND
Alarm 4 to 8	Z4 to Z8	ARAM
Logging data	ZL	LOG
Alarm log	ZG	ALARM
Alarm history	ZH	ALARM
Alarm active	ZA	ALARM
Screen data backup	ZC	MRM
Screen capture	CP	CAPTURE

■File types supported for GP3000 Series node and WinGP node

Data type	File type	Target folder
Filing data	ZF or F	FILE
CSV data	ZR	FILE
Image screen	ZI or I	DATA
Sound data	ZO or O	DATA
Line graph data dedicated to 'GP-Pro EX' (for compatibility)	ZT	TREND
Sampling data dedicated to 'GP-Pro EX' (for compatibility)	ZS	TREND
Alarm 1	Z1 or ZA	ALARM
Alarm 2	Z2 or ZH	ALARM
Alarm 3	Z3 or ZG	ALARM
Alarm 4 to 8	Z4 to Z8	ALARM
Logging data dedicated to 'GP-Pro EX' (for compatibility)	ZL	LOG
Capture data	CP	CAPTURE
Sampling 1 to 64	ZS1 to ZS64	SAMP01 to SAMP64

Function	Writing file into CF card (Optional file name specification)

Writes a specified file into the CF card. You can specify an optional file to write.

INT WINAPI EasyFileWriteInCfCard(LPCSTR sNodeName, LPCSTR pReadFileName, LPCSTR sFolderName, LPCSTR sFileName);

Argument
sNodeName: Name of GP node to write file
pReadFileName: Name of source file to be written into CF card (Full path)
sFolderName: Name of folder containing target file in CF card (Up to 32 single-byte characters)
sFileName: Name of target file in CF card (Up to 8.3 format character string)

Return value
Normal end: 0
Abnormal end: Error code

Special Note

Function	Writing file into CF card (Type specification)

Writes a specified file into the CF card. Only the file type specified with "pWriteFileType" can be written.

INT WINAPI EasyFileWriteCard(LPCSTR sNodeName, LPCSTR pReadFileName, LPCSTR sWriteFileType, WORD wWriteFileNo);

Argument
sNodeName: Name of GP node to write file
pReadFileName: Name of source file to be written into CF card (Full path)
sWriteFileType: Type of target file in CF card
(See <Special Note> of the function for "Reading file into CF card (Type specification)")
wWriteFileNo: File number of target file in CF card

Return value
Normal end: 0
Abnormal end: Error code

Special Note

Function	Deleting file from CF card (Optional file)

Deletes a specified file from the CF card. You can specify an optional file to delete.

INT WINAPI EasyFileDeleteInCfCard(LPCSTR sNodeName, LPCSTR sFolderName, LPCSTR sFileName) ;

Argument
sNodeName: Name of GP node containing file to be deleted
sFolderName: Name of folder containing file to be deleted from CF card (Up to 32 single-byte characters)
sFileName: Name of file to be deleted from CF card (Up to 8.3 format character string)

Return value
Normal end: 0
Abnormal end: Error code

Special Note

Function	Deleting file from CF card (Type specification)

Deletes a specified file from the CF card. Only the file type specified with "pDeleteFileType" can be deleted.

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

Argument
sNodeName: Name of GP node containing file to be deleted
pDeleteFileType: Type of file to be deleted from CF card (See <Special Note>)
wDeleteFileNo: File number to be deleted from CF card

Return value
Normal end: 0
Abnormal end: Error code

Special Note
If this function is executed for a file that does not exist in the CF card, it is not judged as an error, and the processing ends normally.
This API supports the following file types. Only the files saved in a specified CF card folder can be read.

■File types supported for GP Series node

Data type	File type	Target folder
Filing data	ZF	FILE
CSV data	ZR	FILE
Image screen	ZI	DATA
Sound data	ZO	DATA
Line graph data	ZT	TREND
Sampling	ZS	TREND
Alarm 4 to 8	Z4 to Z8	ARAM
Logging data	ZL	LOG
Alarm log	ZG	ALARM
Alarm history	ZH	ALARM
Alarm active	ZA	ALARM
Screen data backup	ZC	MRM
Screen capture	CP	CAPTURE

■File types supported for GP3000 Series node and WinGP node

Data type	File type	Target folder
Filing data	ZF or F	FILE
CSV data	ZR	FILE
Image screen	ZI or I	DATA
Sound data	ZO or O	DATA
Line graph data dedicated to 'GP-Pro EX' (for compatibility)	ZT	TREND
Sampling data dedicated to 'GP-Pro EX' (for compatibility)	ZS	TREND
Alarm 1	Z1 or ZA	ALARM
Alarm 2	Z2 or ZH	ALARM
Alarm 3	Z3 or ZG	ALARM
Alarm 4 to 8	Z4 to Z8	ALARM
Logging data dedicated to 'GP-Pro EX' (for compatibility)	ZL	LOG
Capture data	CP	CAPTURE
Sampling 1 to 64	ZS1 to ZS64	SAMP01 to SAMP64

Function	Renaming file in CF card

Renames a specified file in the CF card.

INT WINAPI EasyFileRenameInCfCard(LPCSTR sNodeName, LPCSTR sFolderName, LPCSTR sFileName, LPCSTR sFileRename) ;

Argument

sNodeName: Name of GP node to write file
sFolderName: Name of folder containing file to be renamed in CF card (Up to 32 single-byte characters)
sFileName: Name to file to be renamed in CF card (Up to 8.3 format character string)
sFileRename: New file name (Up to 8.3 format character string)

Return value

Normal end: 0
Abnormal end: Error code

Special Note

Function	Acquiring information on CF card empty space

Acquires information on empty space in the CF card connected to a specified entry node.

INT WINAPI EasyGetCfFreeSpace(LPCSTR sNodeName, INT* oiUnallocated);

Argument	**Return value**
sNodeName: Name of GP node to output file list oiUnallocated: Empty space in CF card (number of bytes)	Normal end: 0 Abnormal end: Error code

Special Note

Function	FTP passive mode setup

'Pro-Server EX' uses a special protocol to access the CF card in a GP Series node. However, to access a GP3000 Series node and WinGP node, FTP protocol is used.
For FTP protocol, 'Pro-Server EX' supports two modes: normal mode and passive mode.
This API specifies the mode of FTP protocol.

INT WINAPI EasyFileSetPassiveMode(INT iPassive);

Argument	**Return value**
iPassive: (In) 0: Normal mode Other than 0: Passive mode At initialization of ProEasy, the FTP protocol is set to "Normal mode".	Normal end: 0 Abnormal end: Error code

Special Note

26.8 Other APIs

Function	Reading time from GP (DWORD-type)

Acquires current time of a specified node as a DWORD-type value. This function is valid only for the time saved in 6 words from LS2048.

DWORD WINAPI EasyGetGPTime(LPCSTR sNodeName, DWORD* odwTime);

Argument
sNodeName: Name of target node (A Pro-Server EX node cannot be specified.)
odwTime: Acquired time (Time is acquired as a value of DWORD type, (substantially, time_t type defined by ANSI).)

Return value
Normal end: 0
Abnormal end: Error code

Special Note

Function	Reading time from GP (VARIANT-type)

Acquires current time of a specified node as a Variant-type value. This function is valid only for the time saved in 6 words from LS2048.

DWORD WINAPI EasyGetGPTimeVariant(LPCSTR sNodeName, LPVARIANT ovTime);

Argument
sNodeName: Name of target node (A Pro-Server EX node cannot be specified.)
ovTime: Acquired time (Time is acquired as a value of VARIANT type. Internal possessing format is "Date".)

Return value
Normal end: 0
Abnormal end: Error code

Special Note

Function	Reading time from GP (STRING-type)

Acquires current time of a specified node as an LPTSTR-type character string. This function is valid only for the time saved in 6 words from LS2048.

DWORD WINAPI EasyGetGPTimeString(LPCSTR sNodeName, LPCSTR sFormat, LPSTR osTime);

Argument
sNodeName: Name of target node (A Pro-Server EX node cannot be specified.)
pFormat: String to specify the format of time to be acquired as a string. The format specification codes subsequent to the percentage (%) symbol are changed as shown in <Special Note>.
Other characters are expressed without a change.
osTime: Time acquired as a string (If a memory area larger than the acquired string length + 1 (NULL) is not secured, unexpected memory destruction occurs. To prevent this, you must secure a memory area larger than the expected string length + 1 (NULL). Otherwise, the operation cannot be guaranteed.)

Return value
Normal end: 0
Abnormal end: Error code

Special Note
The format specification codes subsequent to the percentage (%) symbol are changed to those listed in the table below. Other characters are expressed without a change. For example, if "%Y_%M %S" is specified, an actual time of "2006/1/2 12:34:56" is expressed as a string of "2006_34 56".

Format specification code	Folder
%a	Abbreviated name of day of week (*2)
%A	Formal name of day of week (*2)
%b	Abbreviated name of month (*2)
%B	Formal name of month (*2)
%c	Expression of date and time depending on locale
%#c	Longer expression of date and time depending on locale
%d	Decimal expression of day of month (01 to 31) (*1)
%H	Time expression on 24-hour basis (00 to 23) (*1)
%I	Time expression on 12-hour basis (01 to 12) (*1)
%j	Decimal expression of day of year (001 to 366) (*1)
%m	Decimal expression of month (01 to 12) (*1)
%M	Decimal expression of minute (00 to 59) (*1)
%p	AM/PM division for current locale (*2)
%S	Decimal expression of second (00 to 59) (*1)
%U	Decimal expression of serial week number. Sunday is regarded as the first day of the week. (00 to 53) (*1)
%w	Decimal expression of day of week. Sunday is regarded as "0 ". (0 to 6) (*1)
%W	Decimal expression of serial week number. Monday is regarded as the first day of the week. (00 to 53) (*1)
%x	Expression of date for current locale
%#x	Longer expression of date for current locale
%X	Expression of time for current local (*2)
%y	Decimal expression of low-order 2 digits of the dominical year (00 to 99) (*1)
%Y	Decimal expression of 4 digits of the dominical year (*1)
%z, %Z	Name or abbreviated name of time zone. If time zone is unknown, leave it blank. (*2)
%%	Percentage symbol (*2)

* 1: If "#" is added before d, H, I, j, m, M, S, U, w, W, y or Y (ex. %#d), leading "0" will be deleted. (ex. "05" is expressed as "5".)

* 2: If "#" is added before a, A, b, B, p, X, z, Z or % (ex. %#a), "#" will be ignored.

Function	Reading time from GP (STRING VARIANT-type)

Acquires current time of a specified node as a Variant-type character string. This function is valid only for the time saved in 6 words from LS2048.

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

Argument

sNodeName: Name of target node (A Pro-Server EX node cannot be specified.)

pFormat: String to specify the format of time to be acquired as a string. The format specification codes subsequent to the percentage (%) symbol are changed to those listed below. Other characters are expressed without a change. (For details, refer to <Special Note> of "Reading time from GP (STRING-type)".)

ovTime: Time acquired as a string (Time is acquired as VARIANT type. Internal possessing format is "BSTR".)

Return value

Normal end: 0

Abnormal end: Error code

Special Note

Function	Reading entry node status

Acquires connected GP node status. Since the response time-out value can be changed, this function can be used to check connection status.

Single

INT WINAPI GetNodeProperty(LPCSTR sNodeName,DWORD dwTimeLimit,LPSTR osGPType,LPSTR osSystemVersion,LPSTR osComVersion,LPSTR osECOMVersion);

Multi

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

Argument

hProServer: (In) Pro-Server handle

sNodeName: (In) Name of GP node to read status

dwTimeLimit: (In) Response time-out setting value
(If "0" is specified, it is set to the default value of 3000 ms.)
The setting range is from 1 to 2,147,483,647. (Unit: ms)

The API returns status information on the target node to the following area.
Secure an area of at least 32 bytes for each item.

osGPType: (Out) GP model code

osSystemVersion: (Out) GP system version

osComVersion: (Out) PLC protocol driver version
For Pro-Server EX nodes or GP3000 Series nodes/WinGP nodes, this item is blank.

osECOMVersion: (Out) 2way driver version
For Pro-Server EX nodes or GP3000 Series nodes/WinGP nodes, this item is blank.

Return value

Normal end: 0

Abnormal end: Error code

Special Note

Function	Acquiring symbol/group byte size

Acquires the total number of bytes required to access a device symbol or group symbol.

INT WINAPI SizeOfSymbol(LPCSTR sNodeName,LPCSTR sSymbolName,INT* oiByteSize);

Argument	**Return value**
sNodeName: (In) Name of entry node with Device/PLC name sSymbolName: (In) Name of target device or symbol name oiByteSize: (Out) Byte size acquired	Normal end: 0 Abnormal end: Error code

Special Note
For "sSymbolName", a device symbol, non-alignment group, whole alignment group, or an element of alignment group can be specified.

Function	Acquiring number of group members

Acquires the number of members of a group or symbol sheet (total number of symbols and group members).

INT WINAPI GetCountOfSymbolMember(LPCSTR sNodeName,LPCSTR sSymbolName,INT* oiCountOfMember);

Argument	**Return value**
sNodeName: (In) Name of entry node with Device/PLC name sSymbolName: (In) Name of target group symbol or symbol sheet oiCountOfMember: (Out) Number of members acquired	Normal end: 0 Abnormal end: Error code

Special Note
When a group symbol exists in a specified group symbol, the number of members is counted as one, even if multiple device symbols exist in the inner group symbol.

Function	Acquiring symbol/group/symbol sheet definition information

Acquires definition information (data type, data quantity, etc.)

INT WINAPI GetSymbolInformation(LPCSTR sNodeName,LPCSTR sSymbolName,INT iMaxCountOfSymbolMember,LPSTR osSymbolSheetName,SymbolInformation* oSymbolInformation,INT* oiGotCountOfSymbolMember);

Argument	**Return value**
sNodeName: (In) Name of entry node with Device/PLC name sSymbolName: (In) Name of symbol/group/symbol sheet iMaxCountOfSymbolMember: (In) Specify a value of the maximum count of desired information + 1. Specify the number of "oSymbolInformation" prepared. osSymbolSheetName: (Out) The API returns the name of symbol sheet that contains the symbol specified with sSymbolName. Prepare 66 bytes or larger work. oSymbolInformation: (Out) The API returns acquired detail information in the alignment structure. Prepare work for the number specified with iMaxCountOfSymbolMember. oiGotCountOfSymbolMember: (Out) The API returns the information quantity that has returned to oSymbolInformation.	Normal end: 0 Abnormal end: Error code

Special Note

- Structure of SymbolInformation

```
struct SymbolInformation
{
   WORDm_wAppKind;// Data type, Symbol: 1 to 12, Group: 0x8000
   WORDm_wDataCount; // Data quantity
   DWORDm_dwSizeOf; // Number of bytes in buffer required for access
   char m_sSymbolName[64+1];// Name of symbol or group
   charm_bDummy1[3];// Reserve
   charm_sDeviceAddress[256+1]; // Device address (For group, leave it blank.)
   charm_bDummy2[3];// Reserve
};
```

Acquired information is returned to oSymbolInformation in the alignment structure specified with SymbolInformation.
Information on the symbol, group or sheet specified with sSymbolName is set in the first element.
Group member information is set in the second and subsequent elements, when sSymbolName indicates a group.
When sSymbolName indicates a sheet, information on the whole sheet is set in these elements.
When sSymbolName indicates a symbol, there is no information in the second or subsequent elements.

If the target symbol is a bit offset symbol, pay attention to the following points:

(1) When a bit offset symbol is directly specified as an information source symbol (a bit offset symbol is directly specified for sSymbolName), "2" is set to m_dwSizeOf of SymbolInformation, or the first element of oSymbolInformation, as the number of bytes required to access the bit symbol.
In this case, since the information source is one symbol, oSymbolInformation does not have second or subsequent element.

(2) When a group symbol is specified as an information source symbol and the specified group contains a bit offset symbol, "0" is set to m_dwSizeOf , or the second or subsequent element of oSymbolInformation, because it indicates the access size required for a group access member.

- If the number of members is unknown, call GetCountOfSymbolMember() to acquire it. To call this function, prepare SymbolInformation as the number of work of the specified count + 1.

26.9 Precautions for Using APIs

■ About data types available with 'Pro-Server EX'

(1) Principal data types that can be specified with APIs, or received in response to APIs

Definition name	Decimal value	Hexadecimal value	Meaning of data
EASY_AppKind_Bit	1	0x0001	Bit Data
EASY_AppKind_SignedWord	2	0x0002	16-bit(Signed) Data
EASY_AppKind_UnsignedWord	3	0x0003	16-bit(Unsigned) Data
EASY_AppKind_HexWord	4	0x0004	16-bit(HEX) Data
EASY_AppKind_BCDWord	5	0x0005	16-bit(BCD) Data
EASY_AppKind_SignedDWord	6	0x0006	32-bit(Signed) Data
EASY_AppKind_UnsignedDWord	7	0x0007	32-bit(Unsigned) Data
EASY_AppKind_HexDWord	8	0x0008	32-bit(HEX) Data
EASY_AppKind_BCDDWord	9	0x0009	32-bit(BCD) Data
EASY_AppKind_Float	10	0xA	Single-precision floating point data
EASY_AppKind_Real	11	0xB	Double-precision floating point data
EASY_AppKind_Str	12	0xC	Character string data

(2) Data types available in special cases

Definition name	Decimal value	Hexadecimal value	Meaning of data
EASY_AppKind_NULL	0	0x0000	Indicates that the data type defined for a symbol is used with the API that can use the symbol as the device address.
EASY_AppKind_BOOL	513	0x0201	Handles bit data as Variant BOOL data per bit.
EASY_AppKind_Group	-32768	0x8000	Group symbol
EASY_AppKind_SymbolSheet	-28672	0x9000	Symbol sheet

■ About entry node name with Device/PLC name

(1) GP3000 Series nodes and WinGP nodes can be connected to multiple devices/PLCs. To access these Device/PLCs, you must specify the names of the entry node and Device/PLCs.

(2) For some arguments of the Pro-Server EX APIs, you may specify an entry node name only. For other arguments, you must specify a Device/PLC name as well as the entry node name.

<How to specify a Device/PLC name>

To specify a D"evice/PLC name, add "." (dot) after the entry node name.

Example)

AGPNode.PLC1

(3) To access a device incorporated in a GP3000 Series node/WinGP node or Pro-Server EX node, specify "#INTERNAL" as the Device/PLC name. (It can be omitted.)

(4) To access the memory of a memory link driver in a GP3000 Series node/WinGP node, specify "#MEMLINK" as the Device/PLC name. (It cannot be omitted.)

(5) To access a GP Series node or Pro-Server EX node, you need not specify a Device/PLC name.
("." (dot) is not necessary.)

(6) To access a Device/PLC assigned to an internal device or "system area device" in a GP3000 Series node/ WinGP node, you can omit specification of the Device/PLC name by specifying an entry node name with Device/ PLC name.
In this case, however, 'Pro-Server EX' searches the target device for an internal device first, and then searches for a Device/PLC assigned to the "system area device".

■ About symbol searching precedence

For the Device Access APIs of 'Pro-Server EX', you must specify the entry node name with Device/PLC name, and the device address or device symbol as a character string. 'Pro-Server EX' judges according to the following order of precedence whether the specified character string directly specifies the device address or a device symbol.

(1) 'Pro-Server EX' searches the symbol sheet for a matching name. If the specified string exists in the symbol sheet, it is regarded as a sheet.

(2) 'Pro-Server EX' regards the specified string as a group name or symbol, and searches a local symbol sheet. If the specified string exists in the local symbol sheet, it is regarded as a local symbol.

(3) If the specified string does not exist in the local symbol sheet, 'Pro-Server EX' searches a global symbol sheet. (In this case, the target global symbol sheet is that for the Device/PLC that has been specified with "entry node name with Device/PLC name". Global symbol sheets for different Device/PLCs are not searched.)

(4) If the specified string does not exist in the global symbol sheet, it is regarded as a device address.

■ Duplication of name

'Pro-Server EX' provides the following name categories:
(1) Node Name
(2) Device/PLC Name
(3) Trigger Condition Name
(4) Symbol Sheet Name
(5) Group/Symbol Name
(6) ACTION Name

In principle, 'Pro-Server EX' must not have a duplicated name, excepting the following cases:

(1) Duplication of a Device/PLC name causes no problem, if they belong to different entry nodes.
(2) Duplication of a group/symbol name causes no problem, if they belong to different entry nodes or different Device/PLCs.

■ Duplication of global symbol name and local symbol name

When a Pro-Server EX API uses a symbol to specify a device address and the same symbol name exists for both local symbol and global symbol, it is regarded as a local symbol.

■ Using Pro-Server EX API for multi-thread application

All functions of Pro-Server EX APIs are synchronous type. (Once a function is called, it will not be returned until processing is completed.)
Therefore, when 'Pro-Server EX' accesses multiple entry nodes by using a single-thread application, processing is executed for individual nodes in sequence.
On the other hand, with a multi-thread application, 'Pro-Server EX' can access another entry node through another thread, even when one thread is used for access to one entry node.
Pro-Server EX APIs can be used for the multi-thread application.
To create a multi-thread application, pay attention to the following points:

(1) In principle, to execute a multi-thread application, use Multi-Handle functions.

(2) To use Multi-Handle functions, you must create Pro-Server EX handles. Use separate Pro-Server EX handles for individual threads.
Even if multiple Pro-Server EX handles are created for one thread, there is no problem. However, you must not use a Pro-Server EX handle that has been created for another thread.
To release a Pro-Server EX handle, use the same thread where the handle has been created.

(3) To use a Pro-Server EX API, you must call EasyInit() first.
However, most Pro-Server EX APIs automatically call EasyInit() when each API is called before EasyInit(). Therefore, when using a single-thread application, you need not consider EasyInit() in your program.

(4) The thread where EasyInit() is called must exist until the end of application. If the thread where EasyInit() is called is closed in the middle of application, the operation cannot be guaranteed.

(5) For general applications, the thread used to start an application will exist until the end of application. (Normally, this applies to applications created by VB or VC.) Therefore, to create a multi-thread application, we recommend you to call EasyInit() at the start of application.

■ Improving cache buffer update efficiency

(1) To use the cache function, you must register a device in the cache buffer. (Register a device on the Pro-Studio EX cache registration screen, or by using the cache buffer control APIs.)
Performance of the whole system varies depending on the registration method.

(2) To select a device to be registered, use the device access log function to identify the device that 'Pro-Server EX' accesses.

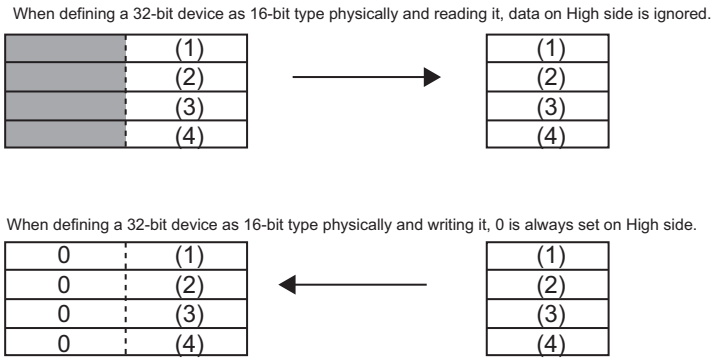
(3) In principle, you should cache-register a device that has been frequently read.

(4) When multiple devices are registered, the processing speed becomes higher if these devices can be registered in series.
(Ex.1) When LS100 and LS101 are registered in a cache buffer, the processing speed becomes higher if two devices are registered in series from LS100, rather than separately registered. Also, if the interval between two devices is only several words, the processing speed may be increased if these devices are registered in series.
(Ex.2) When LS100 and LS103 are registered in a cache buffer, the processing speed becomes higher if four devices are registered in series from LS100, rather than separately registered.

(5) When bit devices are registered in series, the processing speed becomes higher if they can be registered as word devices.
(Ex.) When devices for 20 bits are registered in series from LS123401, the processing speed becomes higher if they are registered in two words from LS1234.

■ 16-bit access operation for device with physically 32-bit width

(1) When a 16-bit symbol is assigned to a device with a physically 32-bit width, and the device is accessed with the 16-bit symbol, or when 16-bit data type is directly specified to access a 32-bit device, 'Pro-Server EX' can handle the 32-bit device as a 16-bit device.
In this case, 'Pro-Server EX' executes the following conversion for READ and WRITE APIs.



(2) The above conversion is executed during access using a data transfer function or API.

(3) When data is transferred between GP Series nodes, an error occures.

(4) With older versions of 'Pro-Server', if 16-bit access is executed for a device with physically 32-bit width, an error occures.

■ 16-bit access operation for device with physically 32-bit width

When a 32-bit symbol is assigned to a device with a physically 16-bit width, and the device is accessed with the 32-bit symbol, or when 32-bit data type is directly specified to access a 16-bit device, 'Pro-Server EX' can handle the 16-bit device as a 32-bit device.
In this case, 'Pro-Server EX' handles a series of two devices with a 16-bit width as one device.

■ About Pro-Server auto start, forced closing and restart

(1) If 'Pro-Server EX' has not been started yet, calling a Pro-Server EX API automatically starts 'Pro-Server EX' (excepting some APIs).
If 'Pro-Server EX' cannot start, the API always returns an error code.

(2) After 'Pro-Server EX' normally starts, calling the second or subsequent API will not start 'Pro-Server EX' again, because 'Pro-Server EX' has already been started.

(3) If 'Pro-Server EX' is closed in the middle of application processing, and then an API is called ('Pro-Server EX' has been closed when the second or subsequent API is called), the API will not start 'Pro-Server EX'. It returns an error code.

(4) Do not close 'Pro-Server EX' in the middle of application processing.
Before closing 'Pro-Server EX', be sure to close the application first. (Do not call an API after closing 'Pro-Server EX'.)

However, if 'Pro-Server EX' is manually restarted from the Windows START menu, the API executes Pro-Server EX recovery processing, and tries to continue processing. If 'Pro-Server EX' can be recovered, it continues processing. However, 'Pro-Server EX' may fail in recovery processing, depending on the previous closing method. For example, recovery processing failures may occur in the following cases:

- When 'Pro-Server EX' is forcibly closed from Task Manager
- When 'Pro-Server EX' is closed during a call of an API

■ About specification of symbol index

Specification of symbol index is enabled only by a device name for an API. Specification of symbol index is to specify a value in [ ] after a symbol name, as shown below. The symbol index indicates the device located ahead from the device specified with the symbol name, by the number of devices specified by the "value" of the symbol data type.

(Symbol name)[Value]

Example) Valve [2]

When valve symbol "D100" is specified as "16-bit signed", Valve [2] indicates D102. When "D100" is specified as "32-bit unsigned", it indicates D104.

■ About queuing cache read and symbol cache read

When queuing cache read (queuing registration using a ReadDevice function (without "D") after BeginQueuingRead) or symbol cache read (ReadSymbol (without "D")) is used, the operation varies depending on which part of target devices has been cache-registered.

- When all target devices have been cache-registered: cache read is executed.
- When all target devices have not been cache-registered: direct read is executed.
- When only some of target devices have been cache-registered: Some of target devices are subjected to cache read, and remaining devices are subjected to direct read. However, cache read is not applied to all of the cache-registered devices. direct read may be applied to some of the cache-registered devices. If you have a trouble in identifying the devices subjected to cache read, you should cache-register all target devices, or use a Direct Read API instead of a Cache Read API.

■ About APIs that cannot be used for .NET

The following APIs cannot be used for .NET. If these APIs are used, operations cannot be guaranteed.

- Symbol access (Byte access)

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

- Symbol size acquisition function

SizeOfSymbol()

■ When using simple DLL in a multi-thread application

All functions of Pro-Easy APIs are synchronous type. (Once a function is called, it will not be returned until processing is completed.) Therefore, when accessing multiple entry nodes by using a single-thread application, processing is executed for individual nodes in sequence. On the other hand, with a multi-thread application, you can access another entry node through another thread, even when one thread is used for access to one entry node. Pro-Easy APIs can be used for the multi-thread application.

To create a multi-thread application, pay attention to the following points:

1. In principle, to execute a multi-thread application, use Multi-Handle functions.
2. To use Multi-Handle functions, you must create 'Pro-Server EX' handles. Use separate 'Pro-Server EX' handles for individual threads. Even if multiple 'Pro-Server EX' handles are created for one thread, there is no problem. However, you must not use a 'Pro-Server EX' handle that has been created for another thread. To release a 'Pro-Server EX' handle, use the same thread where the handle has been created.
3. To use 'Pro-Server EX API', you must call EasyInit() first. As most Pro-Server EX APIs automatically call EasyInit() when each API is called before EasyInit(), you need not to consider EasyInit() call in your program.
4. In the multi-thread program, the program must call EasyInit() first from the thread (main thread) which was started first.When you call a Pro-Server EX API except from the main thread, call EasyInit() from the main thread in advance.

■ Message Process in Windows

Most of the Windows programs are event-driven, i.e. displaying the dialog box or playing the sounds according to various events including "an icon is clicked", "a mouse is moved", or "a key is pressed".

When an event occurs, Windows will send the message showing the event type to the application.The application confirms that the event occurs by receiving the message and executes each process.

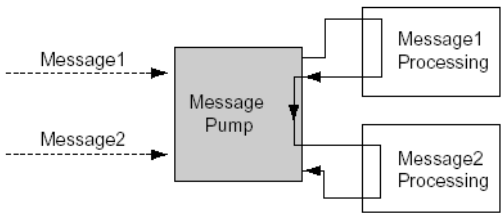
In this manual, the part which receives messages in order and branches into each process (corresponding to DoEvents for VB, or the part executing GetMessage() and DispatchMessage() for VC) is called the message pump. The message pump is not much recognized because it is hidden in the VC or VB framework when programming with VC or VB normally. However, unless this message pump operates properly, Windows applications will cause unintended operation.

For example, when it takes long time for a routine to process a message and recover, the application fails to process the event because it cannot receive an event which occurs in the meantime from Windows.

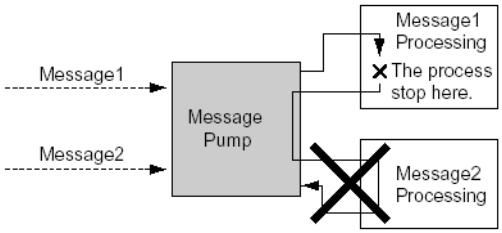
Example) Assume that messages are sent from Windows in the order of message 1 to message 2.

The message pump takes out the message 1 and calls the subroutine for message 1.

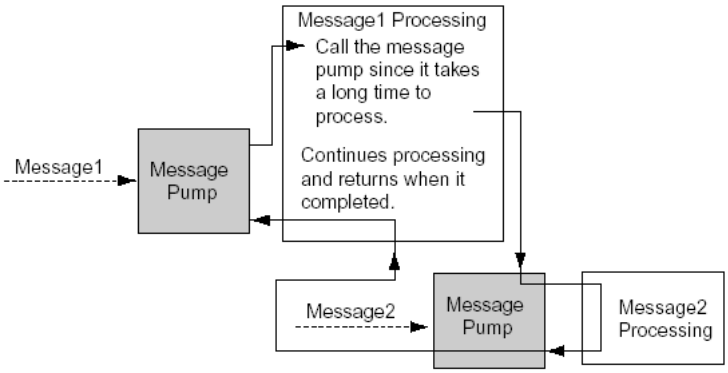
Then, when the message pump recovers from above, it takes out the following message (message 2) and calls the subroutine for message 2.



In this case, assume that it takes long time for processing message 1. Then the message pump cannot process message 2 without recovering.



In such case, force the message pump to run. (calling DoEvents,VC for VB, or GetMessage() and DispatchMessage() for VC)



Windows applications are created assuming an application should run the message pump properly. "Pro-Server EX API" runs the message pump using function for time-consuming process so as to avoid the case shown in (Example).

■ Prohibition of API Double Call

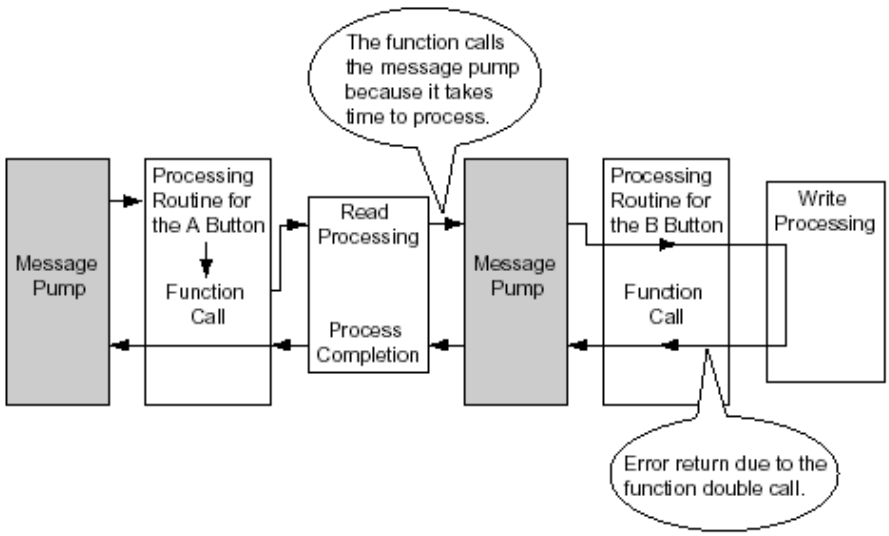
'Pro-Server EX API' prohibits another communication while communicating with a party (while calling a 'Pro-Server EX' function)(double-call). (Double-call is enabled if using the Multi-Handle. For details, refer to the section of Multi-Handle.) However, as 'Pro-Server EX API' runs the message pump inside API, a user program will start to run when an event occurs.

When API is called in the message process routine, double-call may occur.

Examples of double-call are shown below.

1. Double-call by pressing 2 buttons

   Assume that there are 2 buttons, A and B. Device read API is called when A is pressed; device write API is called when B is pressed.In this case, press the button B to cause the device write API to be called while calling the device read API when pressing the button A, which leads API double-call and error occurs.



2. Double-call by timer

When periodical process is executed in the Windows program, timer events are often used. However, API double-call may happen in the program using timer events due to careless programming.

(1) Call the device read API periodically per second, read the device and display it.

(2) Such programs as call the device write API when a button is pressed and write the value in the device causes an error in the following cases.

When pressing the button (2) while reading a timer event (1), and the process (2) starts to run

When a timer event occurs while writing (2) and read (1)

■ Solutions to avoid API Double-Call

Solutions to avoid API double-call are shown below.

(1) Improve the algorithm not to execute API double-call in a user program.For example,

1. Timer should be always cancelled at the head of timer process routine and button process routine.
2. While a process is running by pressing a button, the button or another button should be ignored even if pressed.

(2) API double-call does not occur if the 'Pro-Server EX' handle using multi-handle is different.

Use API in Multi-Handle type to set the handle of the program in the area which is possible to cause double-call to different handle.

(3) Message should not be processed inside API

Call EasySetWaitType() by argument 2. However, in this case, other problems such as an application causes unintended operation may occur, because other messages except the one which causes double-call will not be processed.

■ How to read character strings in VB

(1) Use ReadDeviceStr to read character strings in VB

In this case, you need to specify (fix) the size of storing destination of character strings read in advance.'

```
'
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) Use Variant type if you use ReadDeviceVariant to read character strings in VB, but not specify the size of storing destination of character strings read in advance.

```
'
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
```

Note that GP uses NULL for the completion of character strings. For that reason, you need to shorten the character string if the character string obtained in the above method includes NULL as the completion of character strings.
Sample functions to shorten character strings to NULL are shown below.

```
    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
```

26.10 Using APIs (Examples)

By using the read/write functions provided by 'Pro-Server EX', you can read/write data from/into a VB or VC application.

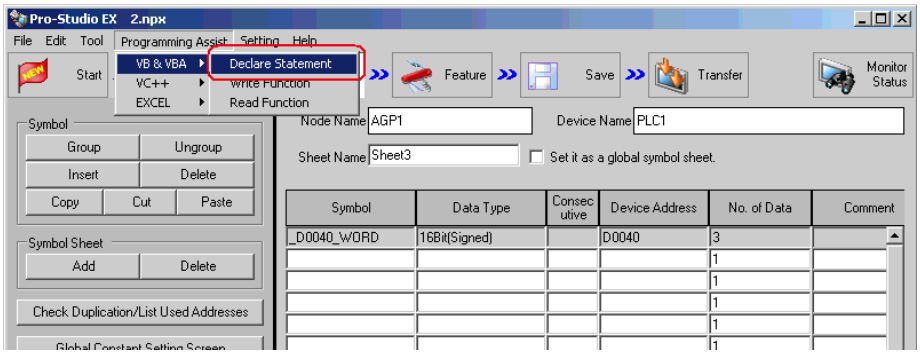
This section describes the procedure for reading/writing a specified symbol with the APIs.

☞ "26.10.1 VB Support Function"
☞ "26.10.2 VC Support Function"
☞ "26.10.3 VB .NET Support Function"
☞ "26.10.4 C# .NET Support Function"

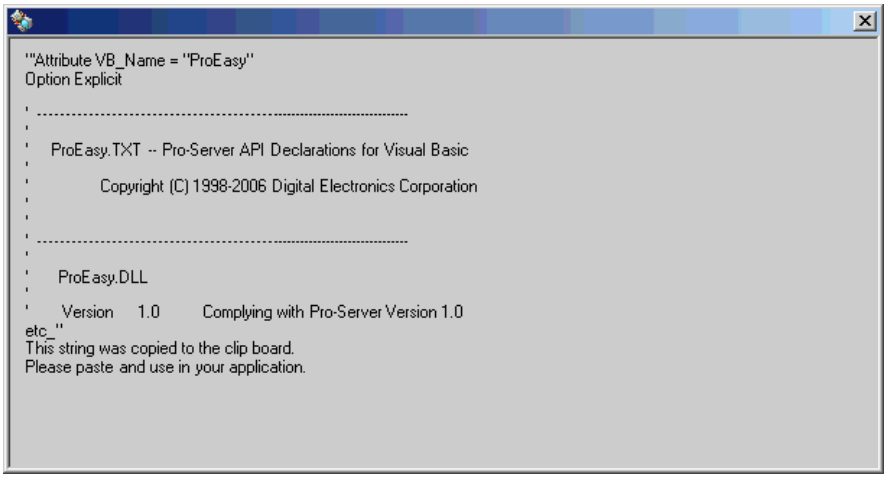
26.10.1 VB Support Function

VB: Declaration statement

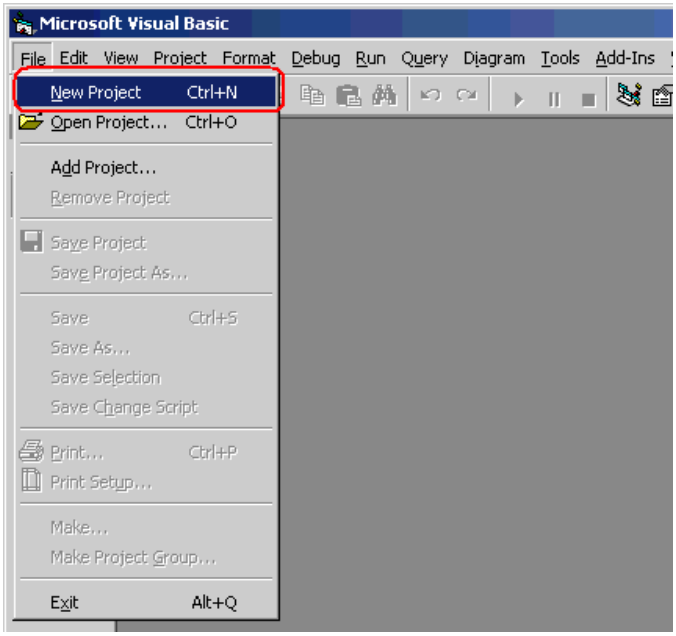
1 Select [Programming Assist] - [VB & VBA] - [Declare Statement].



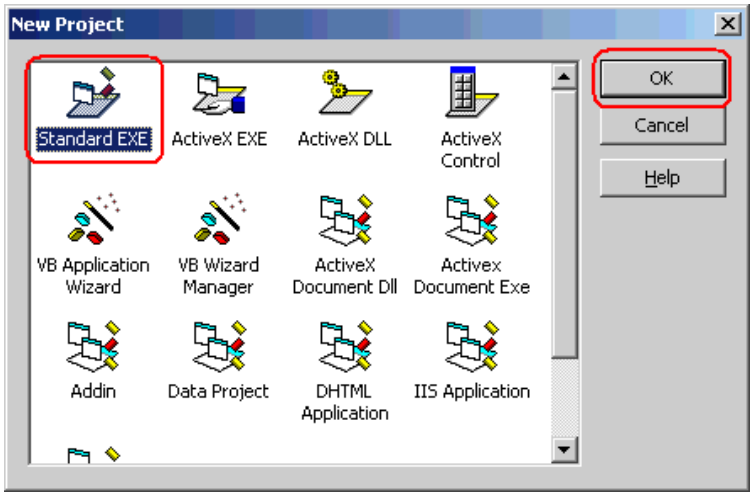
The VB declaration statement is coped to the clipboard.



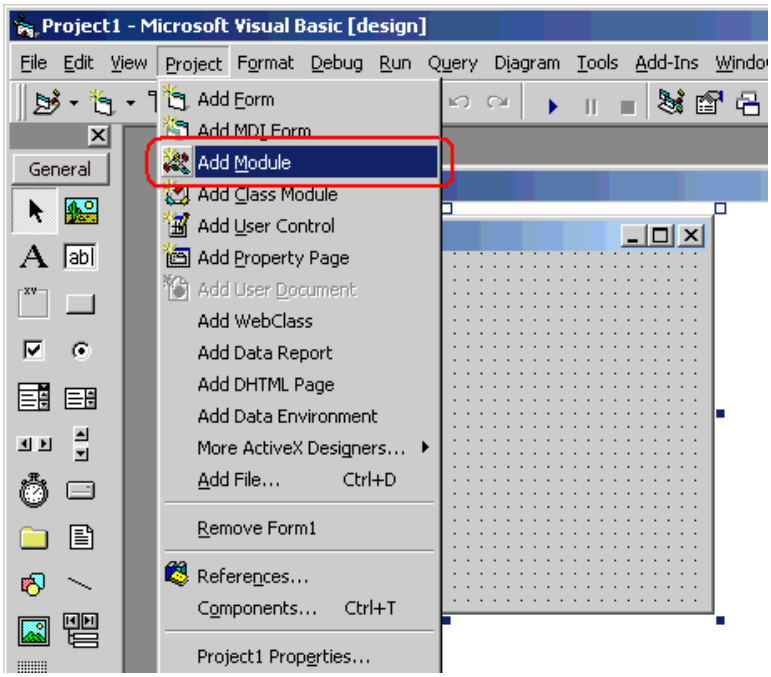
2 Start Microsoft Visual Basic, and select [New Project] from [File] on the menu.



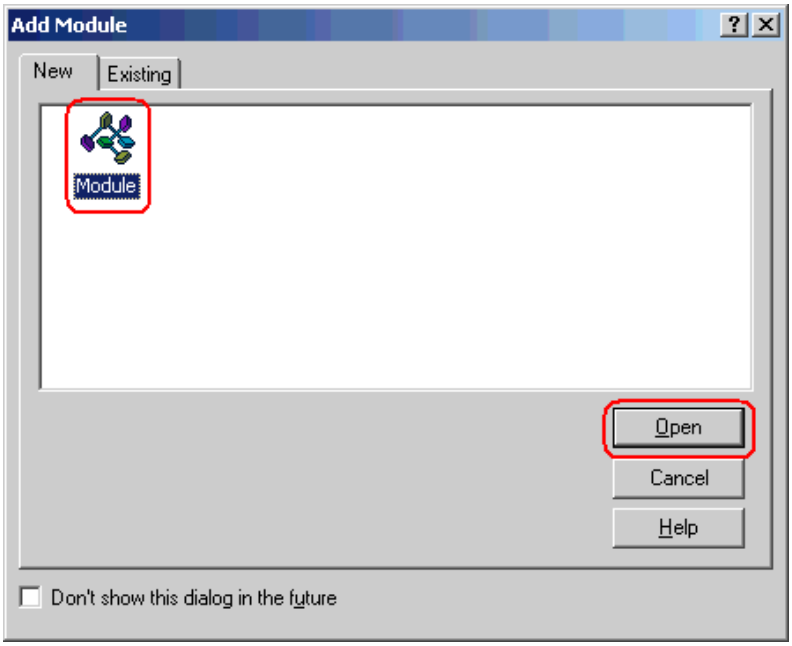
3 Select [Standard EXE], and click the [OK] button.



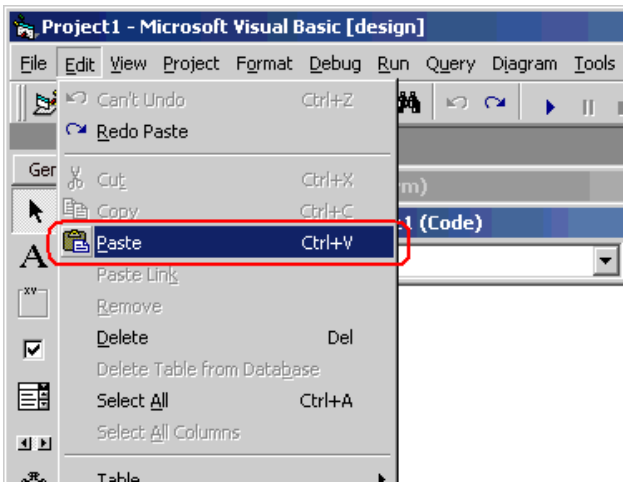
4 Select [Add Module] from [Project] on the Microsoft Visual Basic menu.



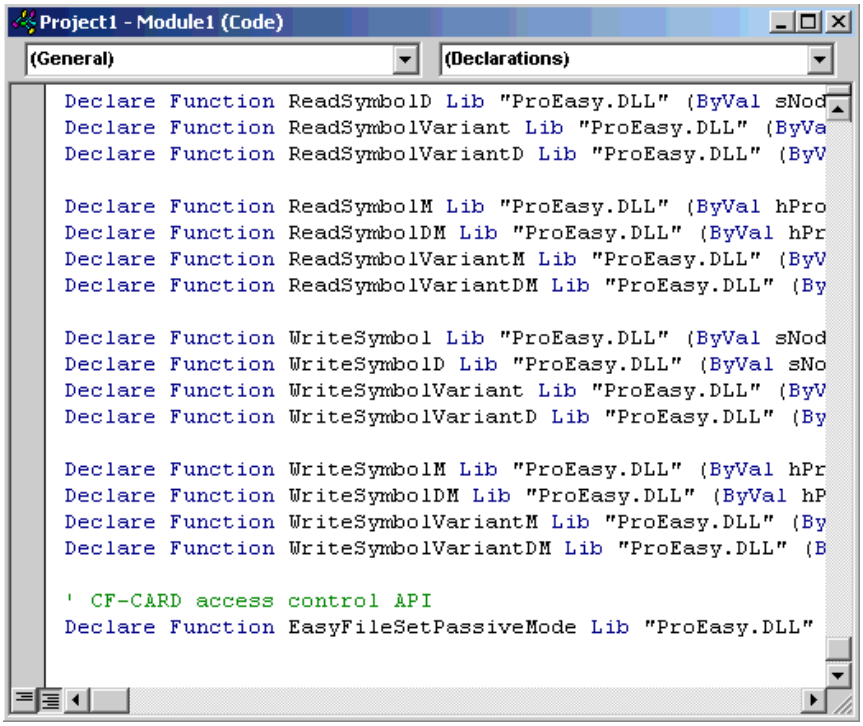
5 Select [Module] in the [New] tab, and click the [Open] button.



6 Select [Paste] from [Edit] on the Microsoft Visual Basic menu, and paste the declaration statement (data on the clipboard) to the added standard module.



The deceleration statement is now pasted.



This is the end of the function (read/write function) declaration procedure.

The above 1 to 6 steps apply to both reading and writing applications.

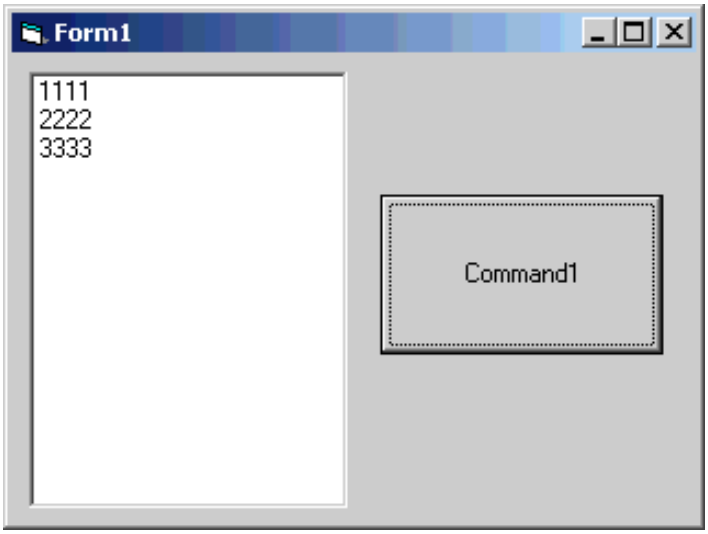
The following procedure varies depending on whether the application is intended for reading or writing, and so is explained individually.

To create a "Reading" application, refer to steps 7 to 16.

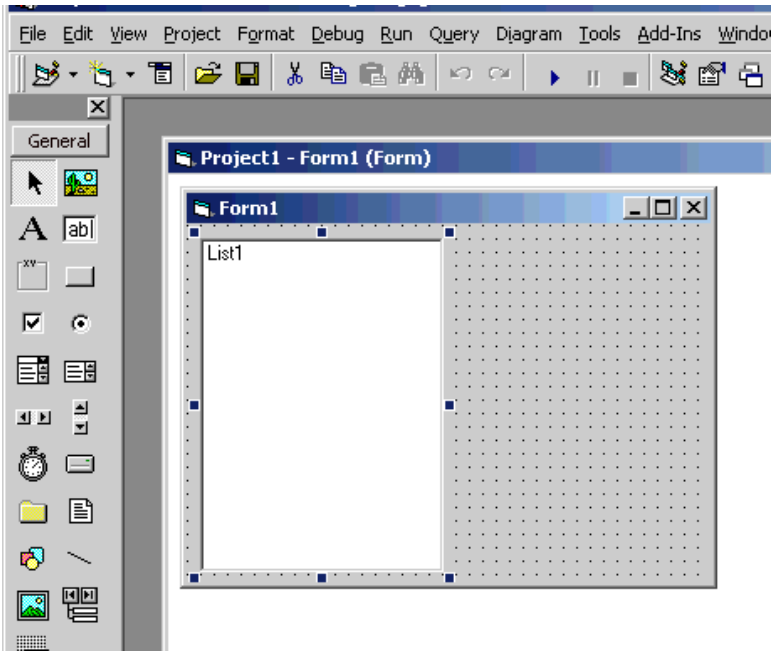
To create a "Writing" application, refer to steps 17 to 26.

Creating "Reading" application

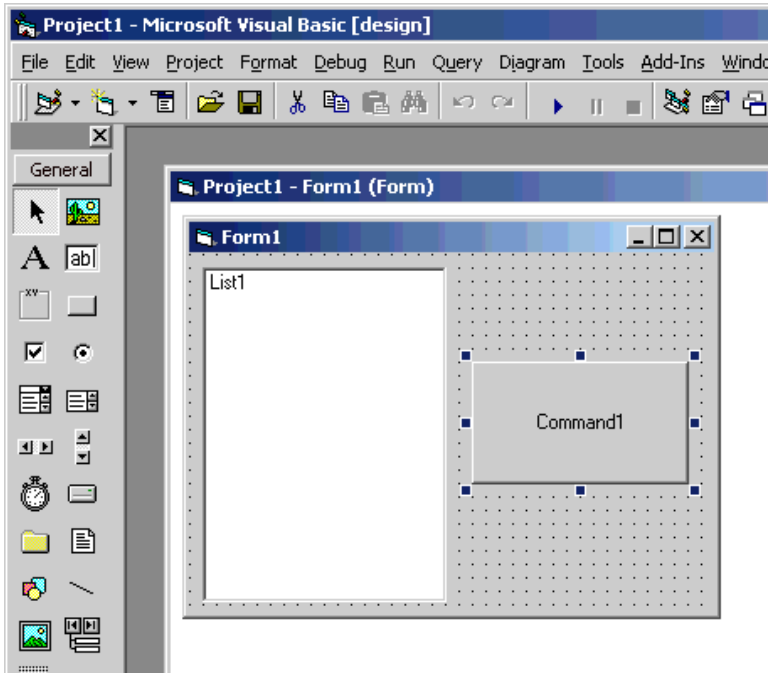
This section describes the procedure for creating an application that reads and displays data (16-bit signed data) for three points with a click on [Command1].



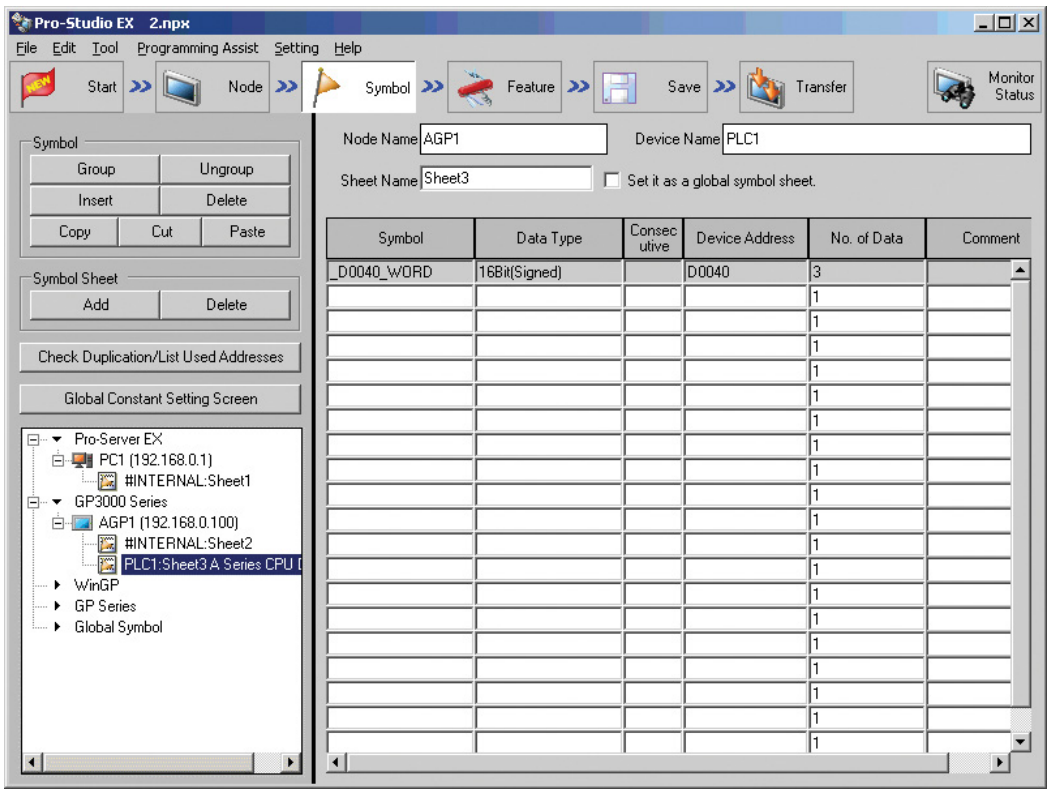
7 Select [ListBox] and paste it to [Form1].



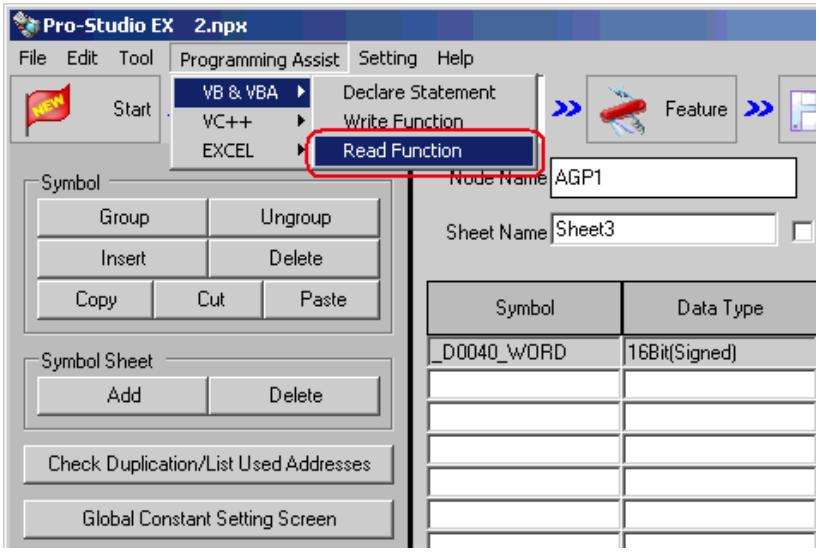
8 Select [CommandButton] and paste it [Form1].



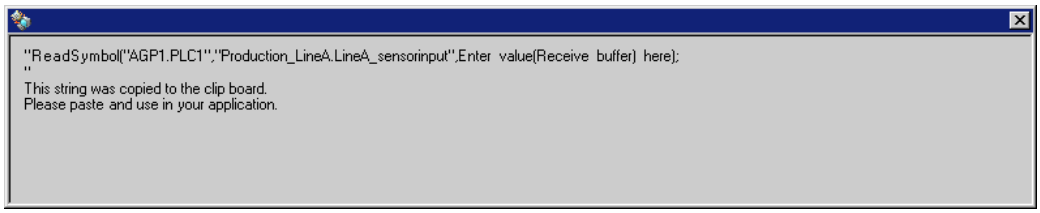
9 Select a target symbol name from those registered in 'Pro-Server EX'. (Select the symbol with first-address for reading.)



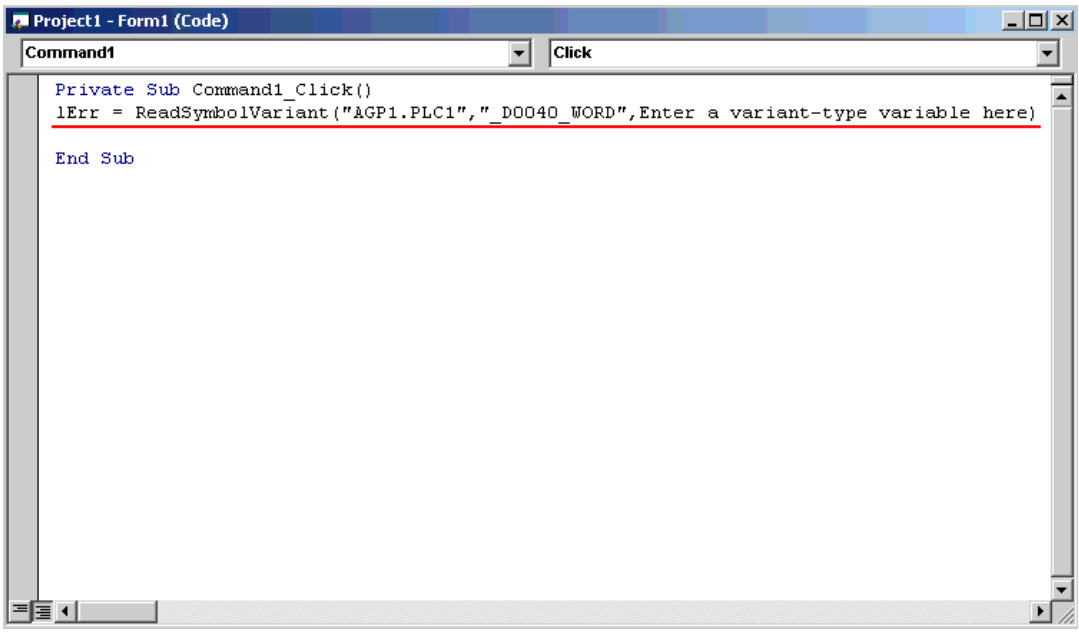
10 Select [Programming Assist] - [VB & VBA] - [Read Function] on the menu.



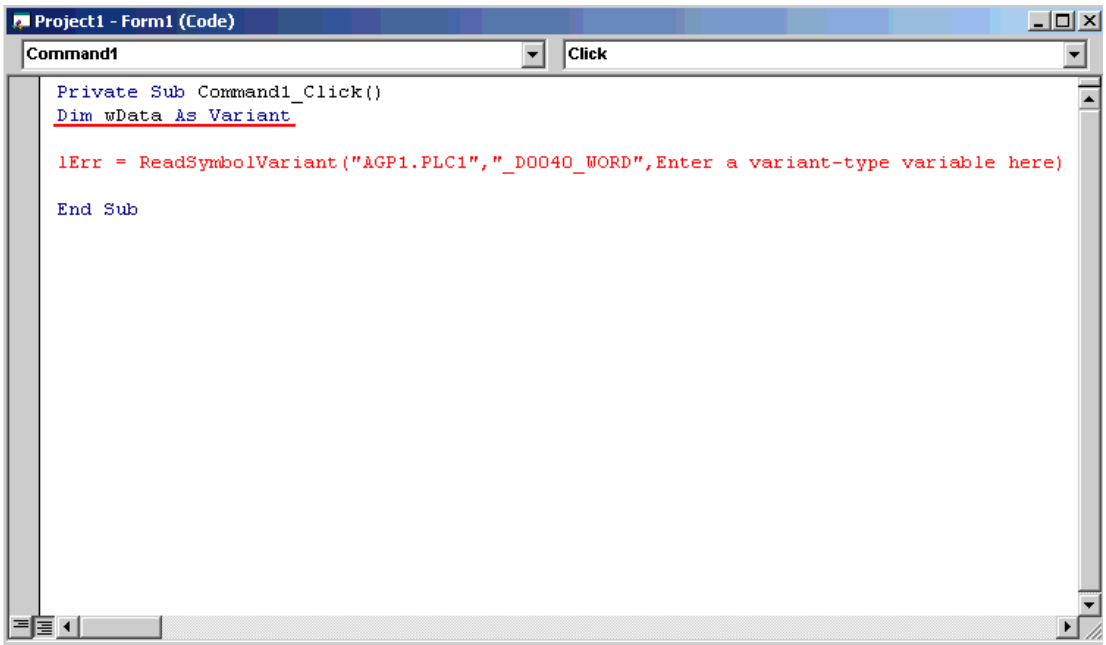
The read function is copied to the clipboard.



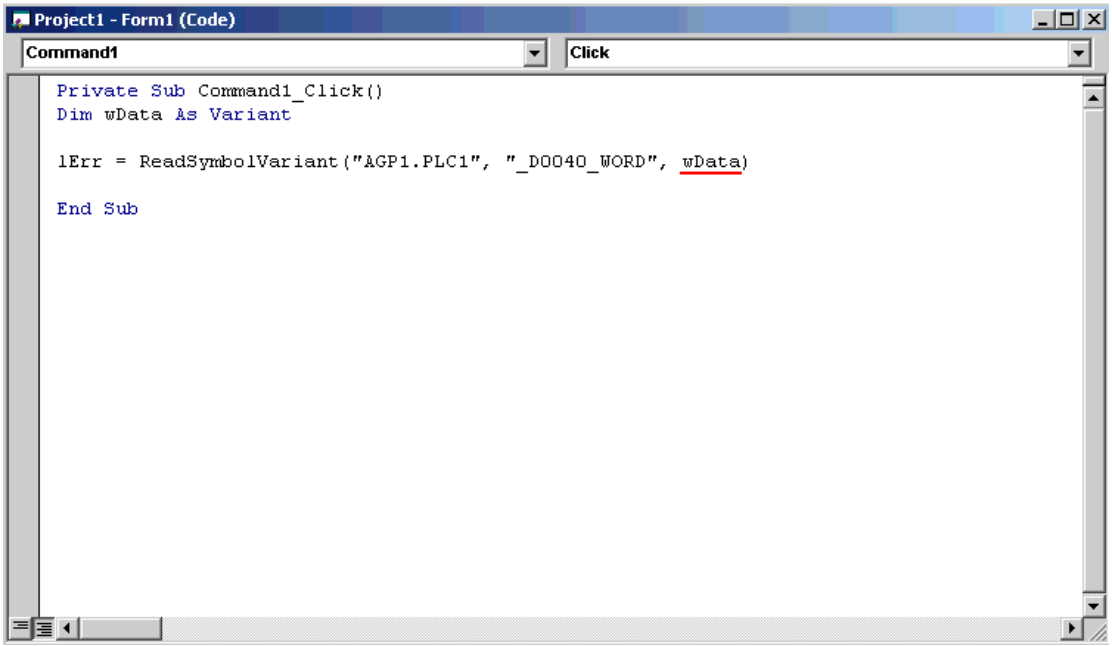
11 Double-click [Command1] on [Form1], and paste the data on the clipboard (read function) between 'private sub Command1_Click()' and 'End Sub'.



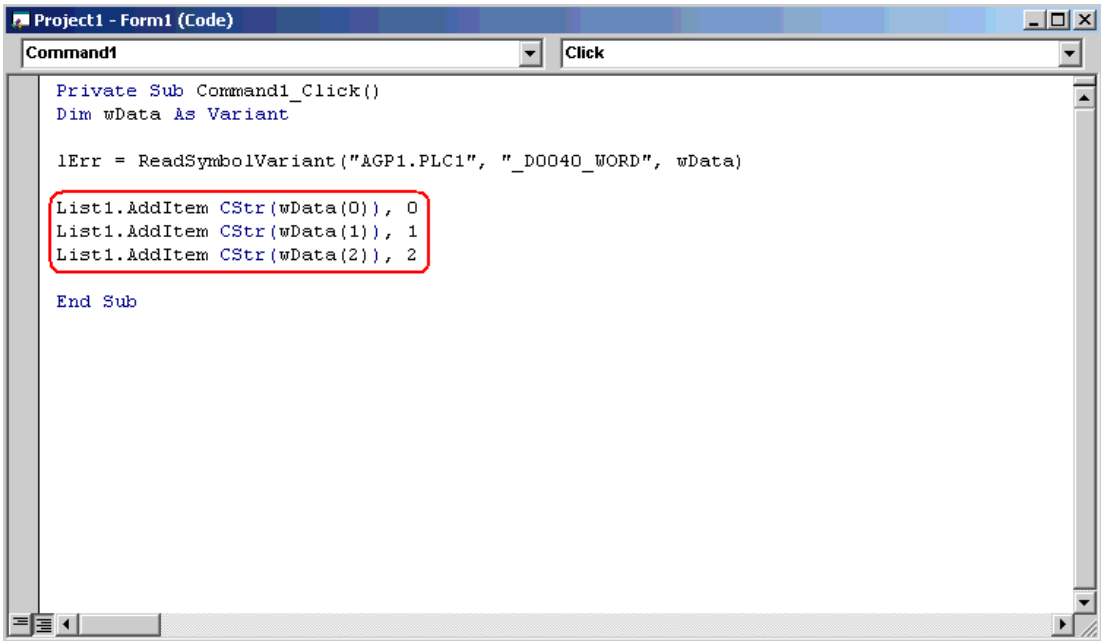
12 Declare the area (Array) to store the read data. Ensure that the array type (in this example, Variant-type) is matched with the data type of the symbol being used.



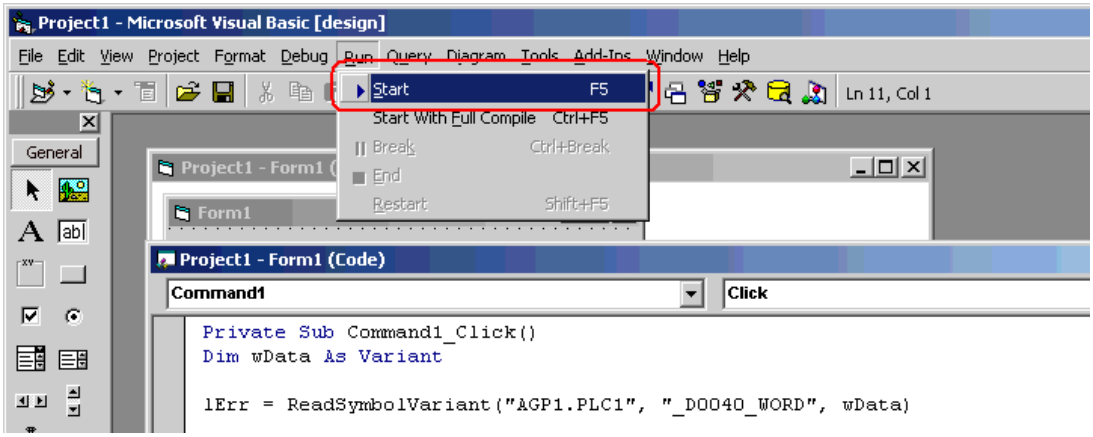
13 Specify the first area (wData) to store the read data.



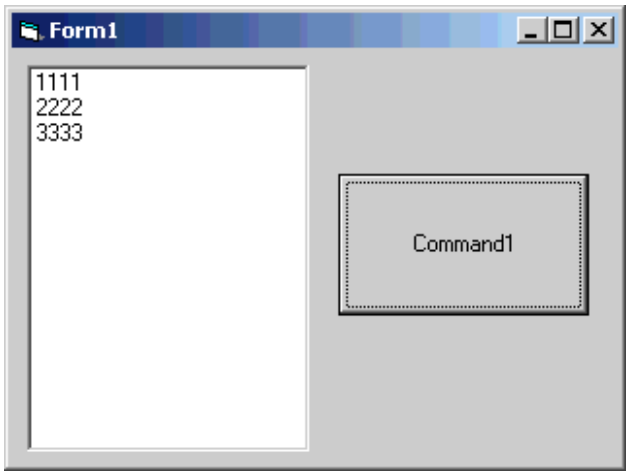
14 The List Box displays the read data for three points (wData(0), wData(1) and wData(2)) in sequence.



15 Select [Start] from [Run] on the Microsoft Visual Basic menu.

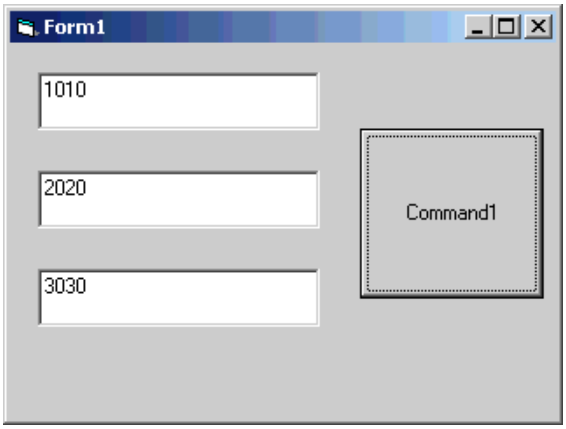


16 Click [Command1]. Then, the List Box displays the data for three points from the symbol "_D0040_WORD".

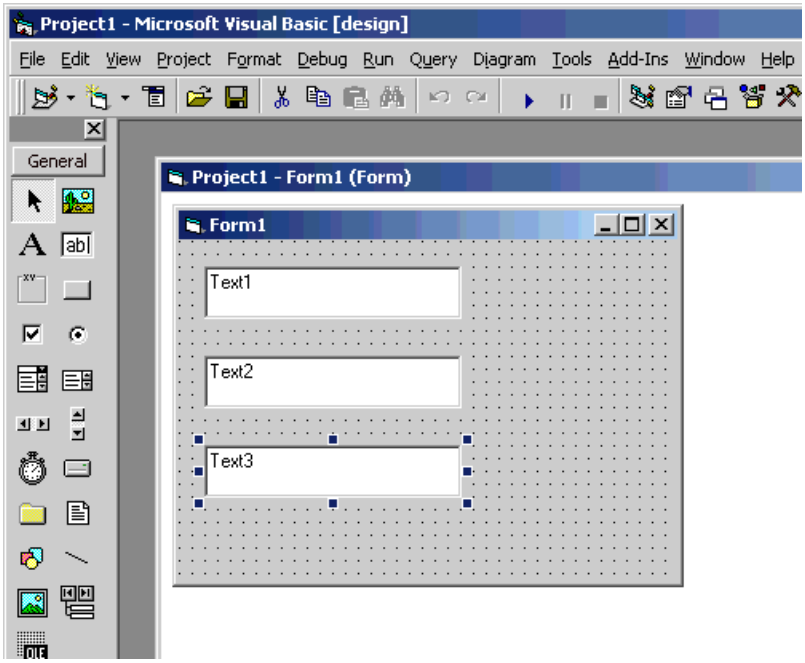


Creating "Writing" application

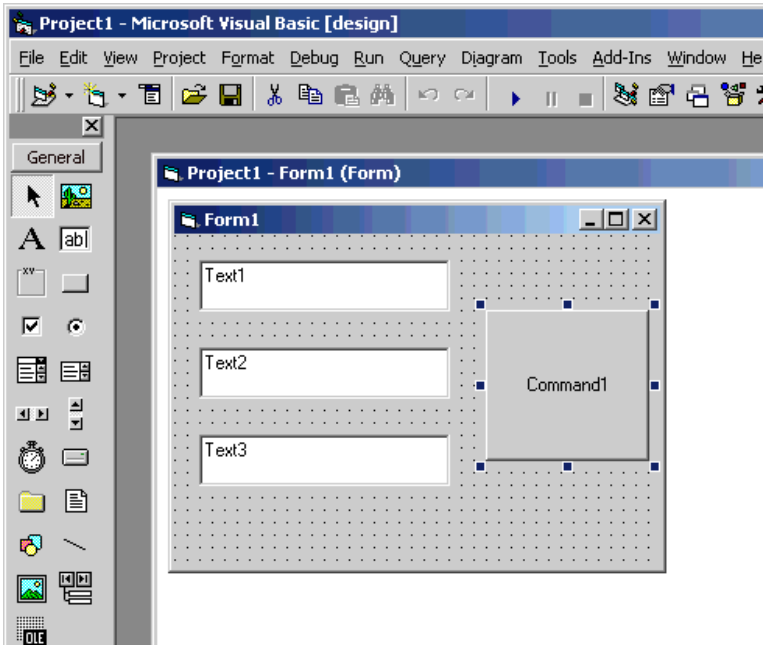
This section describes the procedure for creating an application that writes the data (16-bit signed data) entered for three points with a click on [Command1].



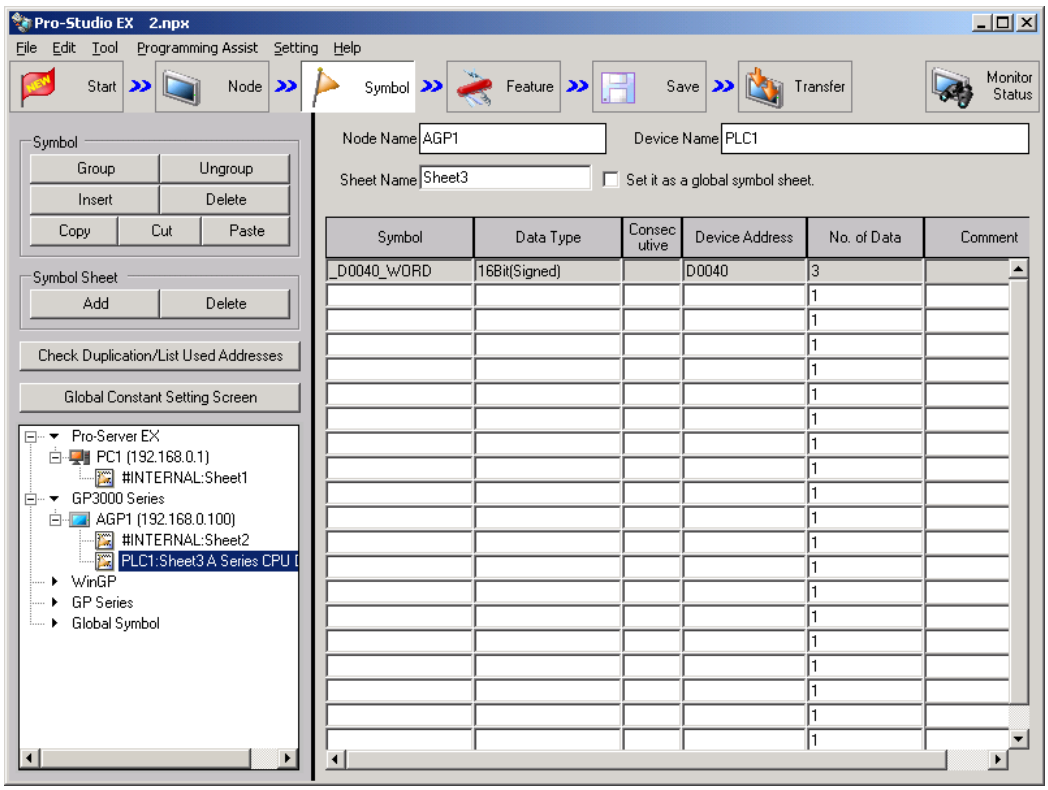
17 Select [TextBox] and paste it to [Form1]. Paste [Text Box] for three items.



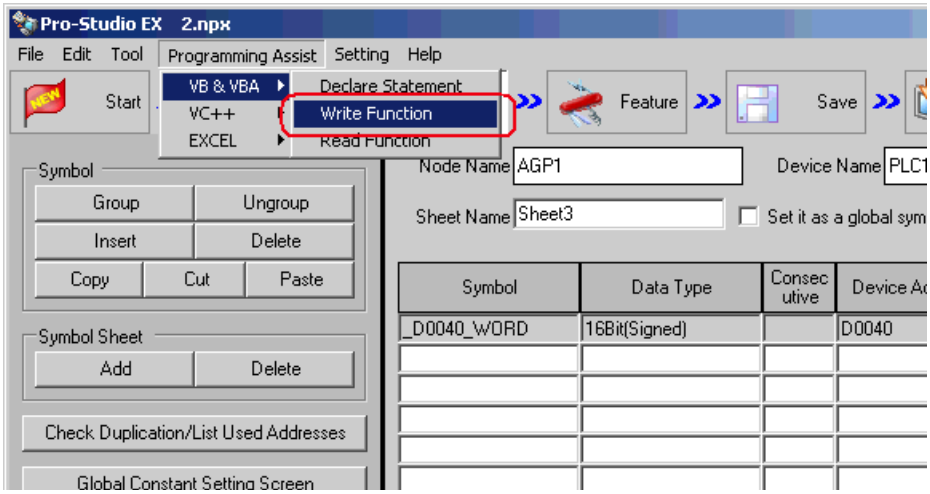
18 Select [CommandButton] and paste it [Form1].



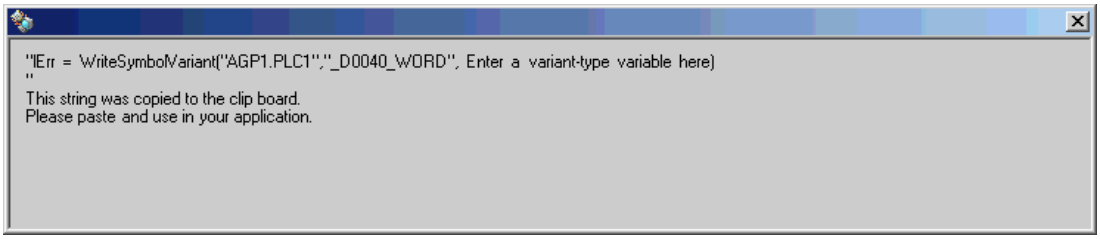
19 Select a target symbol name from those registered in 'Pro-Server EX'. (Select the symbol with first-address for writting.)



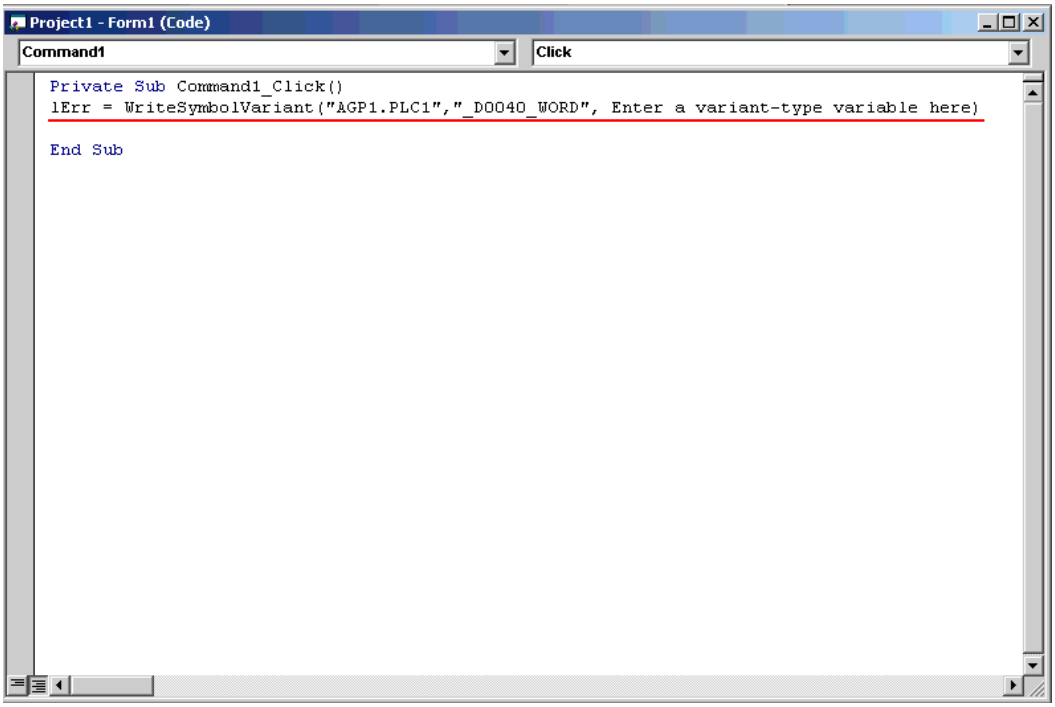
20 Select [Programming Assist] - [VB & VBA] - [Write Function] on the menu.



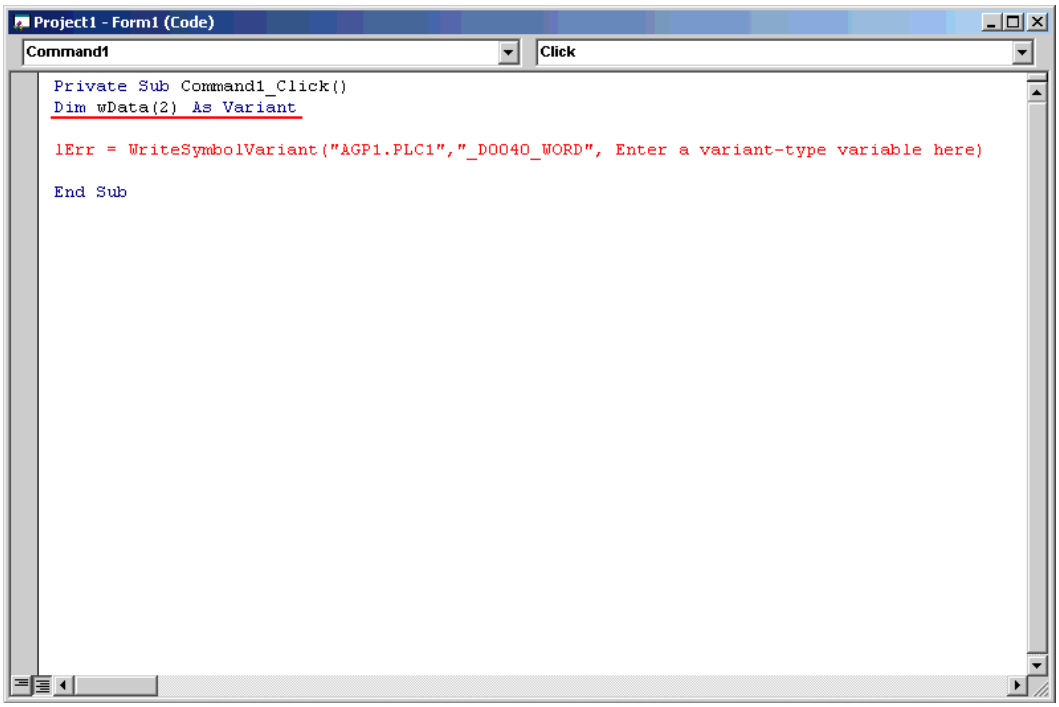
The write function is copied to the clipboard.



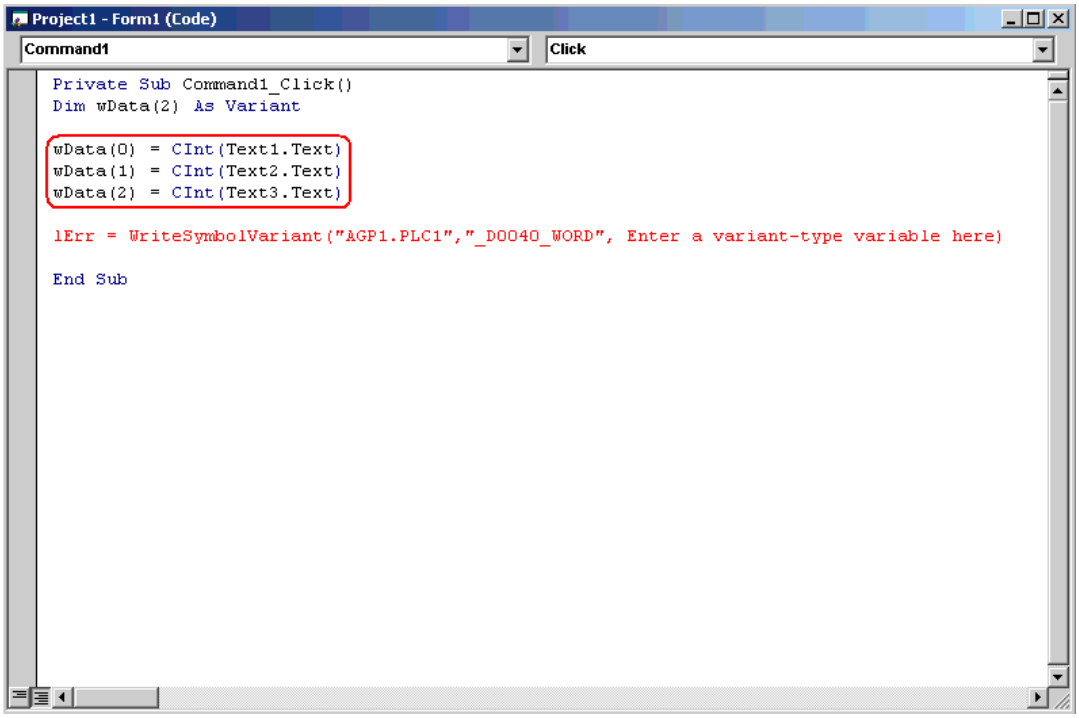
21 Double-click [Command1] on [Form1], and paste the data on the clipboard (write function) between the Sub statement and the End Sub statement.



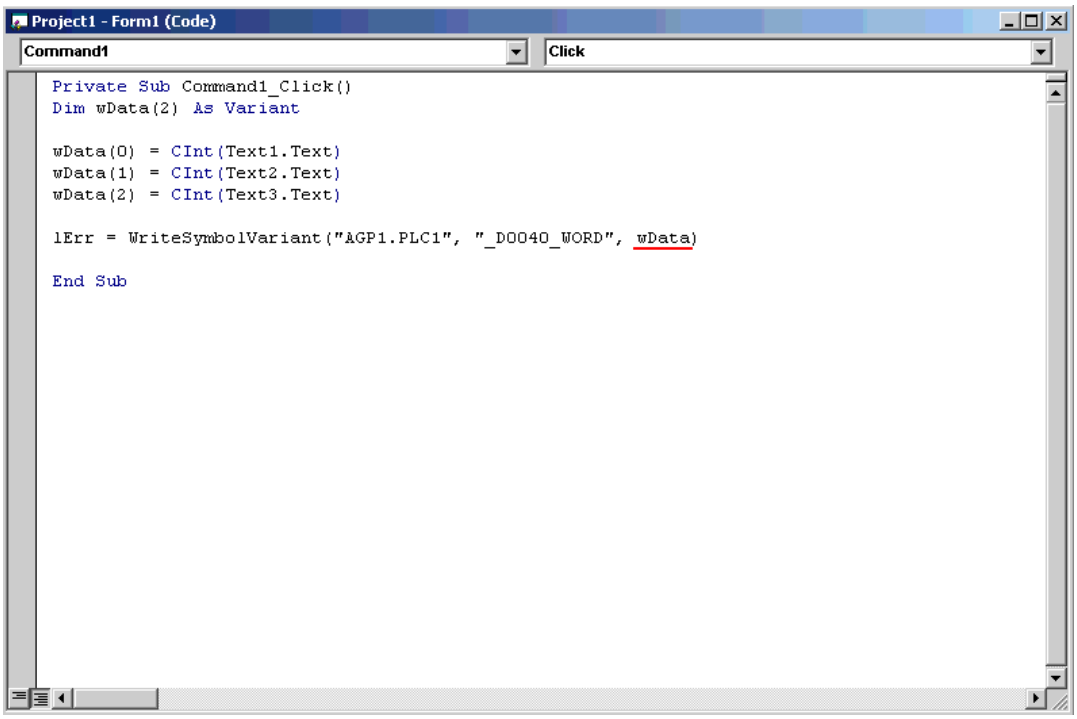
22 Declare the area (alignment) to store the written data. Ensure that the alignment type (in this example, Variant-type) is matched with the data type of the symbol being used.



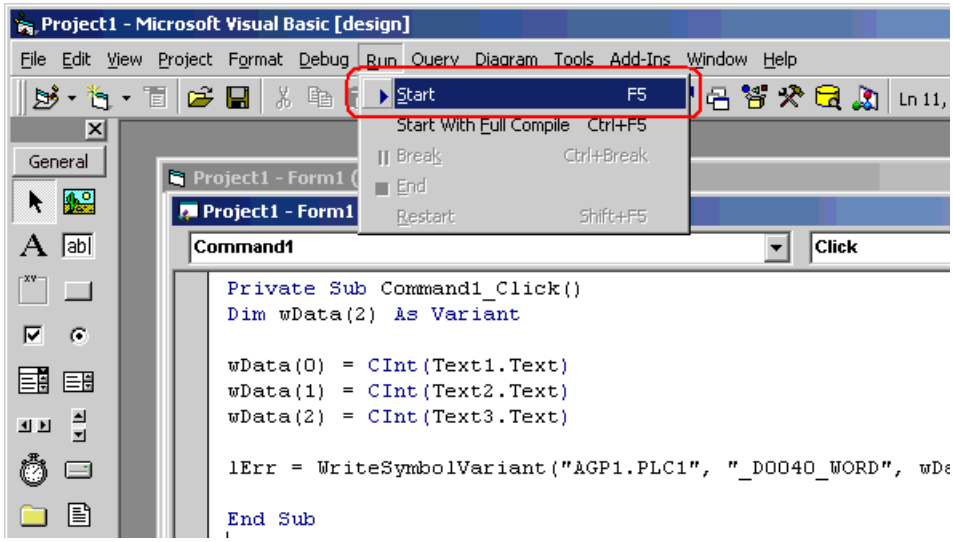
23 Set the data entered in [TextBox] into the alignment.



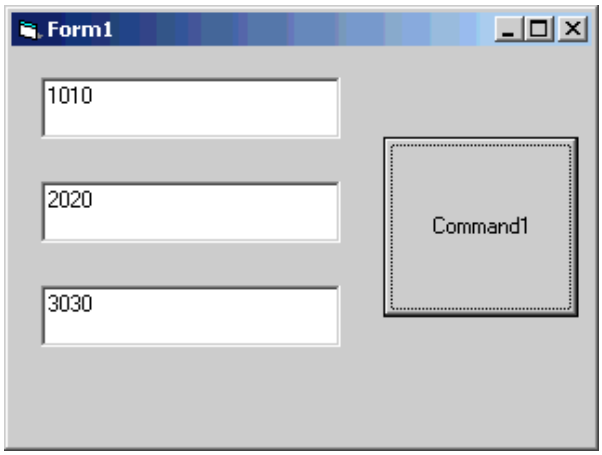
24 Specify the first area (wData) where the written data has been set.



25 Select [Start] from [Run] on the Microsoft Visual Basic menu.



26 After entering values (for three points) in [TextBox], click [Command1]. Then, 'Pro-Server EX' executes the writing of the data for three points from the symbol "_D0040_WORD".

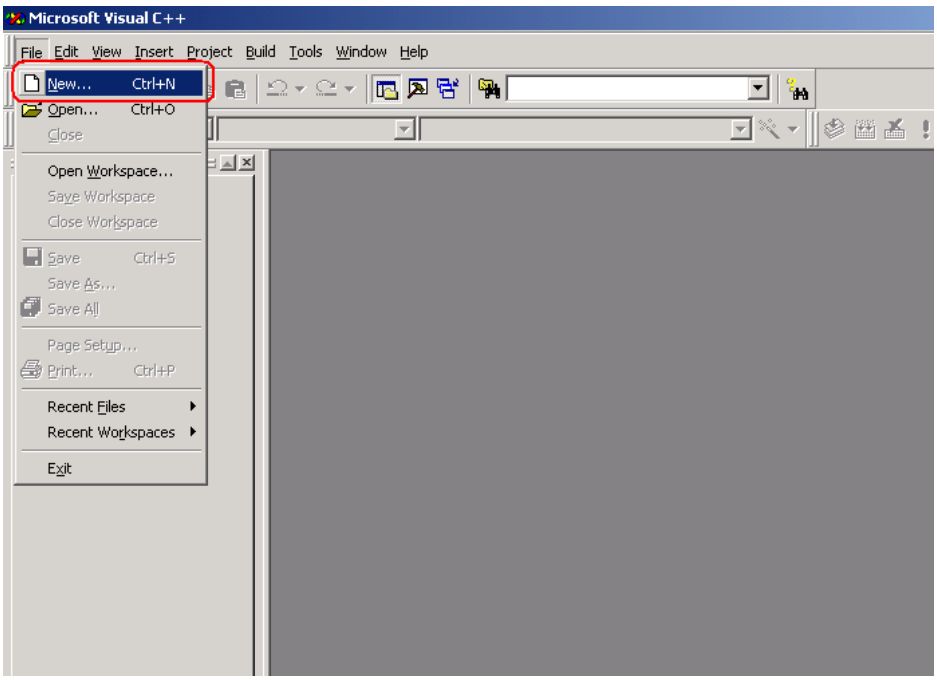


26.10.2 VC Support Function

For example, this section describes the procedure for creating a dialog-based application by using MFC (Microsoft Foundation Class).

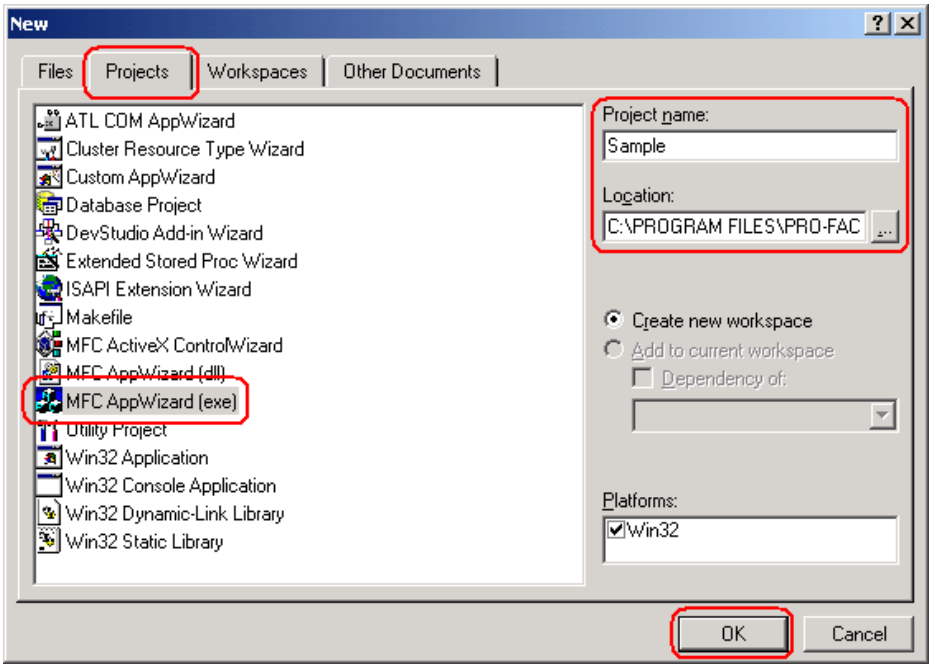
VC: Declaration statement

1 Start Microsoft Visual C++, and select [New] from [File].

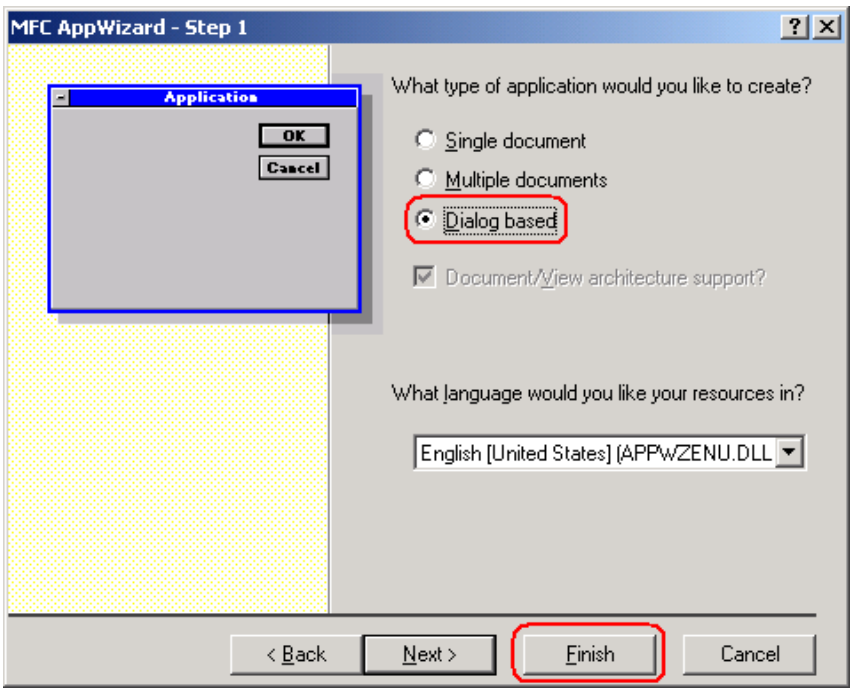


2 After selecting [MFC AppWizard(exe)] in the [Projects] tab, enter [Project name] and [Location], and click the [OK] button.

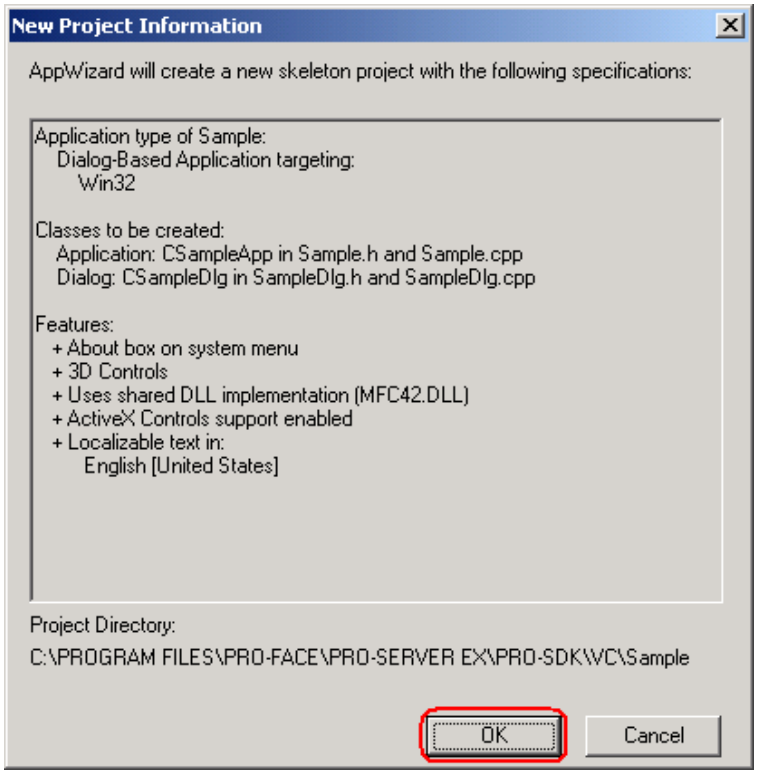
In this example, "Sample" is entered for [Project name], and "C:\Program Files\Pro-face\Pro-Server EX\PRO-SDK\VC" is entered for [Location].



3 Select [Dialog Based] for "What type of application would you like to create?", and click the [Finish] button.

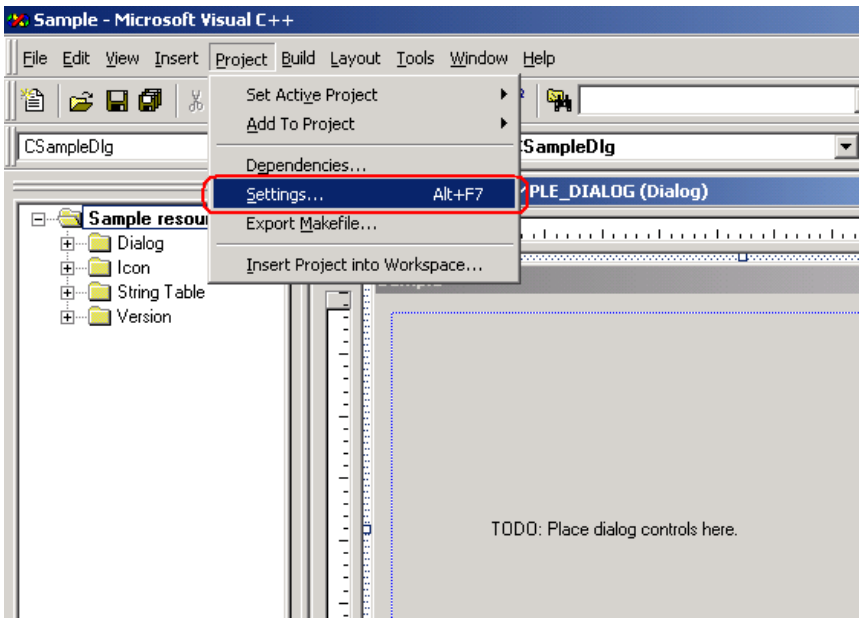


4 Click the [OK] button to complete the project.

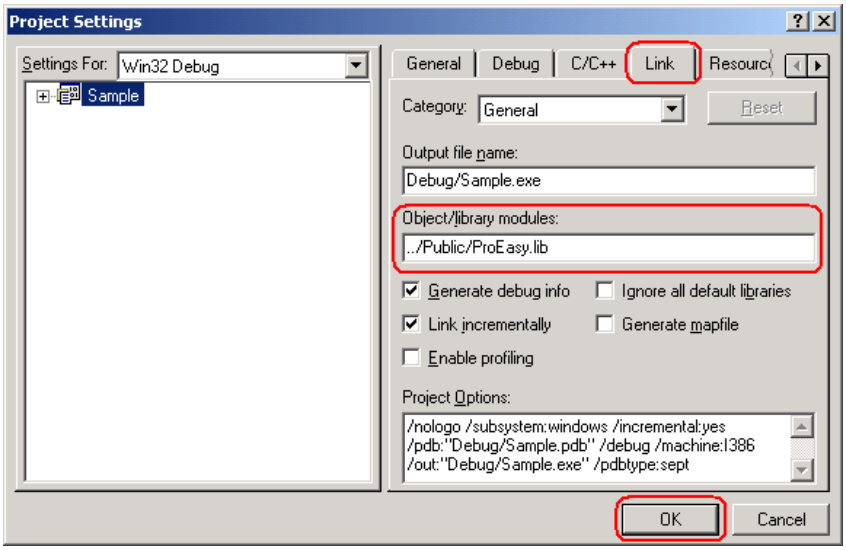


The read/write functions provided by 'Pro-Server EX' are available as DLL. To use DLL, you must specify a LIB file.

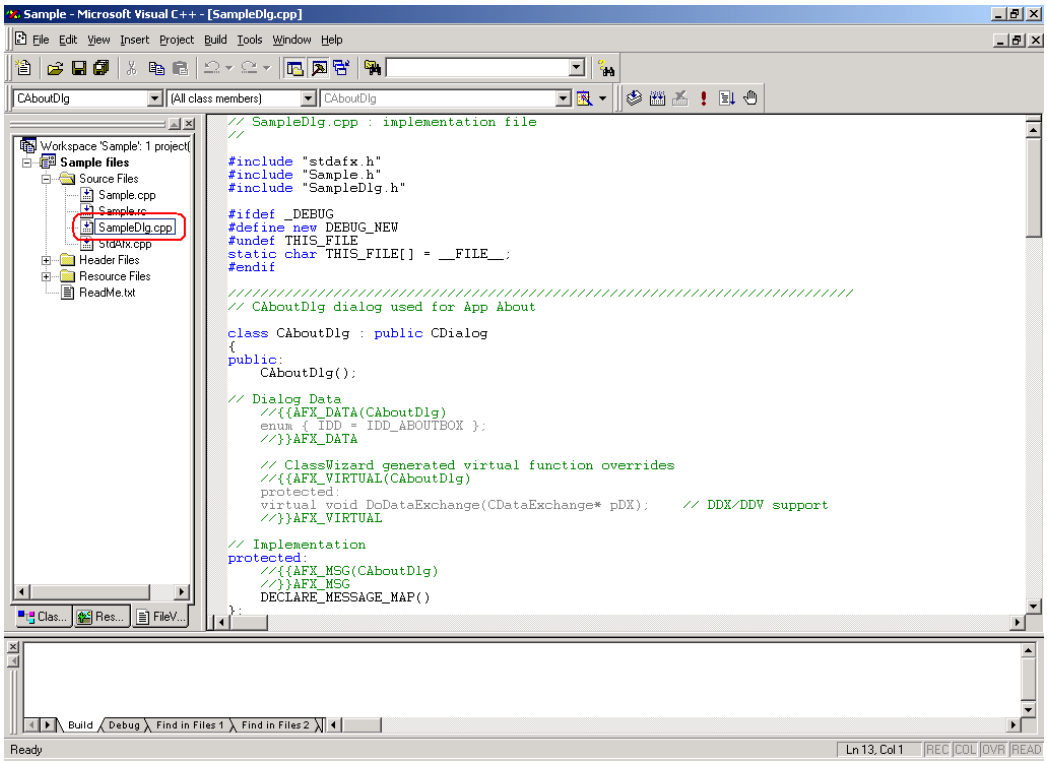
5 Select [Settings] from [Project] on the Microsoft Visual C++ menu.



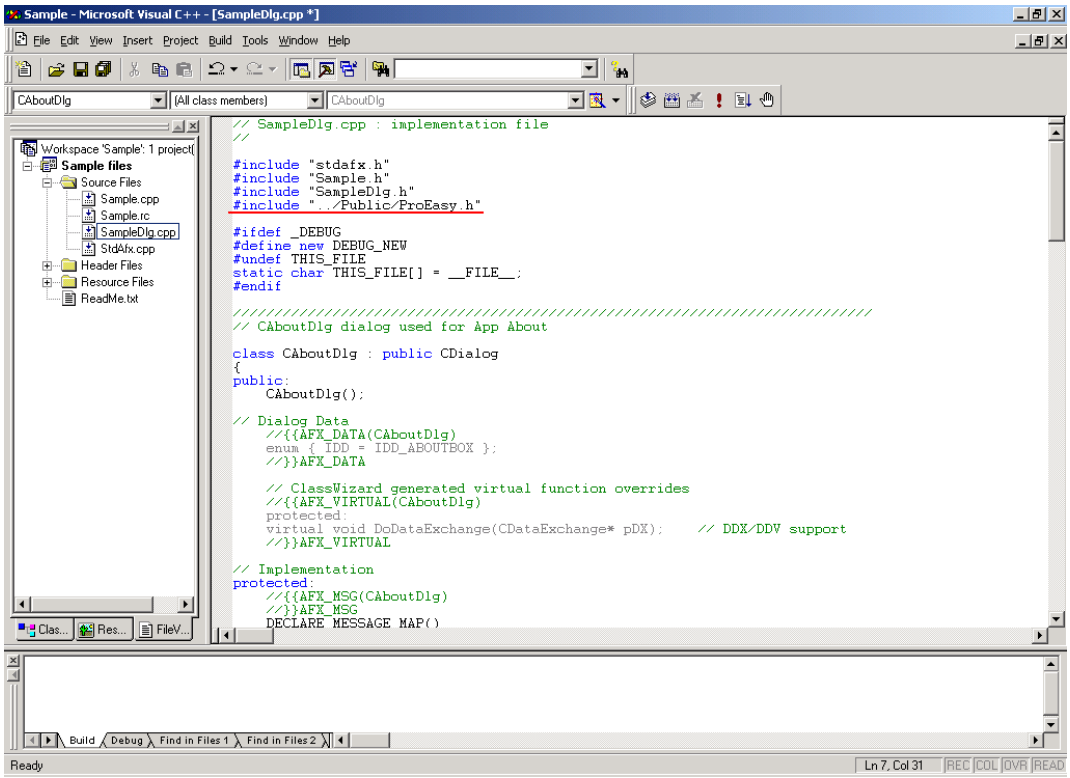
6 Specify a LIB file for [Object/library modules] in the [Link] tab. Then, click the [OK] button.
The LIB file (ProEasy.lib) exists in "PRO-SDK\Vc\Public" in the folder where 'Pro-Server EX' has been installed.
In this example, "..\Public\ProEasy.lib" is specified.



7 To use read/write functions provided by 'Pro-Server EX', you must include a header file (ProEasy.h). After clicking the [FileView] tab in the [Work Space] window of Microsoft Visual C++, double-click the "SampleDig.cpp" file.
In this example, the read/write functions are used in the "SampleDig.cpp" file.



8 Add #include "..\Public\ProEasy.h" to the "SampleDig.cpp" file. This completes the function (read/write function) decleration procedure.



The above 1 to 8 steps apply to both reading and writing applications.

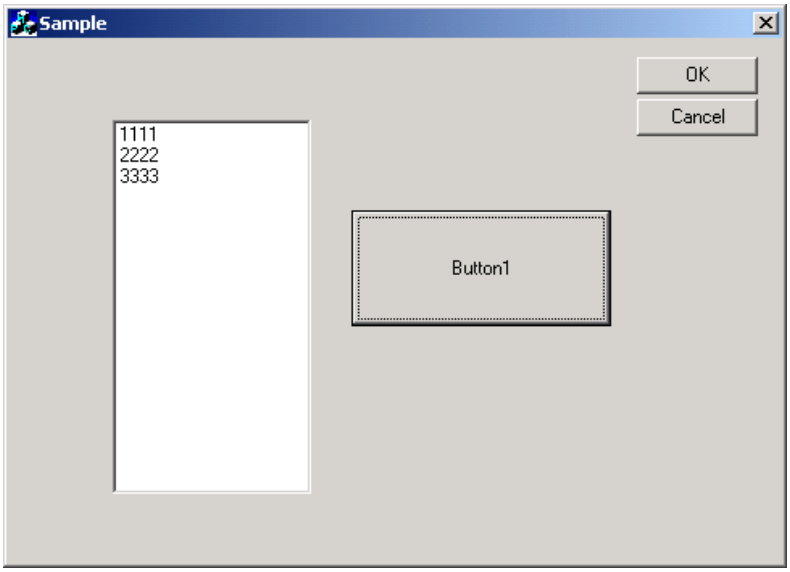
The following procedure varies depending on whether the application is intended for reading or writing, and so is explained individually.

To create a "Reading" application, refer to steps 9 to 30.

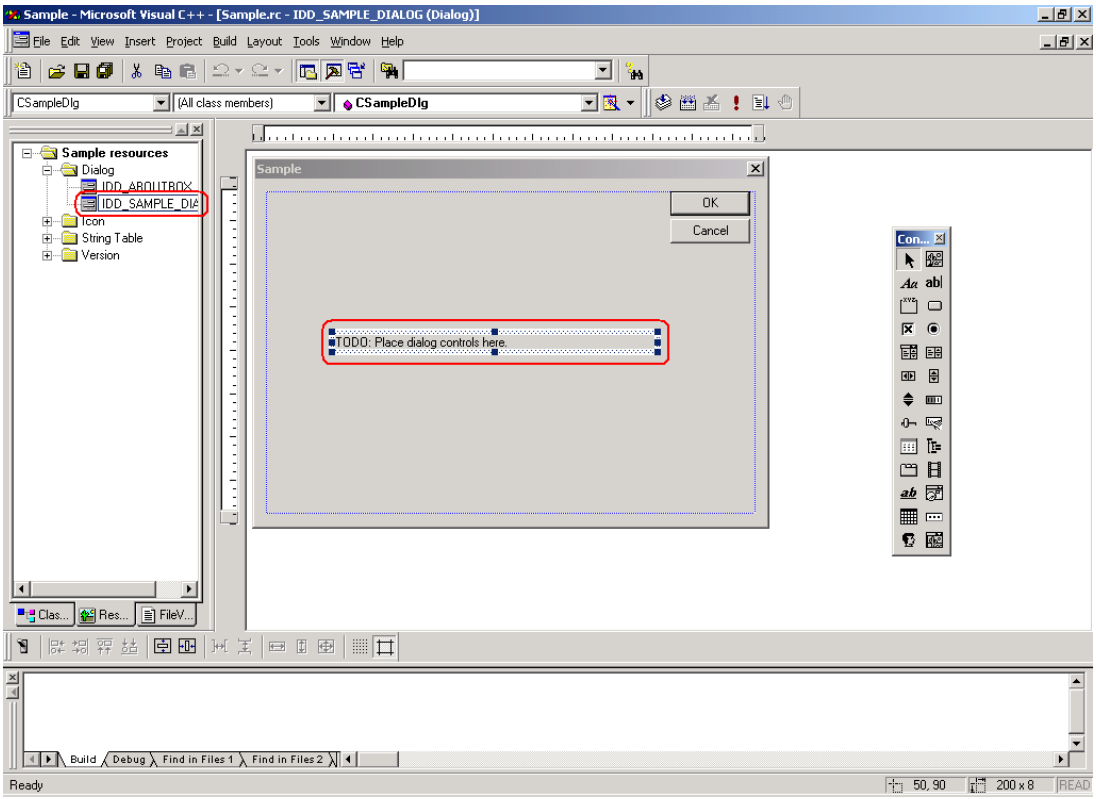
To create a "Writing" application, refer to steps 31 to 47.

Creating "Reading" application

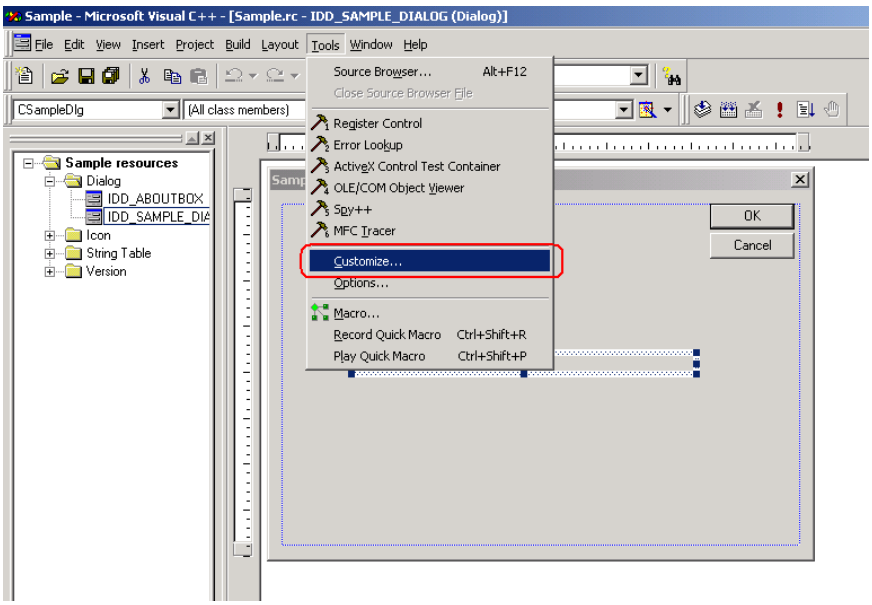
This section describes the procedure for creating an application that reads and displays data (16-bit signed data) for three points with a click on [Button1].



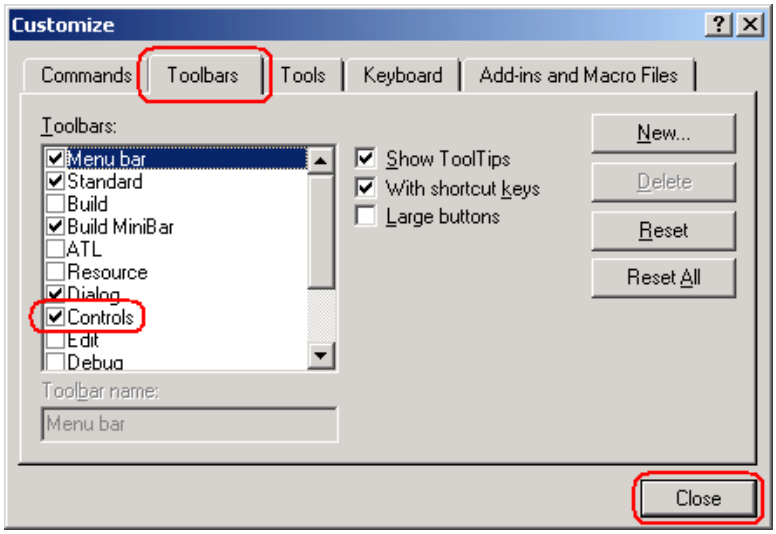
9 After clicking the [ResourceView] tab in the [Work Space] window of Microsoft Visual C++, double-click [IDD_SAMPLE_DIALOG].
Select [Static Text] at the center of the dialog box, and delete it.



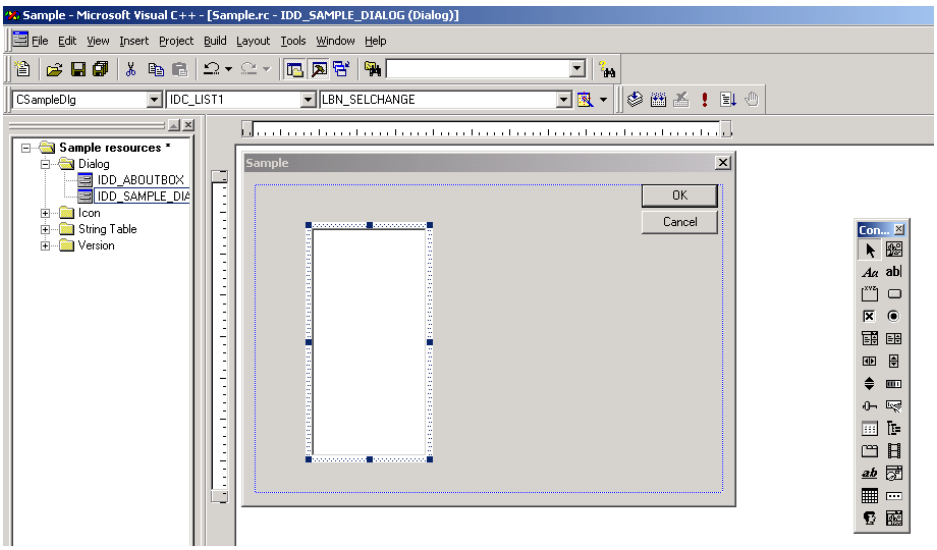
10 Select [Customize] from [Tools] on the Microsoft Visual C++ menu.



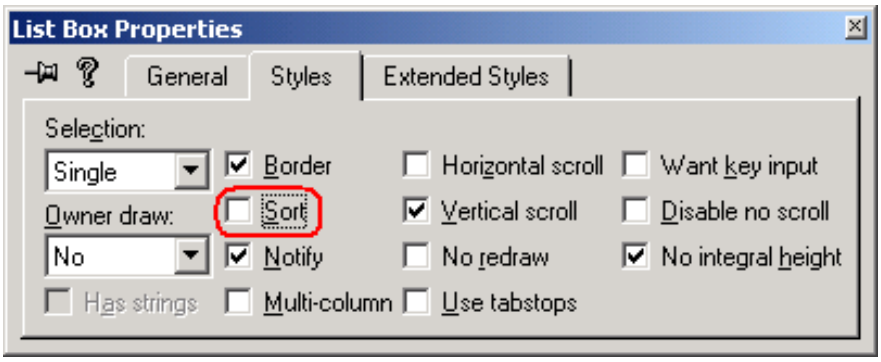
11 Check the [Controls] checkbox in the [Toolbars] tab, and click the [Close] button.



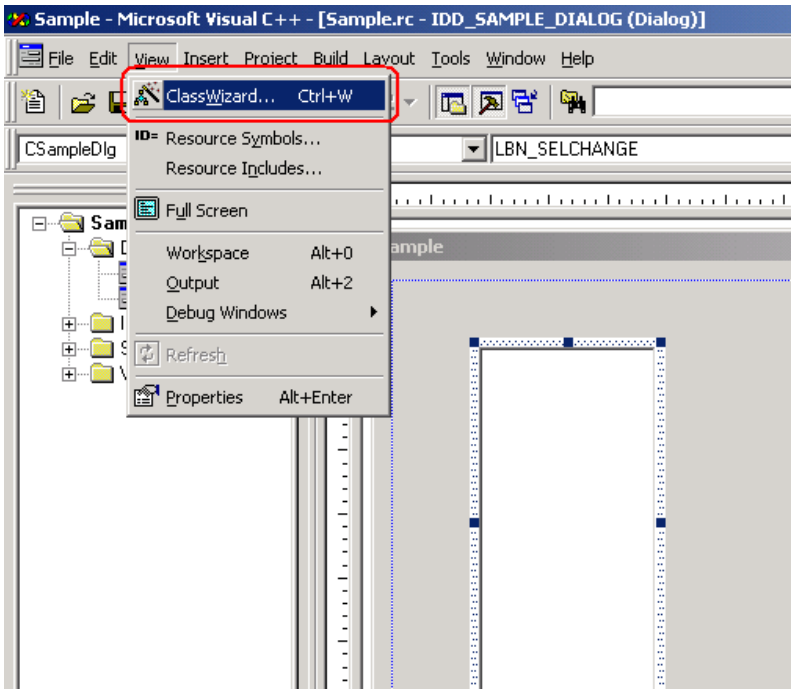
12 Select [ListBox], and paste it to the dialog box.



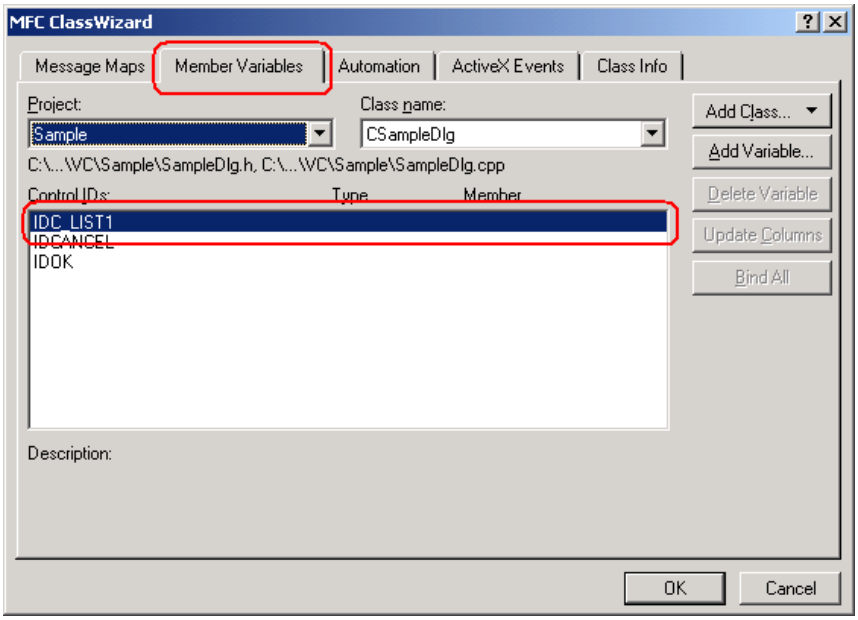
13 Right-click the pasted [ListBox], and select [Property]. The [List Box Propertis] dialog box appears. Then, uncheck the [Sort] checkbox.



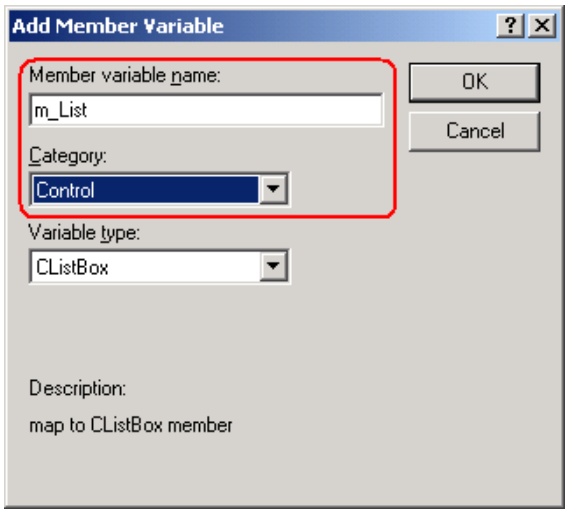
14 Select [ClassWizard] from [View] on the Microsoft Visual C++ menu.



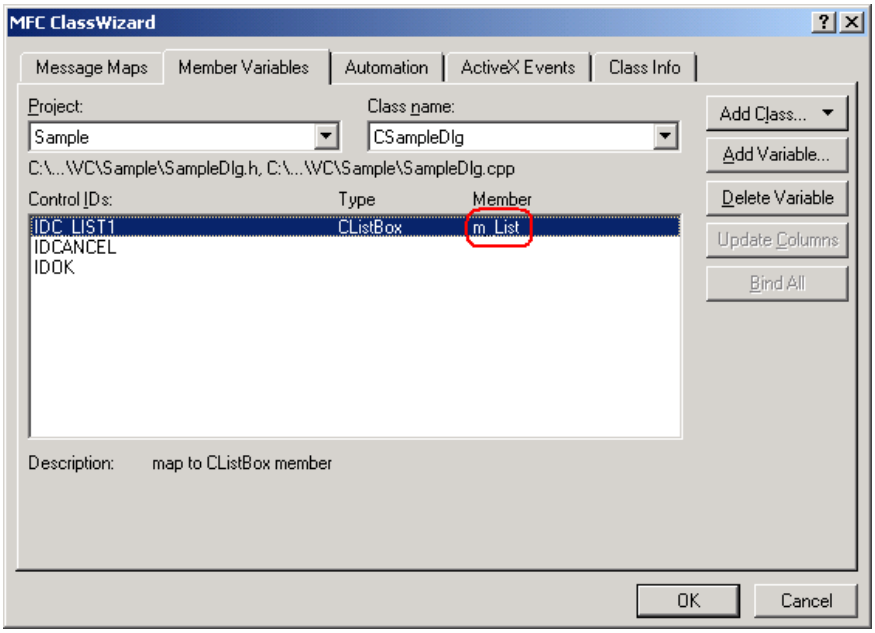
15 Select the [Member Variables] tab, and select "IDC_LIST1" for [Control IDs].



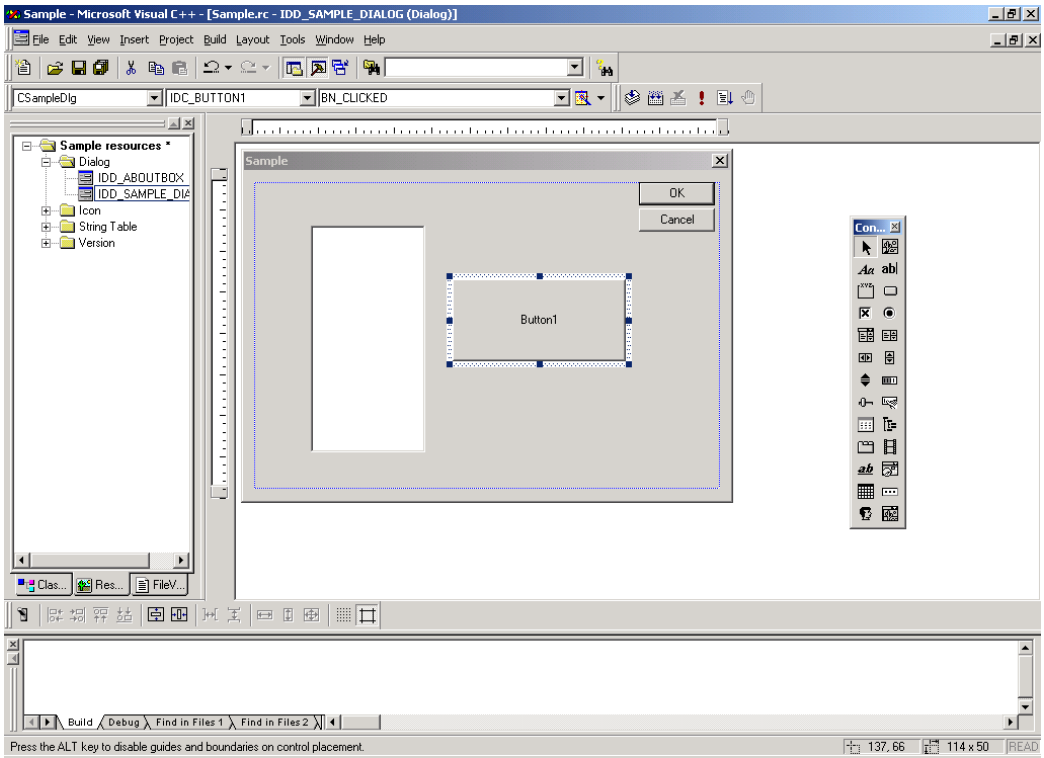
16 Click [Add Variable], and enter "m_List" for [Member variable name]. After selecting "Control" for [Category], click the [OK] button.



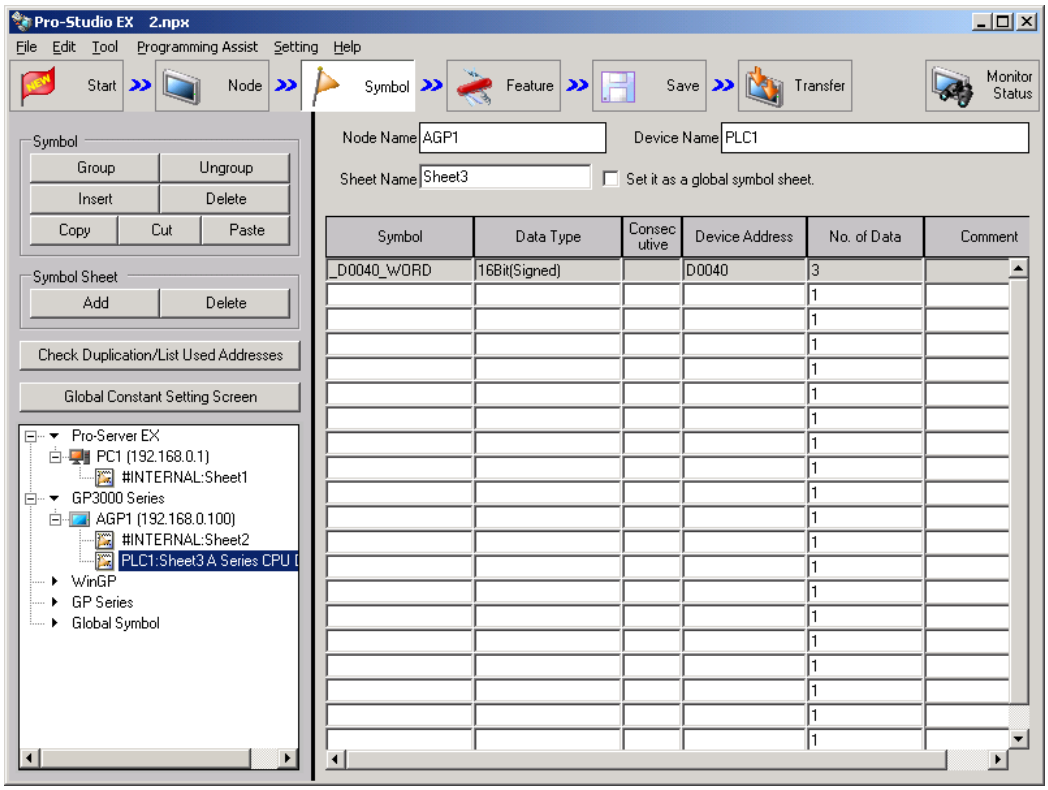
17 After confirming that the member variable has been added, click the [OK] button.



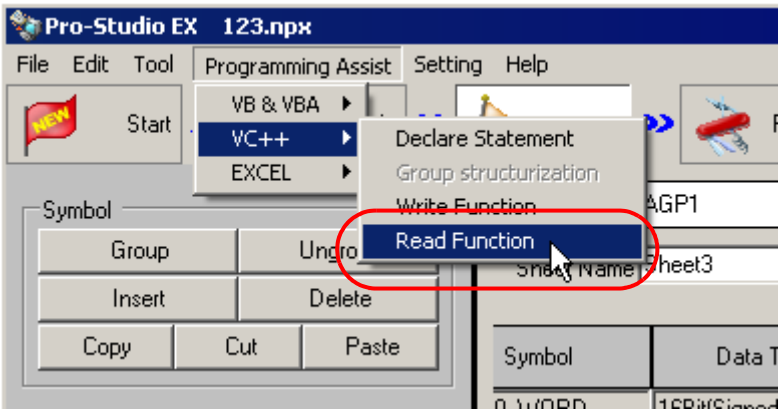
18 Select [Button], and paste it to the dialog box.



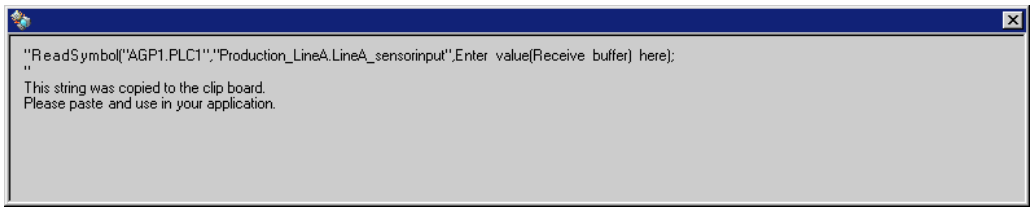
19 Select a target symbol name from those registered in 'Pro-Studio EX'. (Select the symbol with first-adress for reading.)



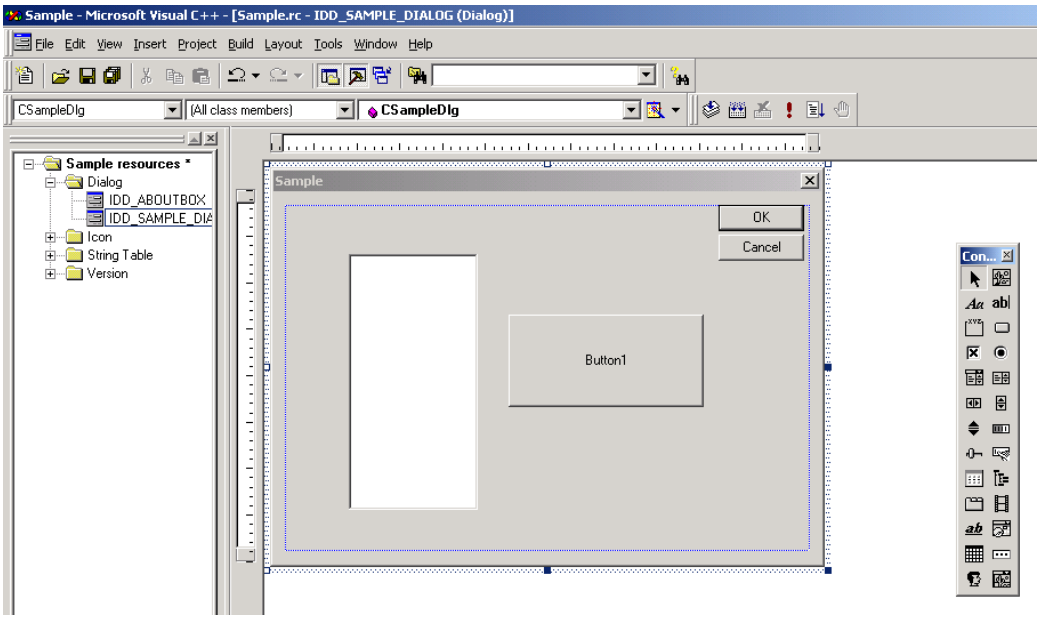
20 Select [VC++] - [Read Function] from "Programming Assist" on the menu.



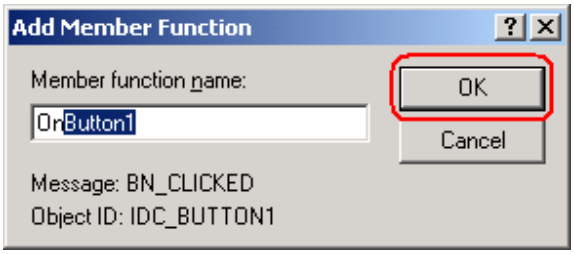
The read function is copied to the clipboard.



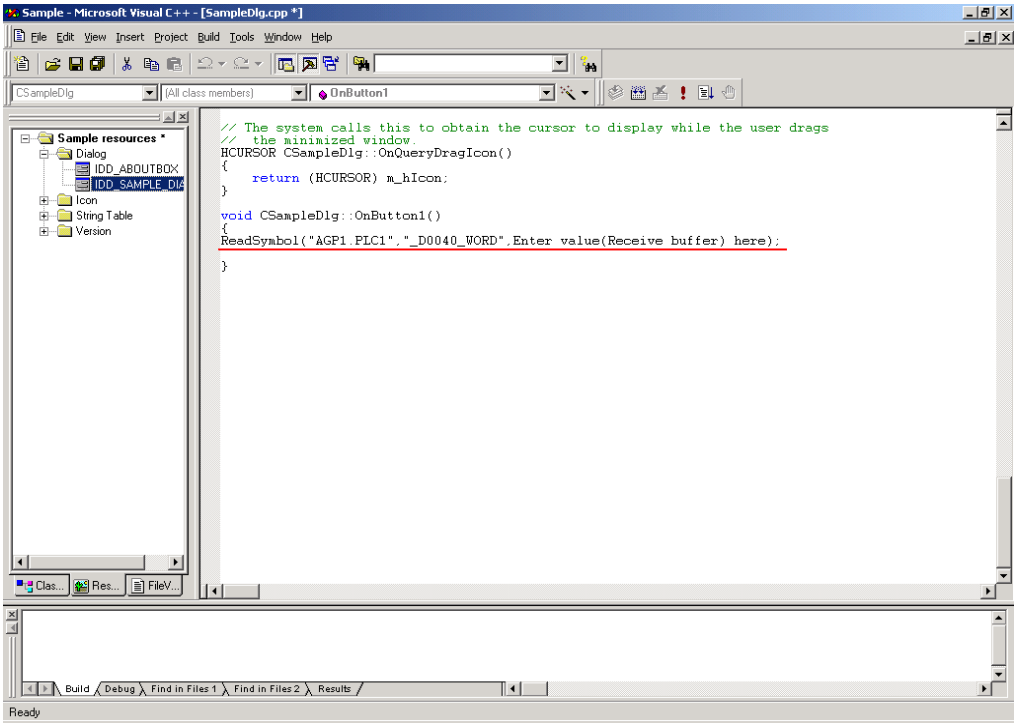
21 Double-click [Button1] that has been pasted to [Dialog] in Microsoft Visual C++.



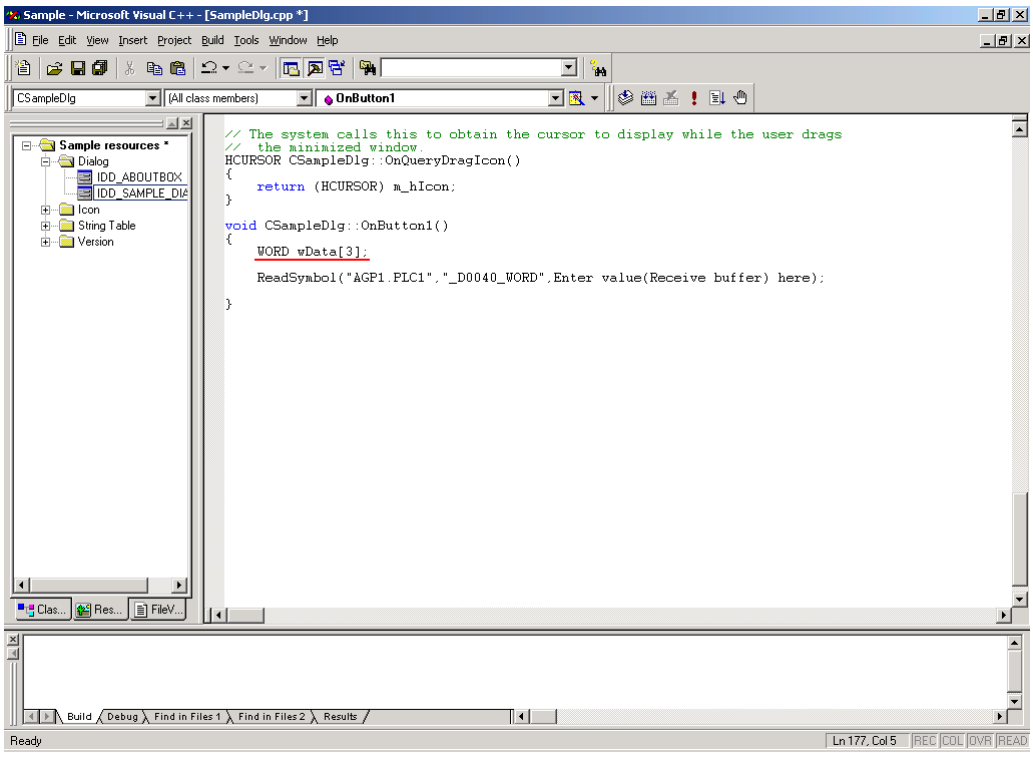
22 Click the [OK] button.



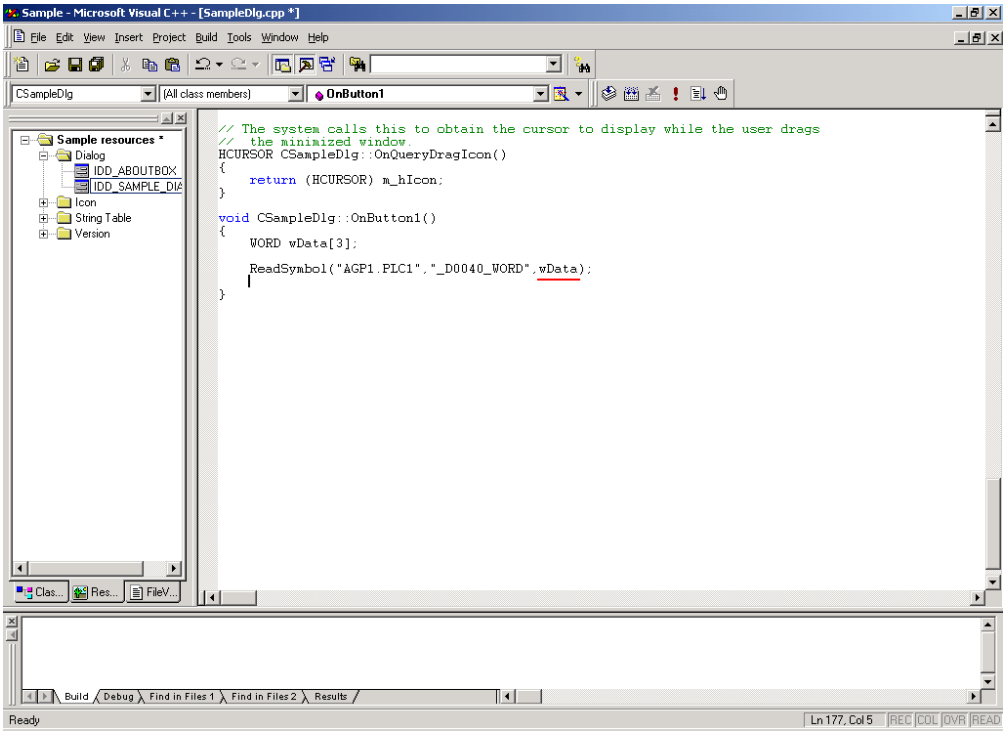
23 Paste the data on the clipboard (read function) into the OnButton1 member function.



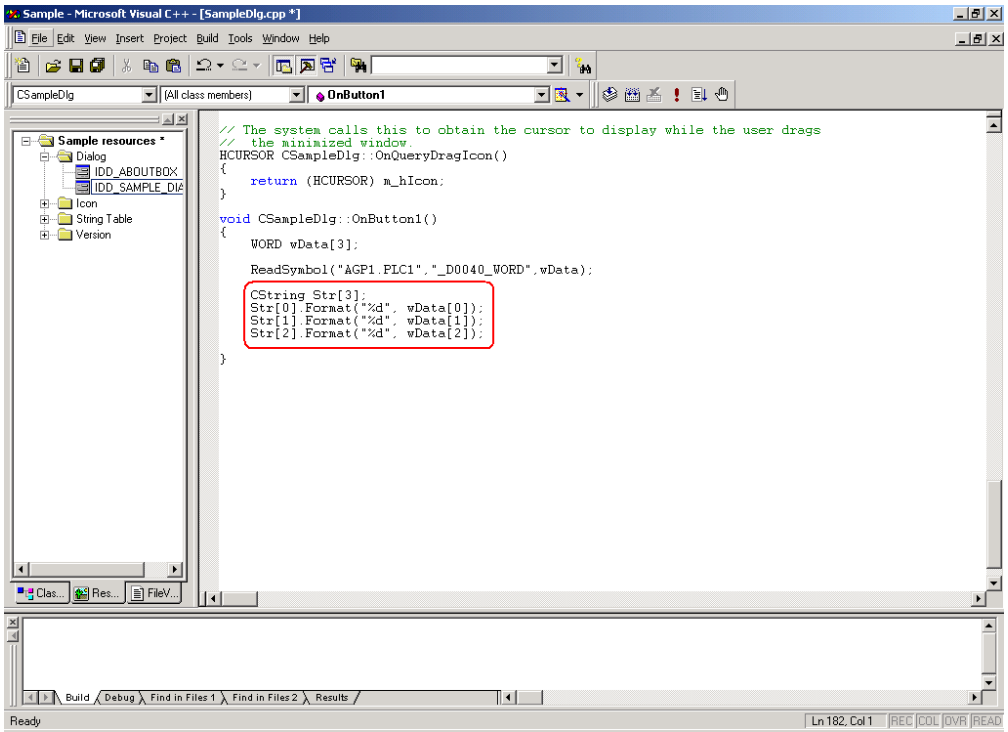
24 Declare the area (Array) to store the read data.



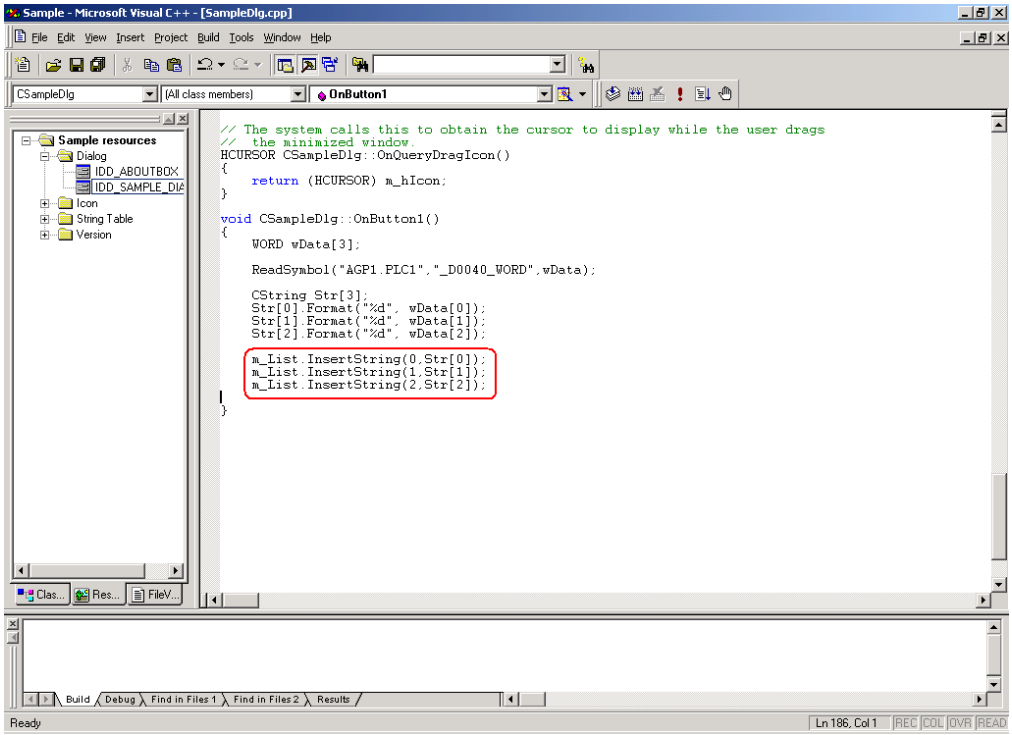
25 Specify the first area (wData) to store the read data.



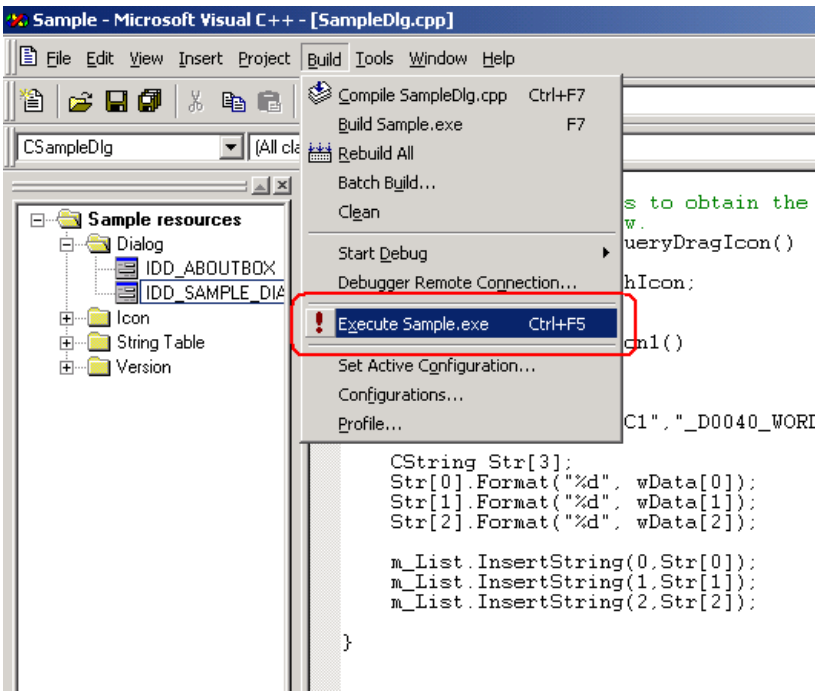
26 To display the read data for three points (wData(0), wData(1) and wData(2)) in the list box, convert the data into Cstring-type string data.



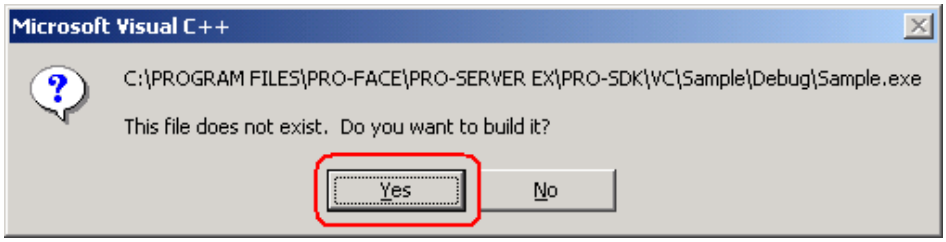
27 The list box (m_List) displays the read data (that has been converted into string data) in sequence.



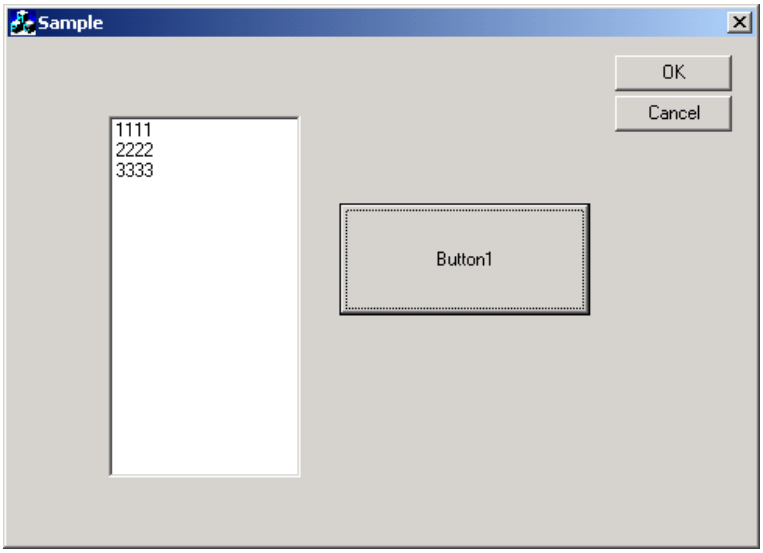
28 Select [Execute Sample.exe] from [Build] on the Microsoft Visual C++ menu.



29 Click the [Yes] button.



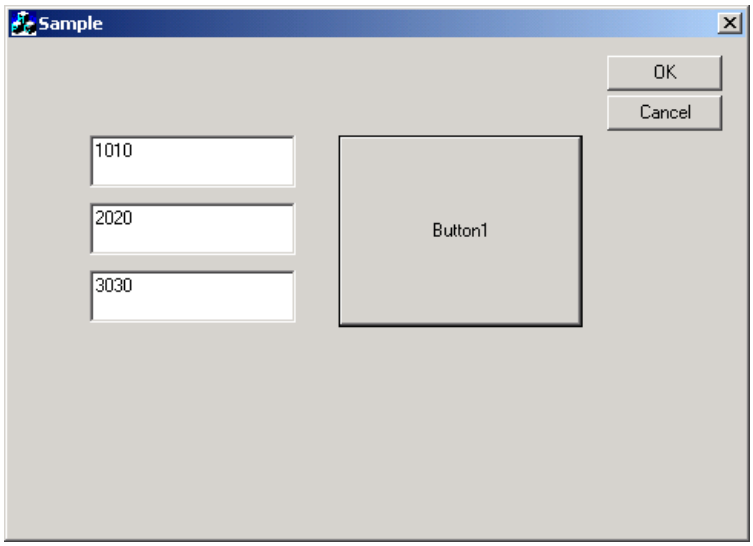
30 Click [Button1]. Then, the list box displays the data for three points from the symbol "_D0040_WORD".



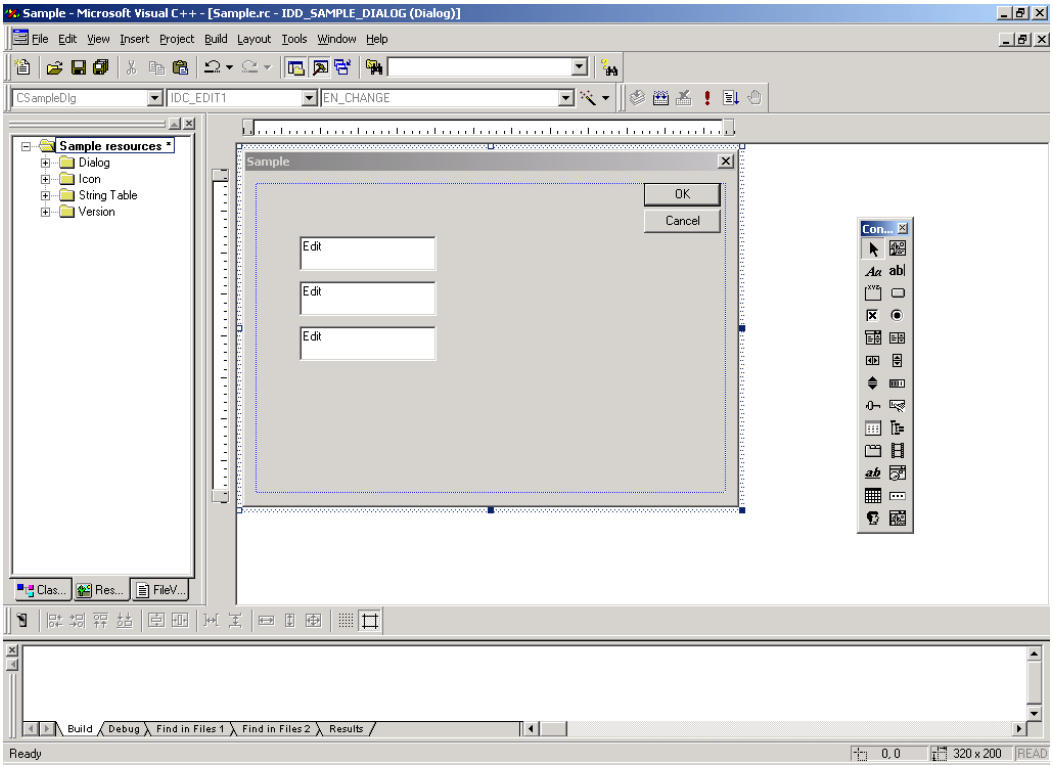
Creating "Writing" application

This section describes the procedure for creating an application that writes the data entered for three points with a click on [Button1].

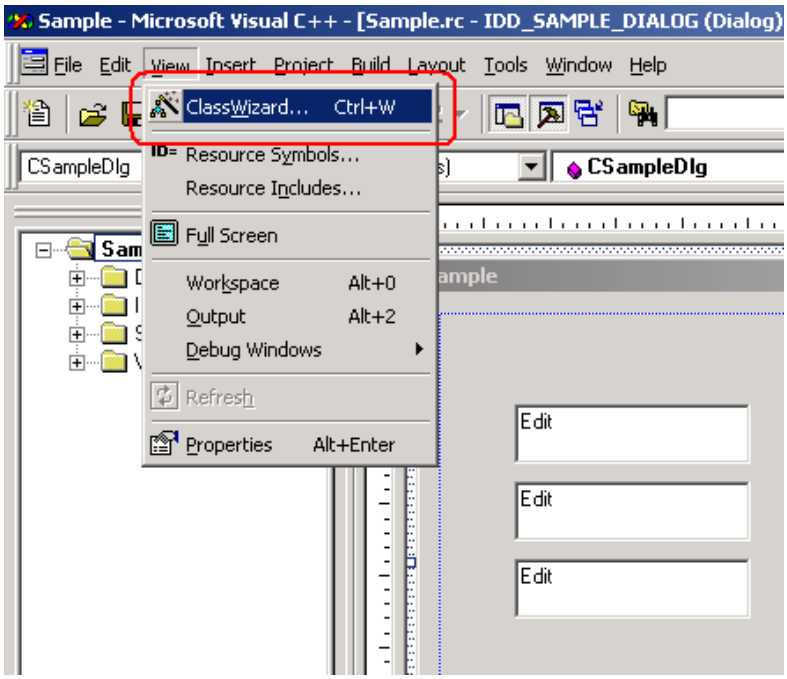
Steps 9 to 11 are the same as those for creating "Reading" application.



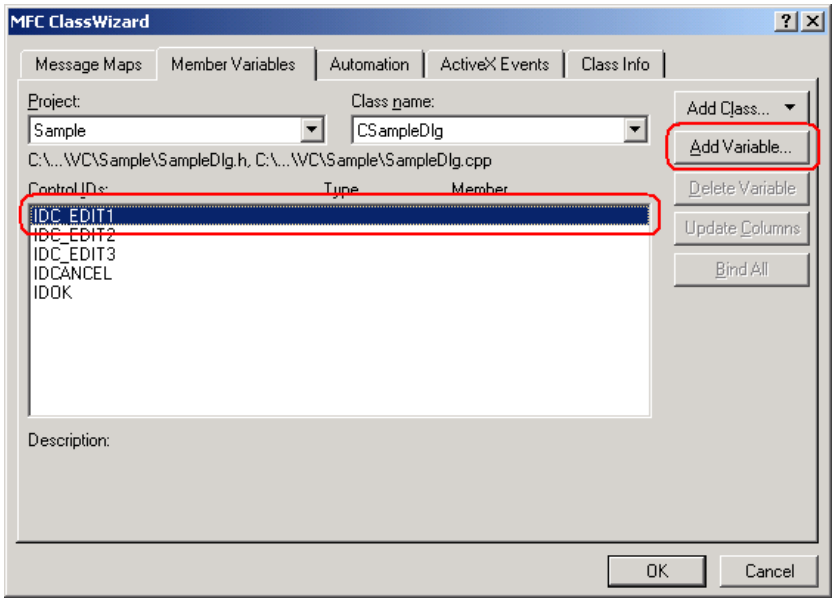
31 Select [EditBox], and paste it to [Dialog]. Paste [Edit Box] for three items.



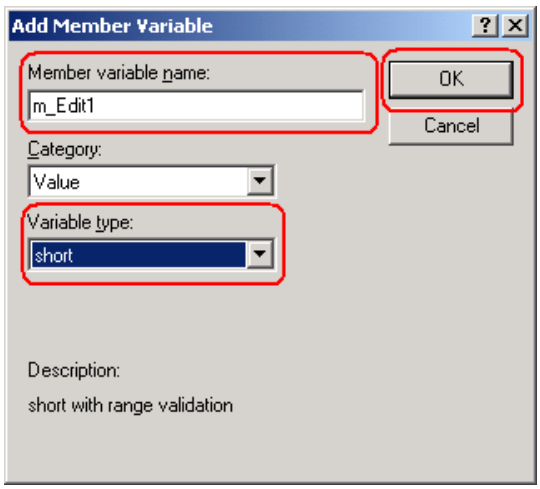
32 Select [ClassWizard] from [View] on the Microsoft Visual C++ menu.



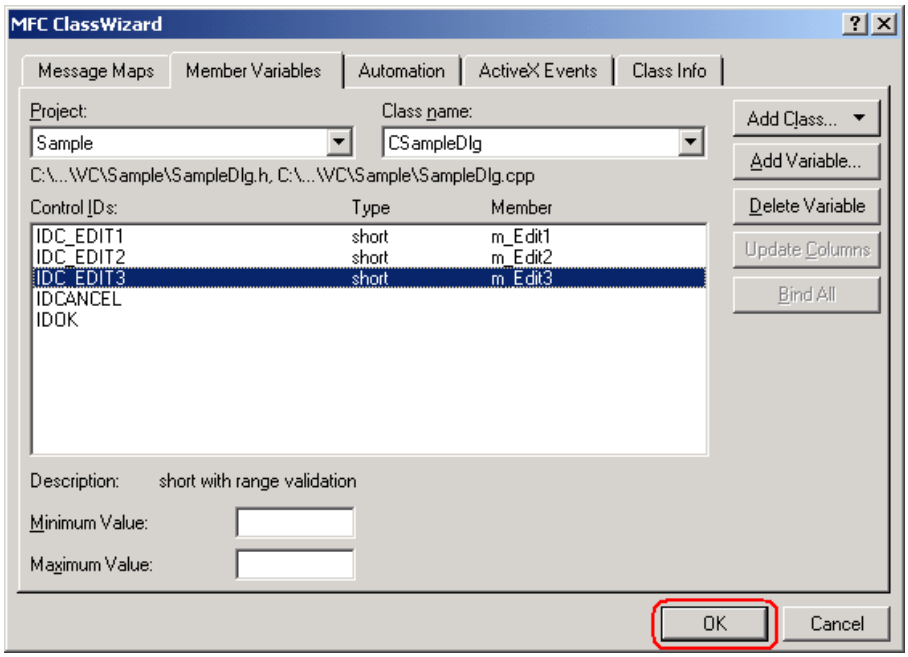
33 Select "IDC_EDIT1" for [Control IDs] in the [Member Variables] tab, and click the [Add Variable] button.



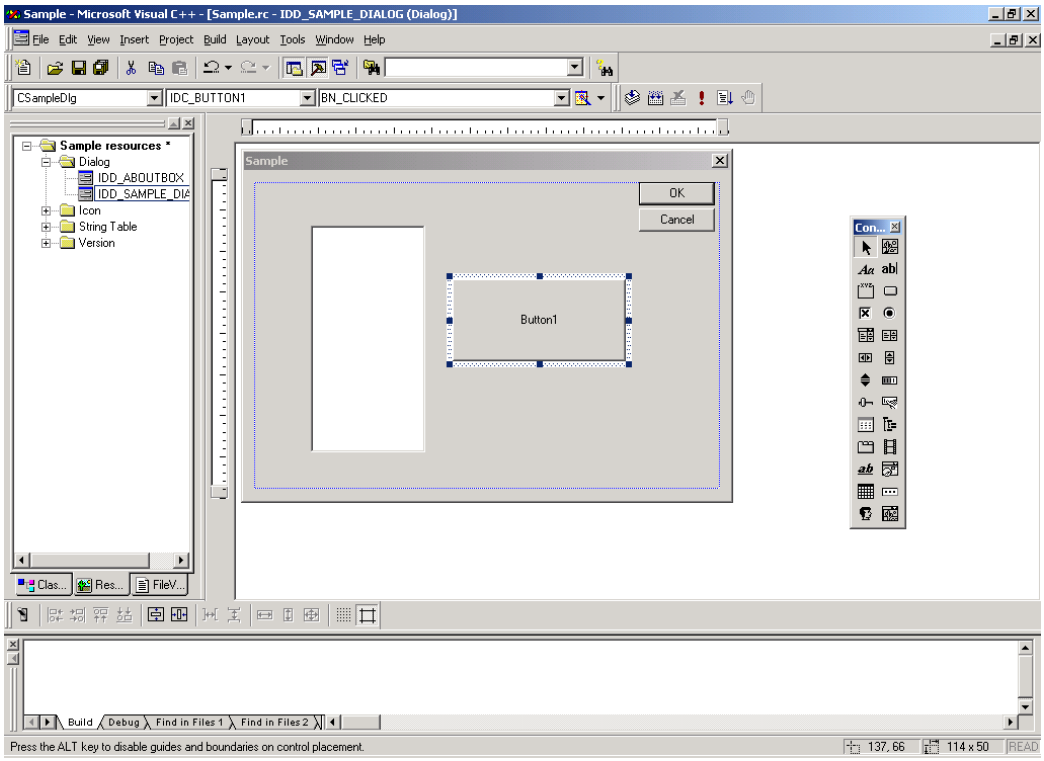
34 Enter "m_Edit1" for [Member Variable], and select "short" for [Variable type]. Then, click the [OK] button.
For remaining two [Edit Box], repeat steps 33 and 34. Specify "m_Edit2" and "m_Edit3" for member variables, respectively.



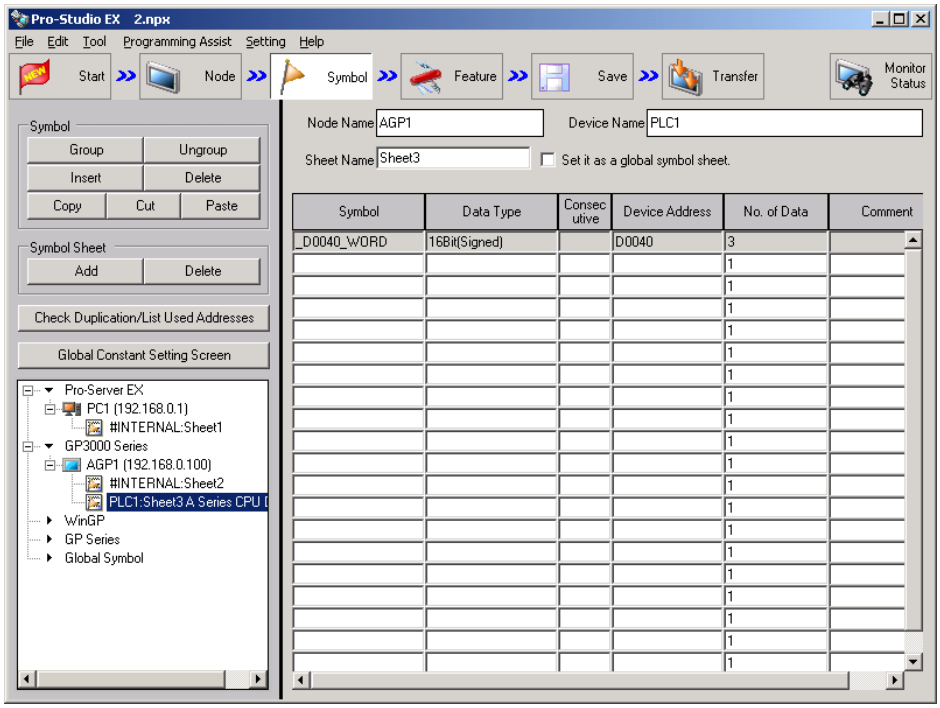
35 Click the [OK] button.



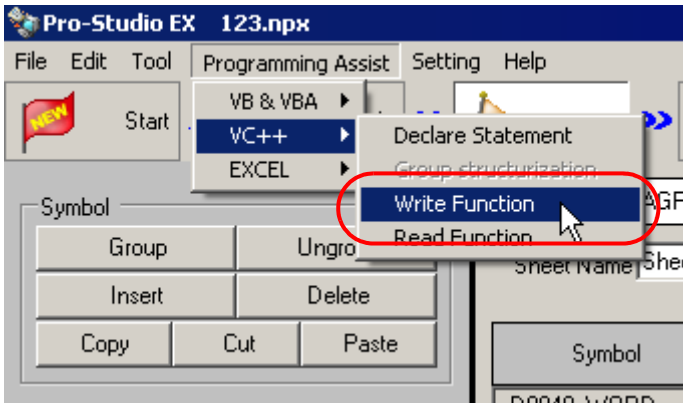
36 Select [Button], and paste it to [Dialog].



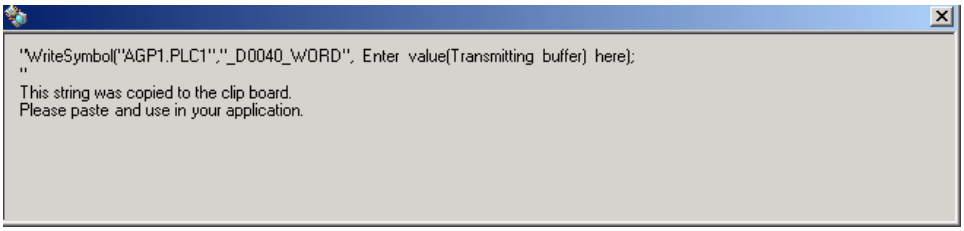
37 Select a target symbol name from those registered in 'Pro-Studio EX'. (Select the symbol with first-adress for writting.)



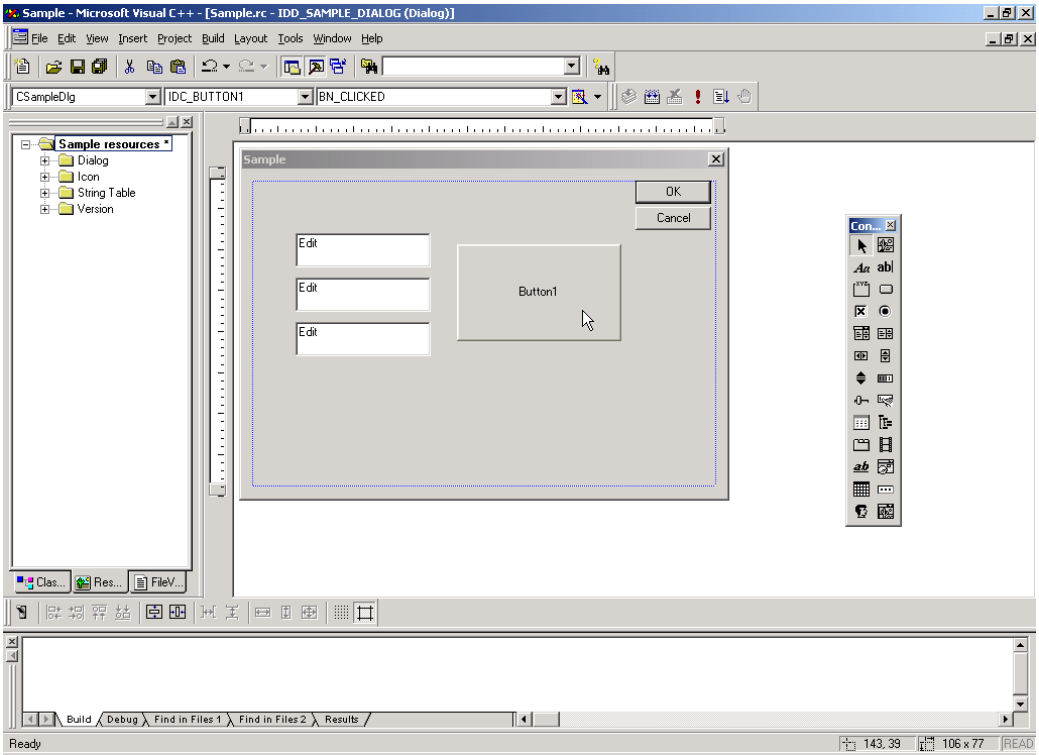
38 Select [Programming Assist] - [VC++] - [Write Function] on the menu.



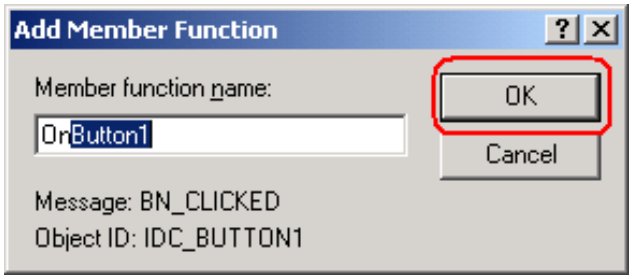
The write function is copied to the clipboard.



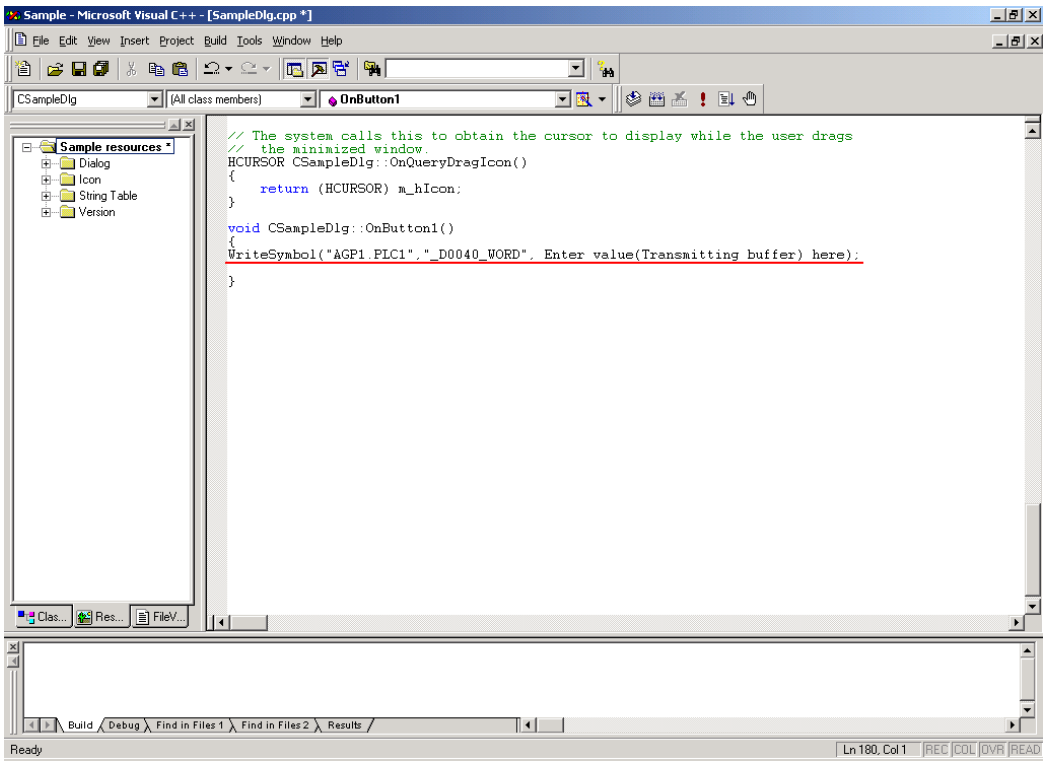
39 Double-click [Button1] that has been pasted to [Dialog] in Microsoft Visual C++.



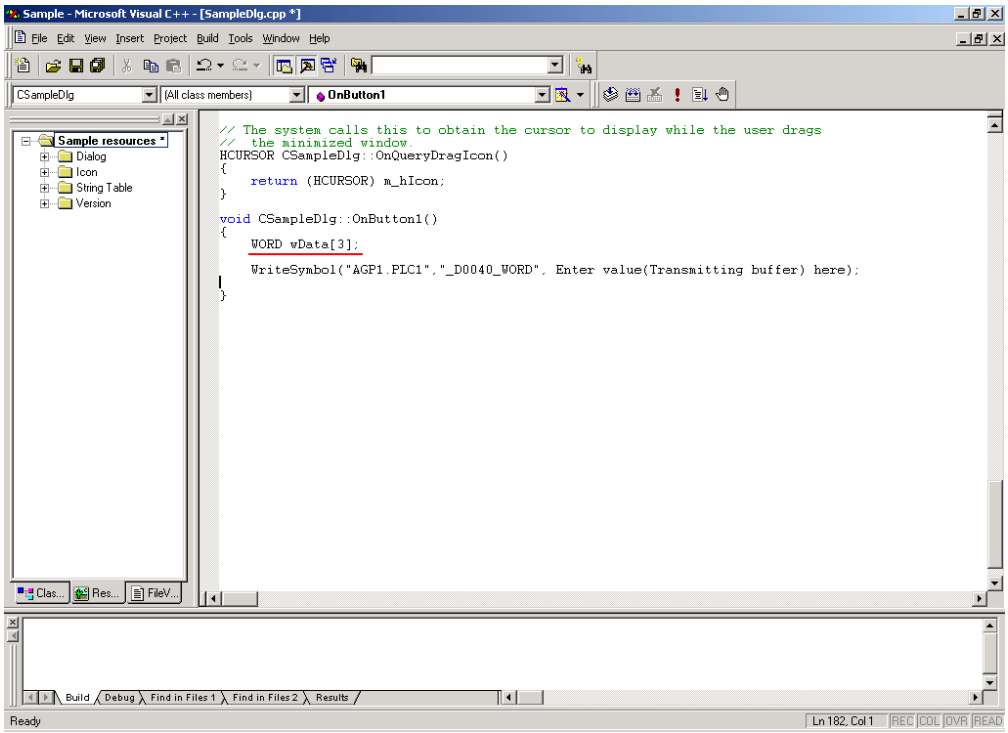
40 Click the [OK] button.



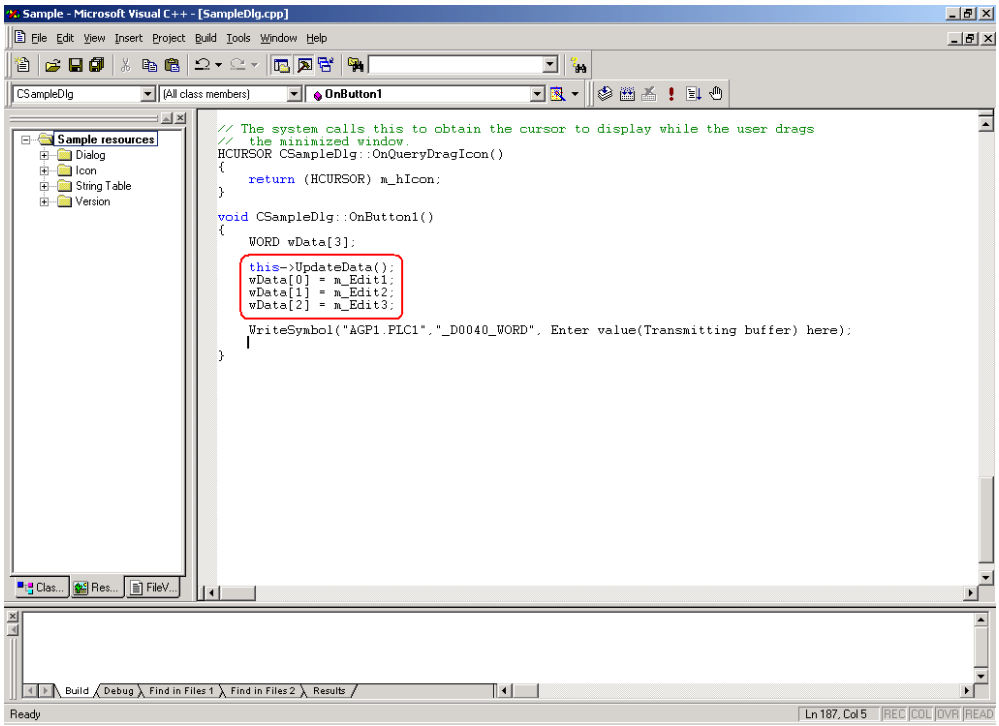
41 Paste the data on the clipboard (write function) into the OnButton1 member function.



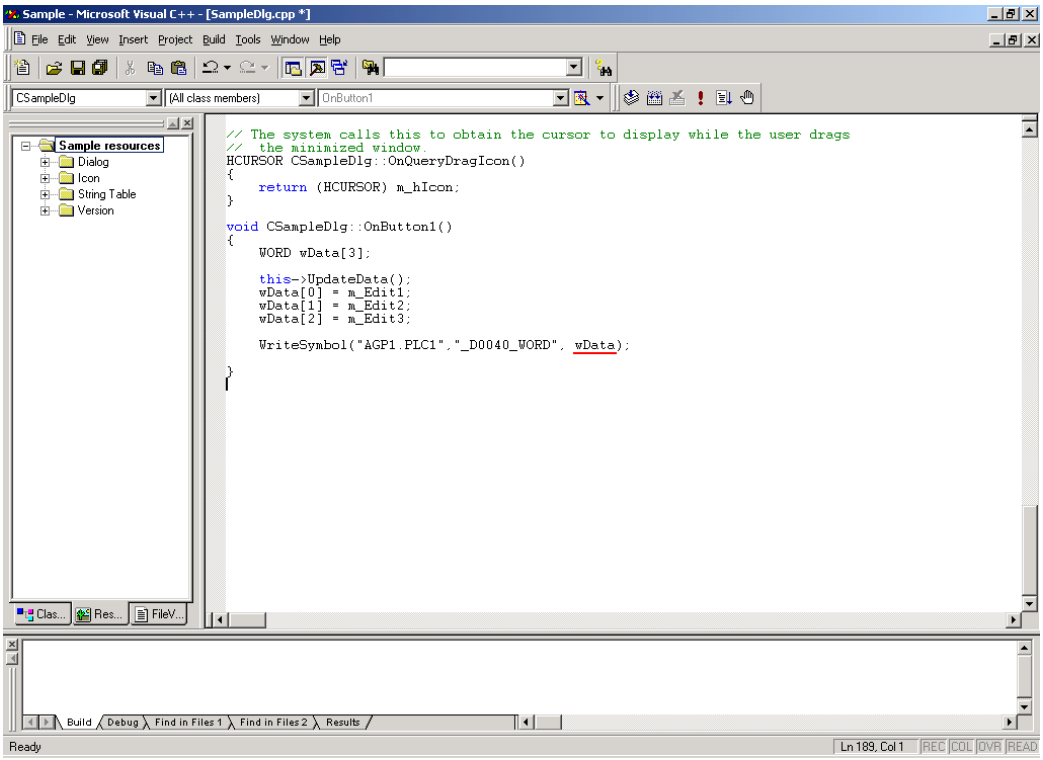
42 Declare the area (Array) to store the write data. For three or more writing points, specify three or more array elements.



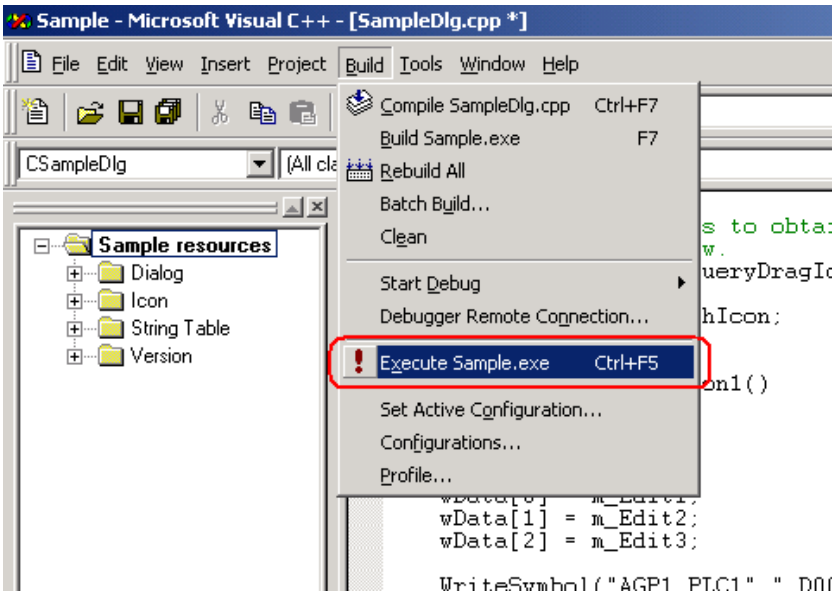
43 Set the data entered in [Edit Box] (for three points) into the array.



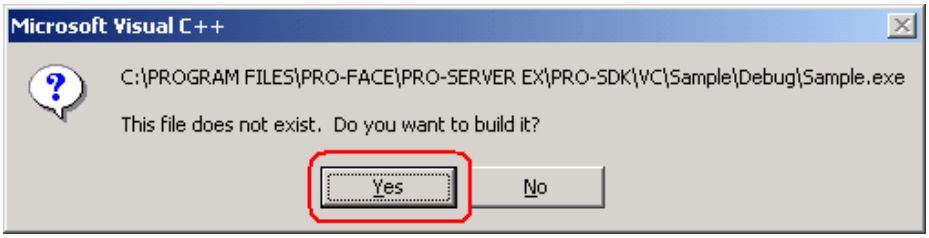
44 Specify the first alignment (wData) where the written data has been set.



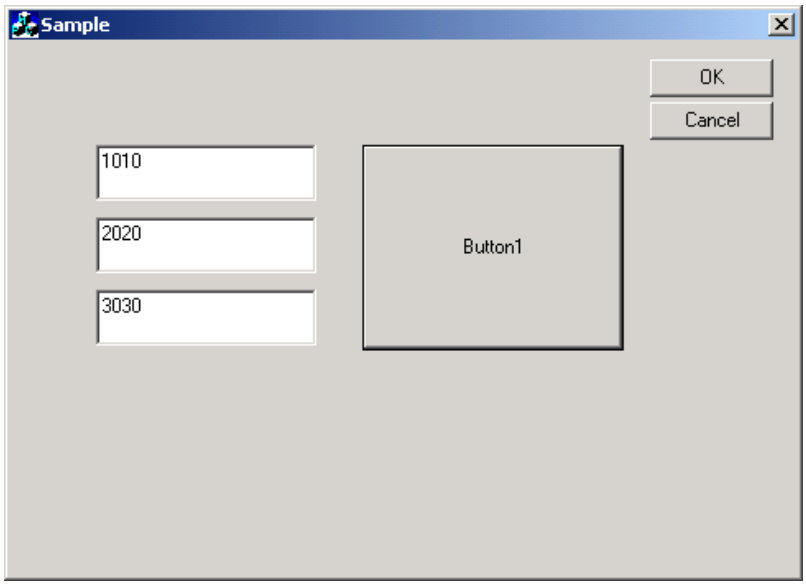
45 Select [Execute Sample.exe] from [Build] on the Microsoft Visual C++ menu.



46 Click the [Yes] button.

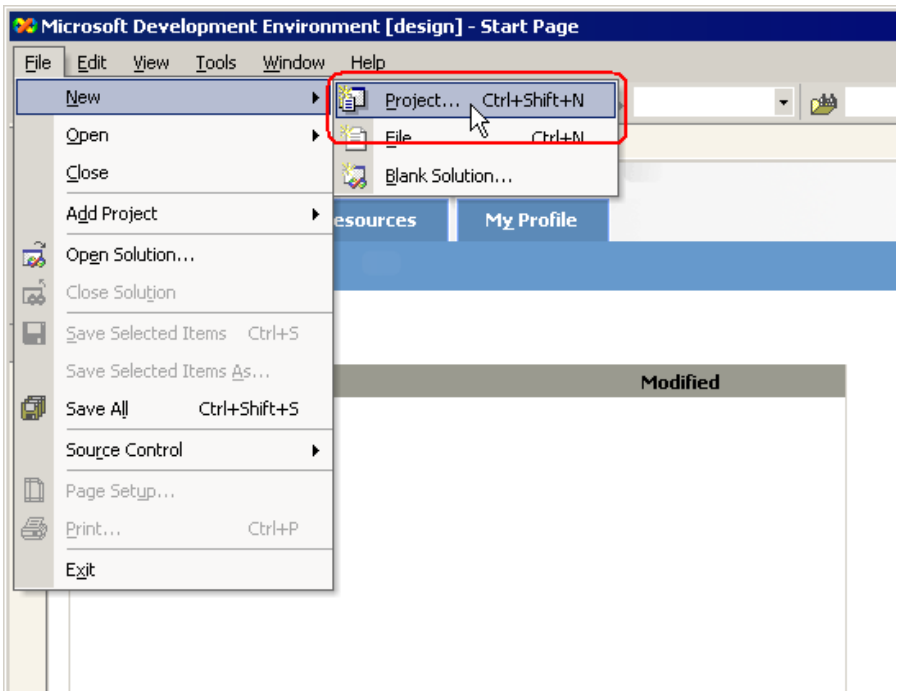


47 After entering the values for three points in each [Edit Box], click [Button1]. Then, 'Pro-Server EX' executes the writing of the data for three points from the symbol "_D0040_WORD".

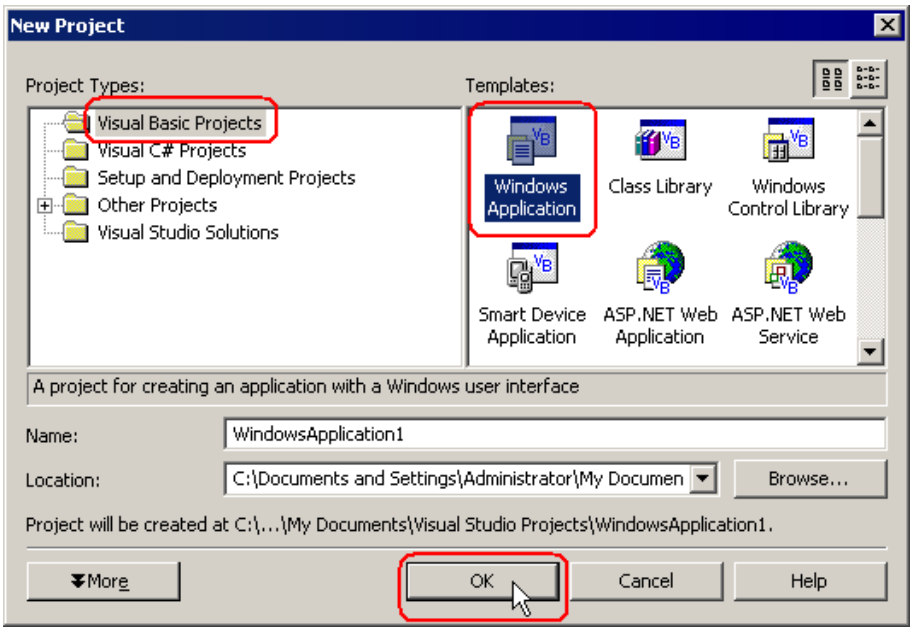


26.10.3 VB .NET Support Function

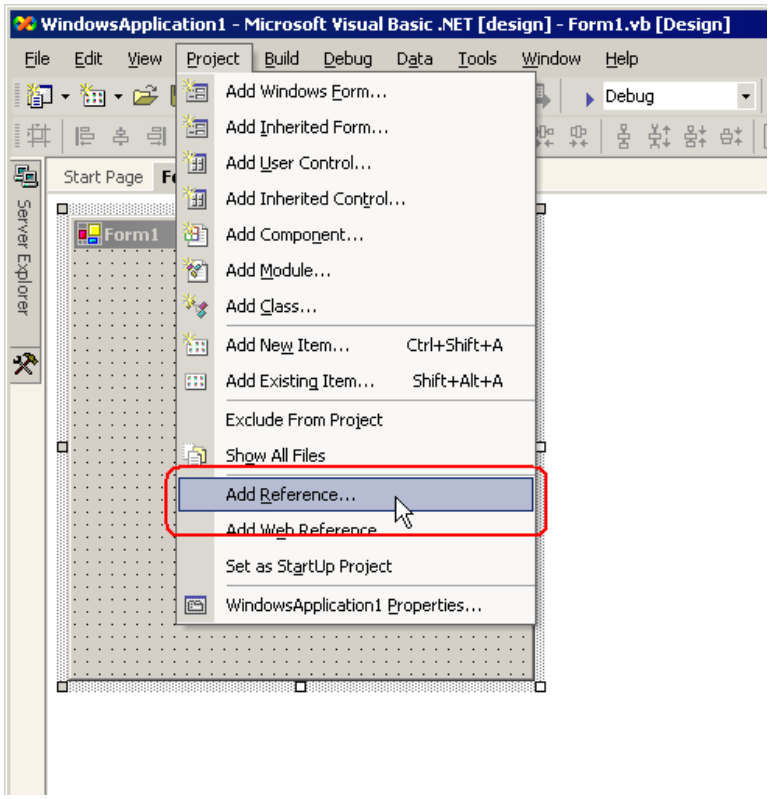
1 Start Microsoft Visual Studio .NET 2003 (or later version), and select [New] - [Project] from the [File] menu.



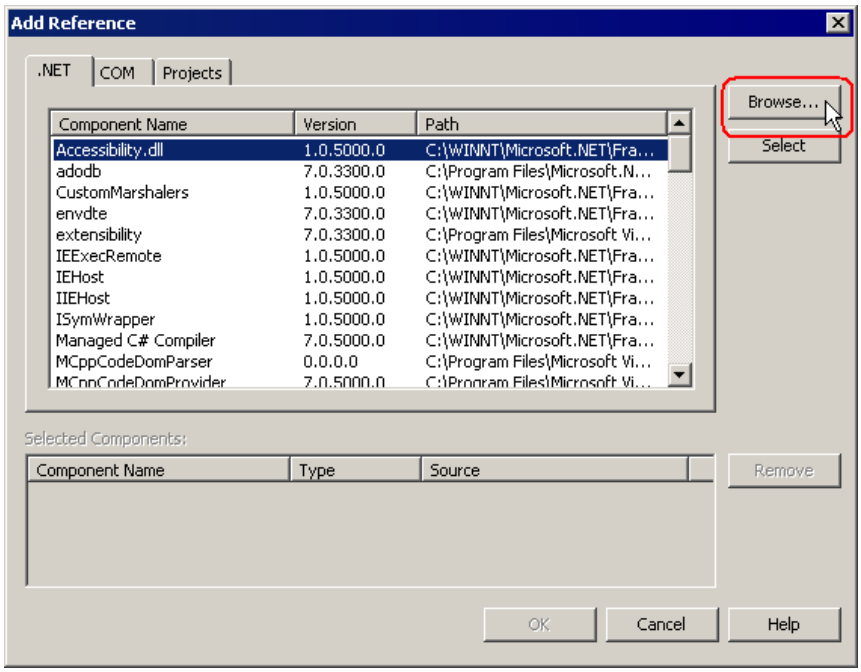
2 After selecting [Visual Basic Projects] in [Project Types:], select [Windows Application] in [Templates:], and click the [OK] button.



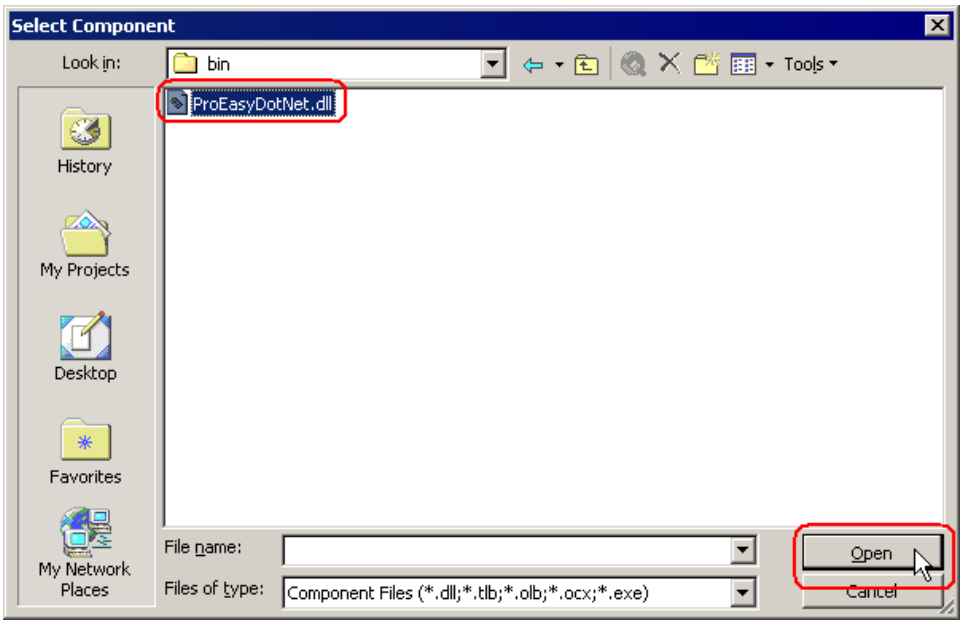
3 Select [Add Reference] from the [Project] menu.



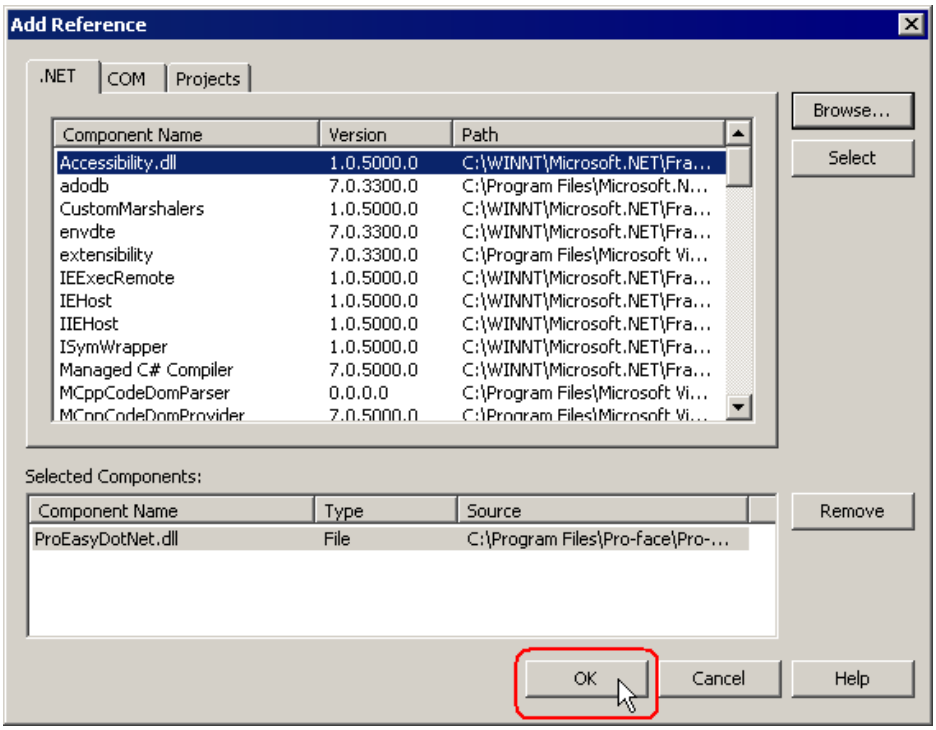
4 Click the [Browse] button.



5 Specify the directory for ProEazyDotNet.dll to be installed, and click the [Open] button. (When installed as standard, the directory is "C:\Program Files\Pro-face\Pro-Server EX\PRO-SDK\DotNet\bin\ProEazyDotNet.dll".)



6 Click the [OK] button.



"ProEasyDotNet.dll" will be registered.

This completes the VB.NET operating environment setup.

The above 1 to 6 steps apply to both reading and writing applications.

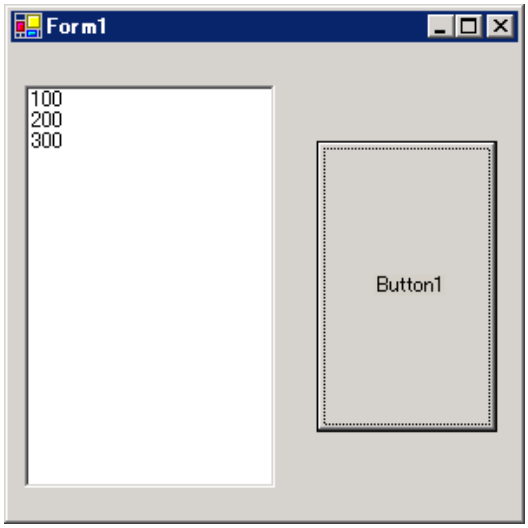
The following procedure varies depending on whether the application is intended for reading or writing, and so is explained individually.

To create a "Reading" application, refer to steps 7 to 19.

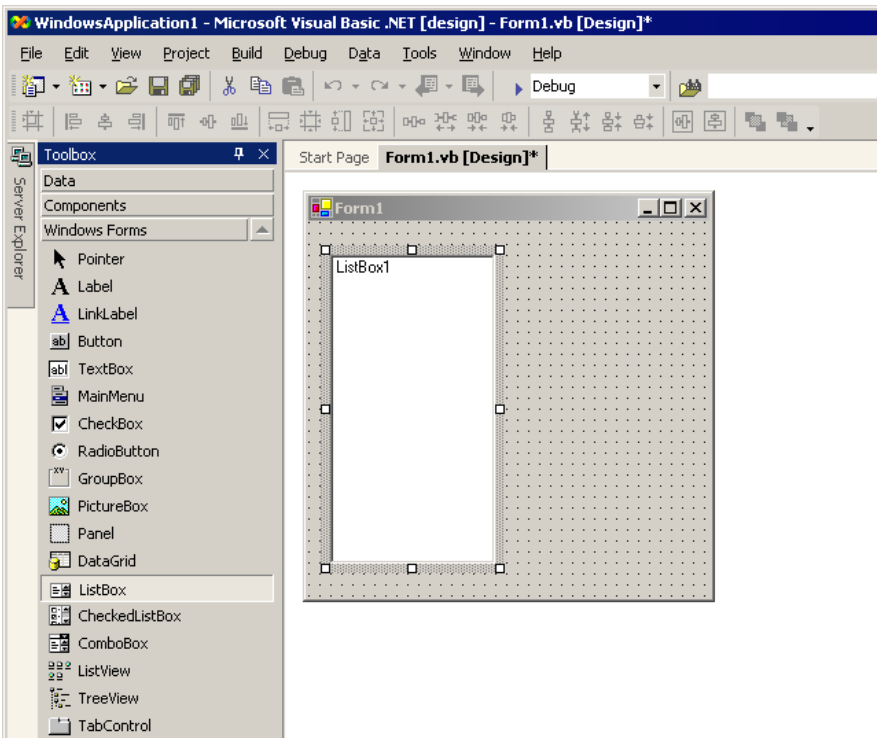
To create a "Writing" application, refer to steps 20 to 32.

Creating "Reading" application

This section describes the application that reads and displays data (signed 16 bits) on three items when you click [Button1].

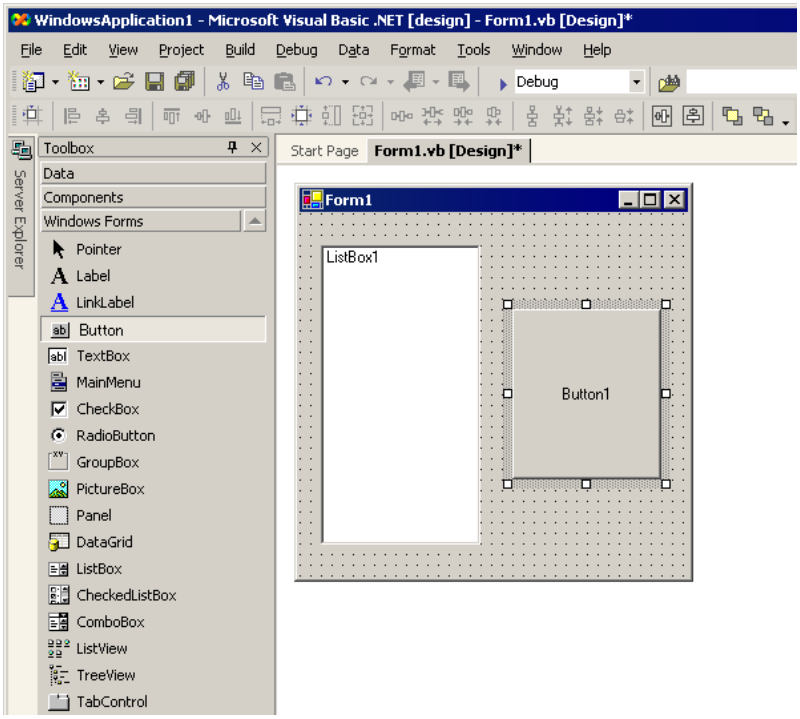


7 After selecting [ListBox] in [Toolbox], clip and paste it onto [Form1].

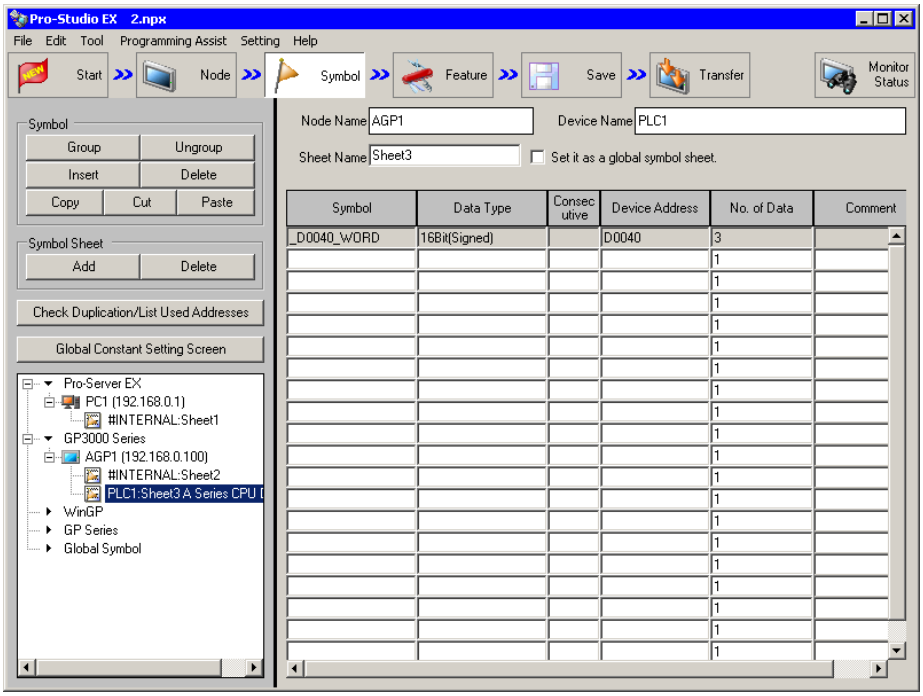


* If [Toolbox] is not displayed, select [Toolbox] from the [View] menu.

8 After selecting [Button] in [Toolbox], clip and paste it onto [Form1].

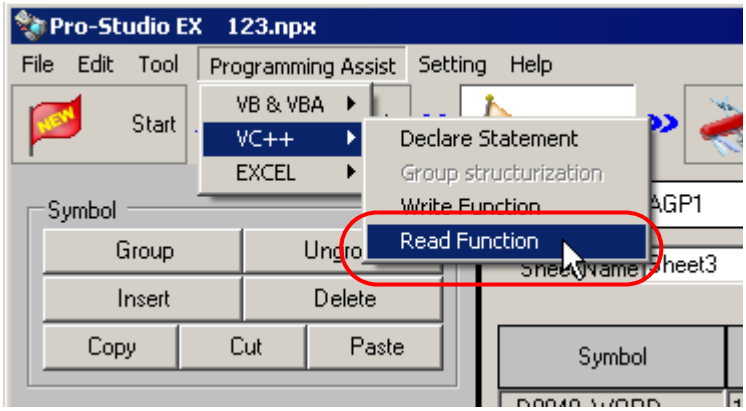


9 Select a desired read symbol name from the symbols that have been registered in 'Pro-Studio EX'.

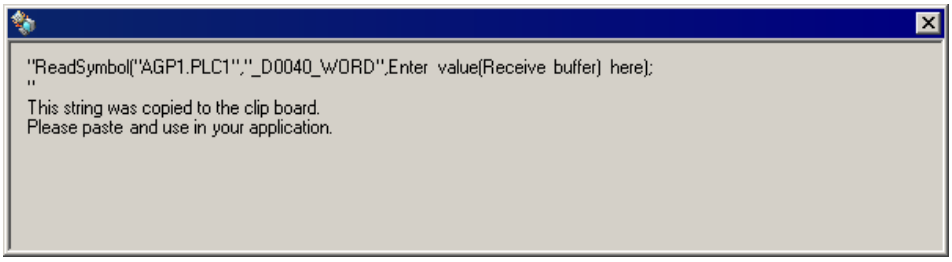


* The above example shows the symbol for the data type of [16Bit (Signed)] and the data quantity of "3".

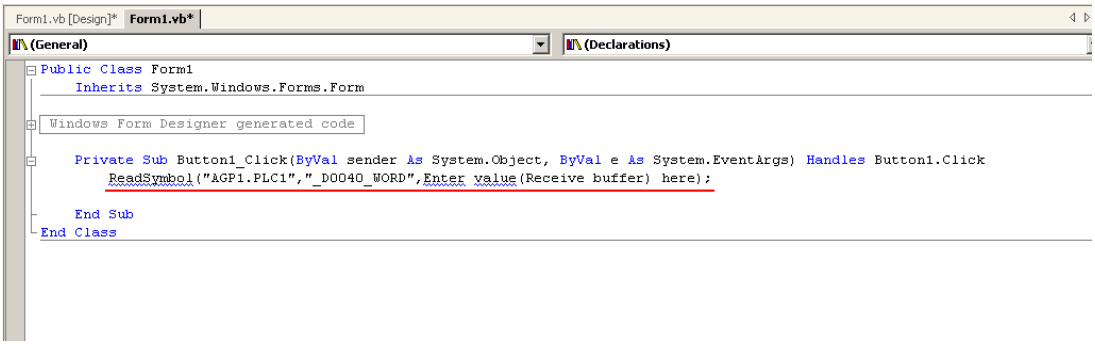
10 Select [VC++] - [Read Function] from the [Programming Assist] menu.



The read function is copied to the clipboard.

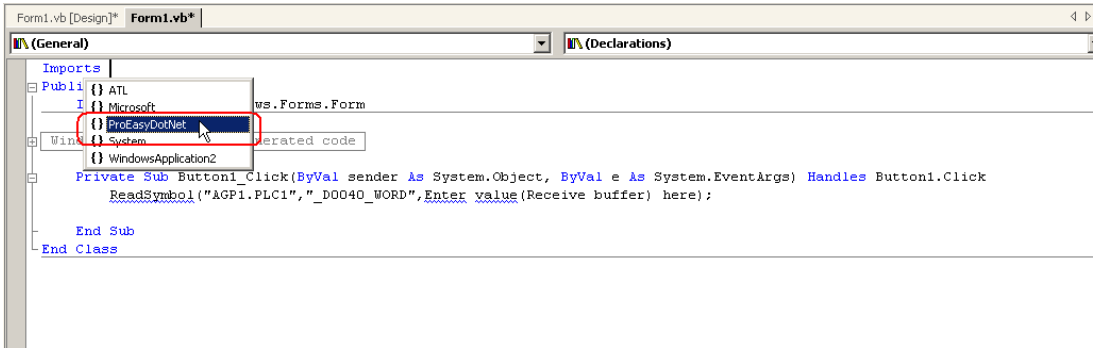


11 Double-click [Button1] in [Form1], and paste the clipboard data (read function) between the Sub statement and the End Sub statement.



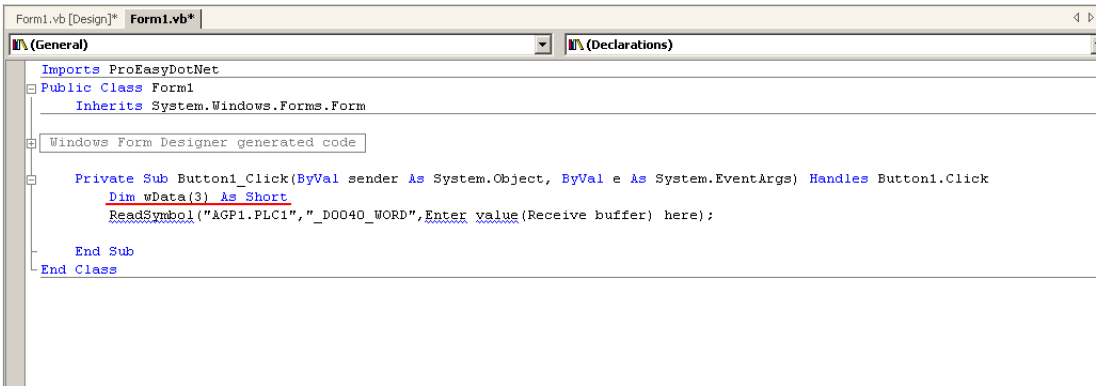
12 Import the ProEasyDotNet library.

Enter "Imports" at the head of the source code, and select [ProEasyDotNet] from the displayed list box.

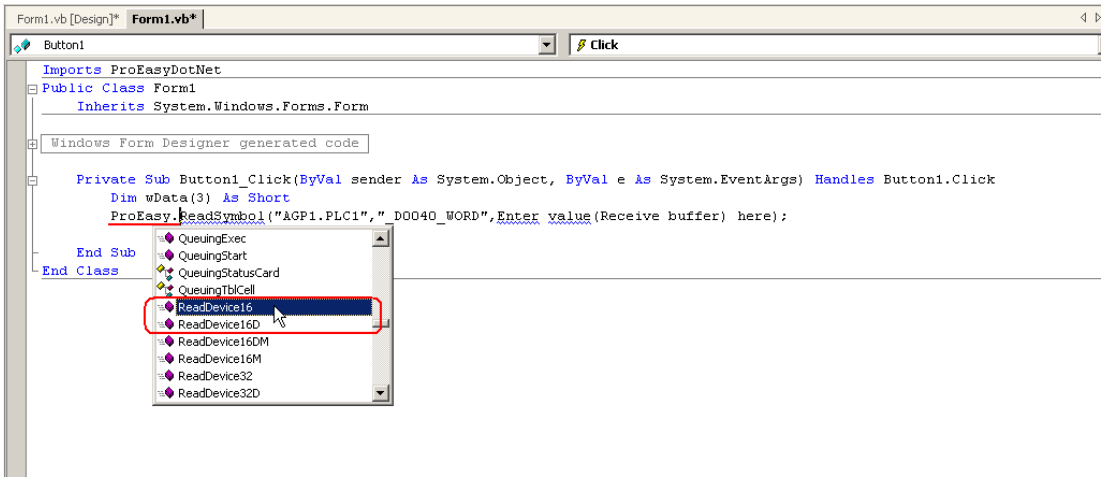


13 For the read data storing area, declare a variable "wData".

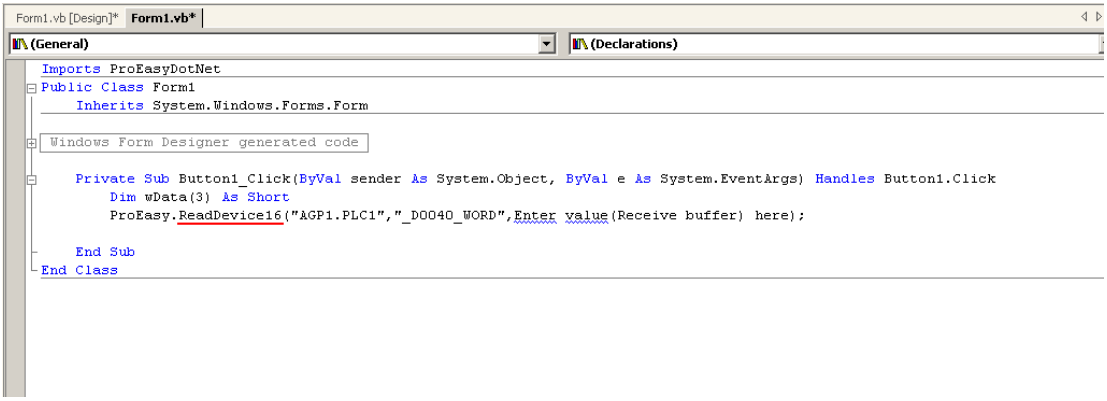
The array type ("Short" in this example) must conform to the data type of the target symbol. Specify the same data length as the target symbol ("3" in this example).



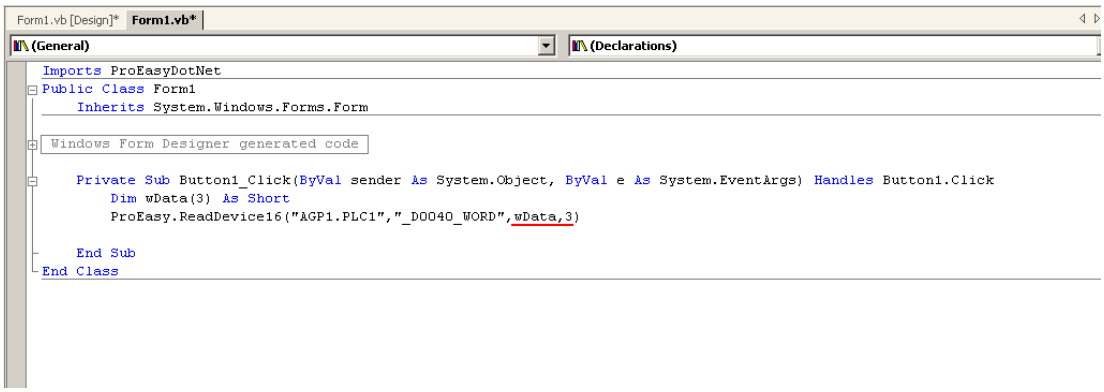
14 Enter "ProEasy." before "ReadSymbol", and select [ReadDevice16] from the displayed list box.



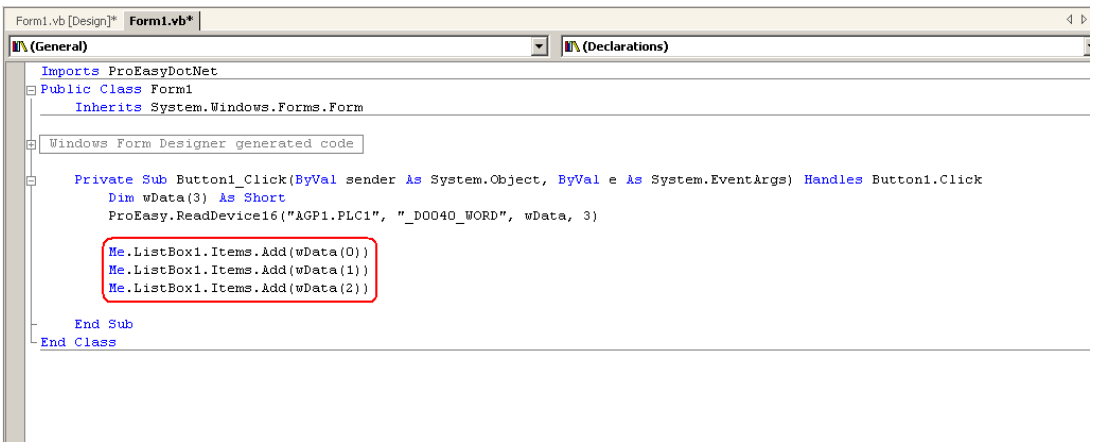
15 Delete "ReadSymbol" from the character string (read function) that has been pasted from the clipboard.



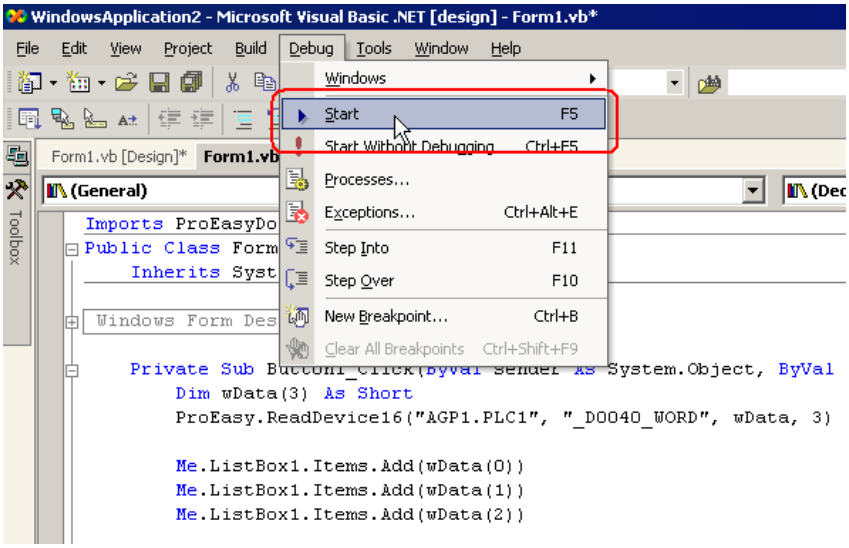
16 Specify a data storing area "wData" as the third argument. Enter ", " (comma) at the end of the third argument, and then enter "3" to specify the length of the target symbol as the fourth argument. After that, delete ";" (semicolon) at the end of the line.



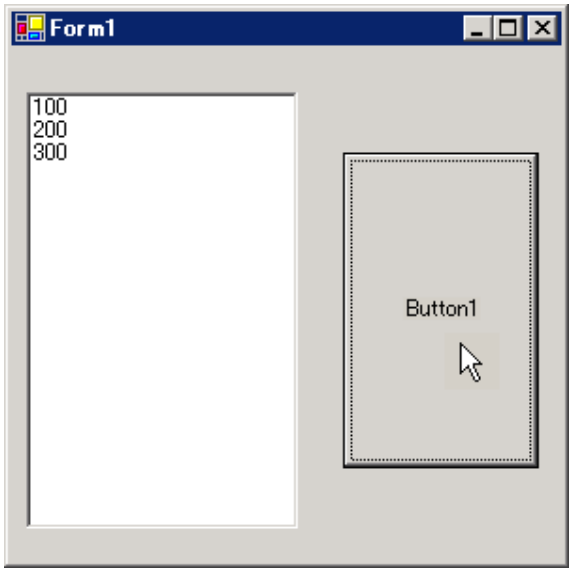
17 Add the read data on three items (wData(0), wData(1), wData(2)) into [ListBox1] in this order.



18 Select [Start] from the [Debug] menu.

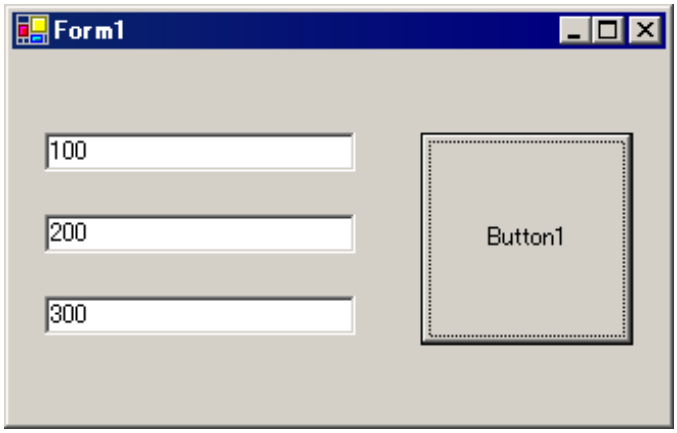


19 If you click [Button1], the target symbol data (three items) are displayed in [ListBox].

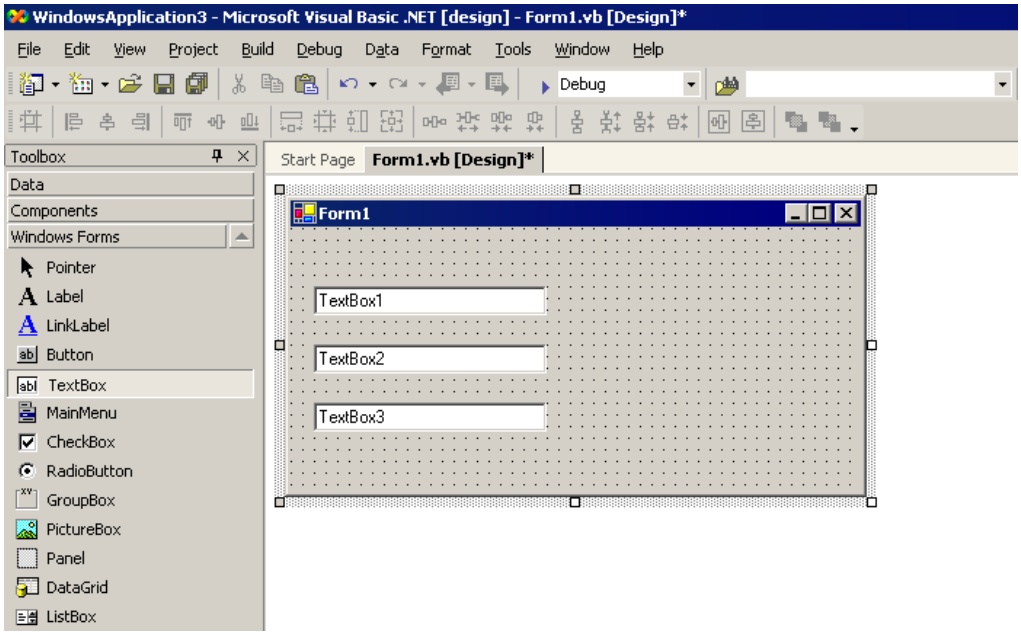


Creating "Writing" application

This section describes the application that writes data (signed 16 bits) on three items when you click [Button1].

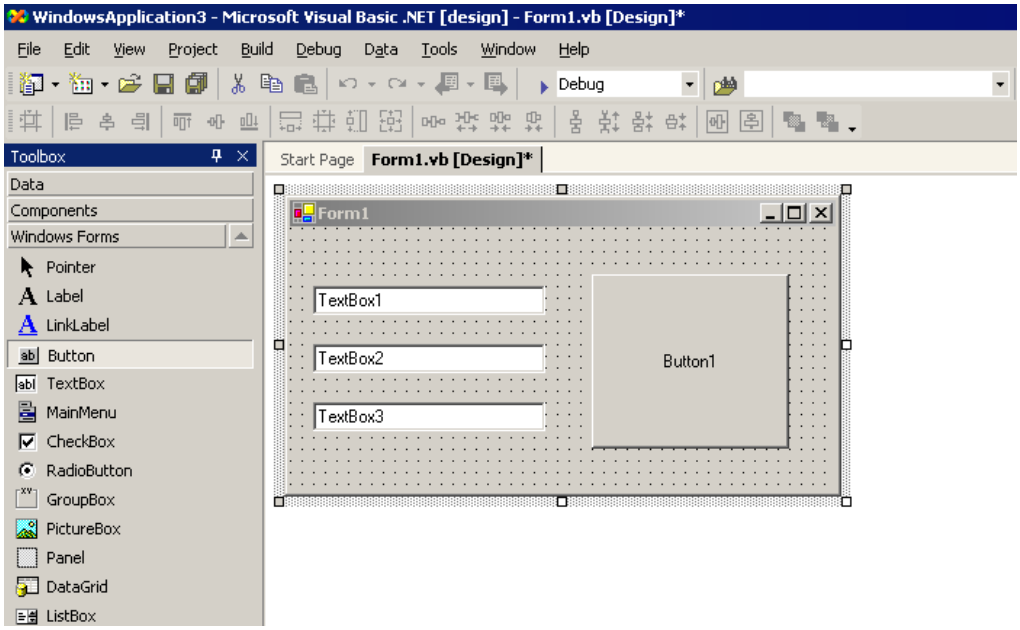


20 After selecting [TextBox] in [Toolbox], clip and paste three text boxes onto [Form1].

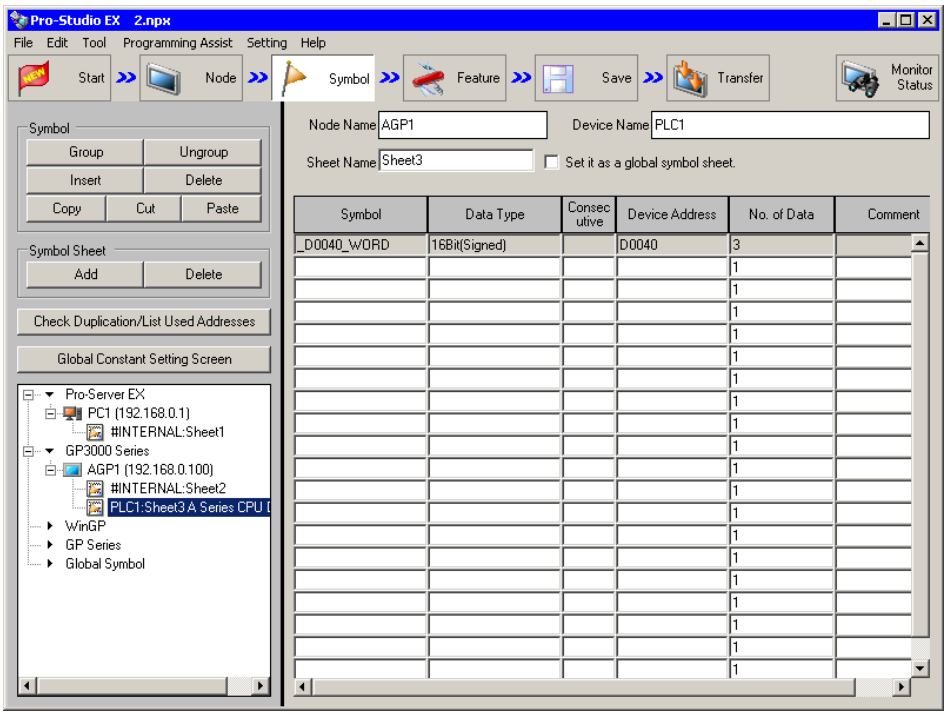


* If [Toolbox] is not displayed, select [Toolbox] from the [View] menu.

21 After selecting [Button] in [Toolbox], clip and paste it onto [Form1].

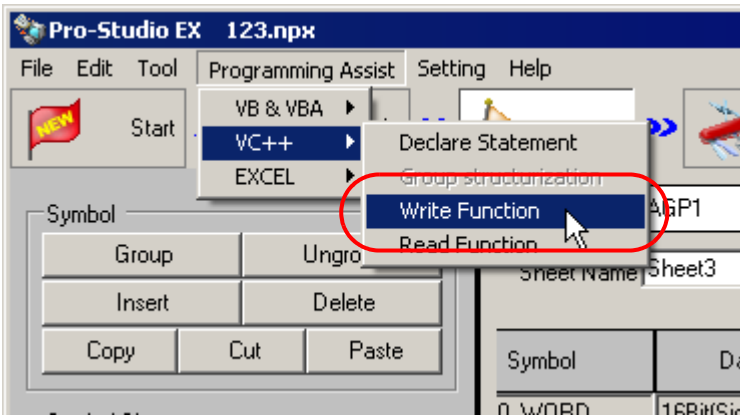


22 Select a desired write symbol name from the symbols that have been registered in 'Pro-Studio EX'. (Select the first writing area.)

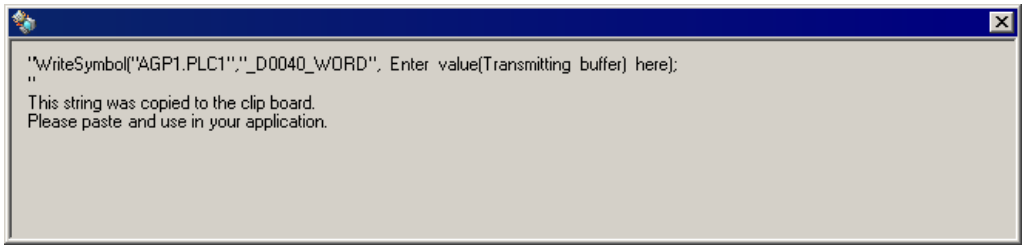


* The above example shows the symbol for the data type of [16Bit (Signed)] and the data quantity of "3".

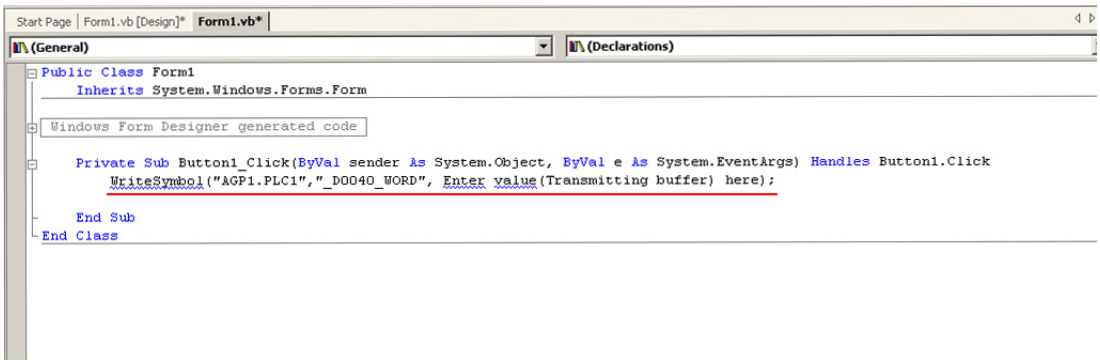
23 Select [VC++] - [Write Function] from the [Programming Assist] menu.



The write function is copied to the clipboard.

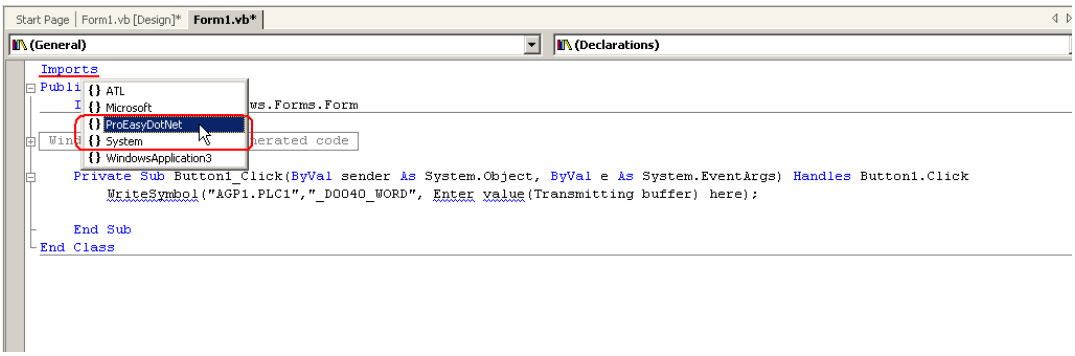


24 Double-click [Button1] in [Form1], and paste the clipboard data (write function) below the [Button1_Click] method ("Private Sub Button1_Click..." character string).



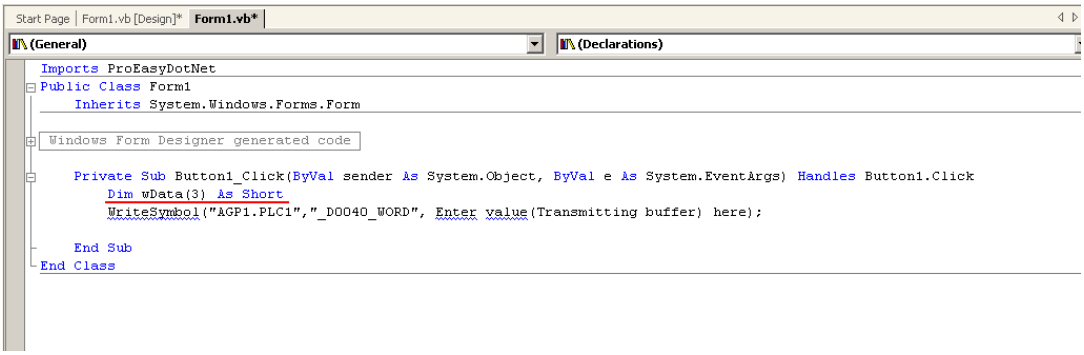
25 Import the ProEasyDotNet library.

Enter "Imports" at the head of the source code, and select [ProEasyDotNet] from the displayed list box.

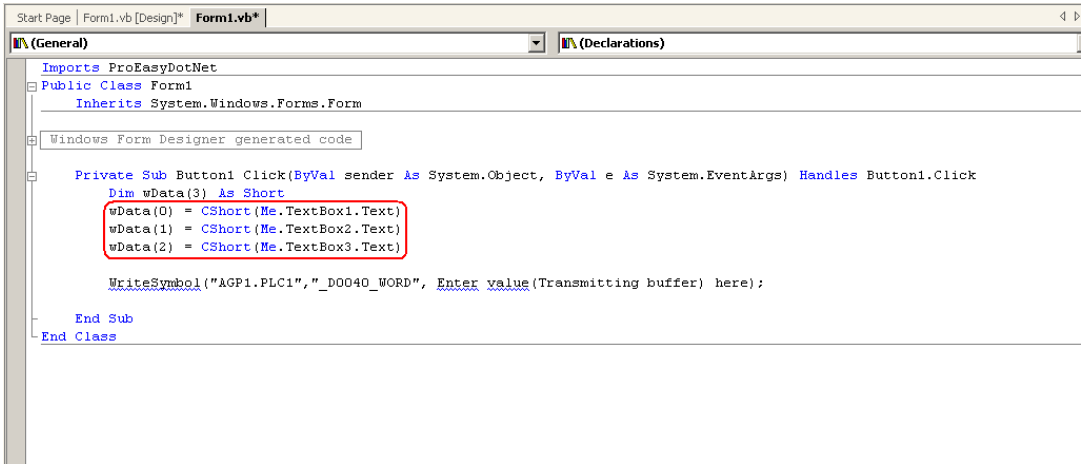


26 For the write data storing area, declare a variable "wData".

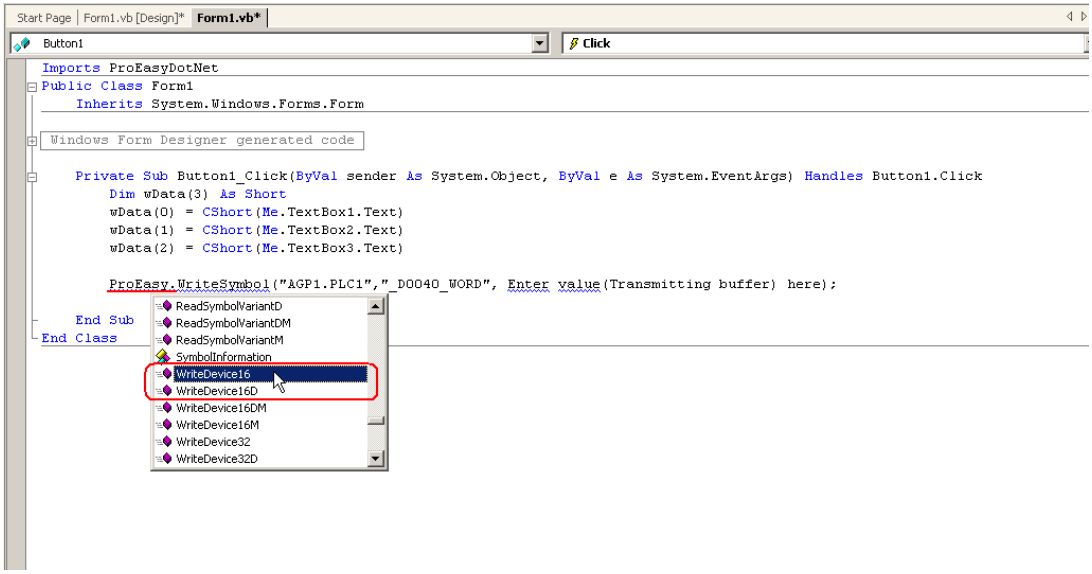
The array type ("Short" in this example) must conform to the data type of the target symbol. Specify the same data length as the target symbol ("3" in this example).



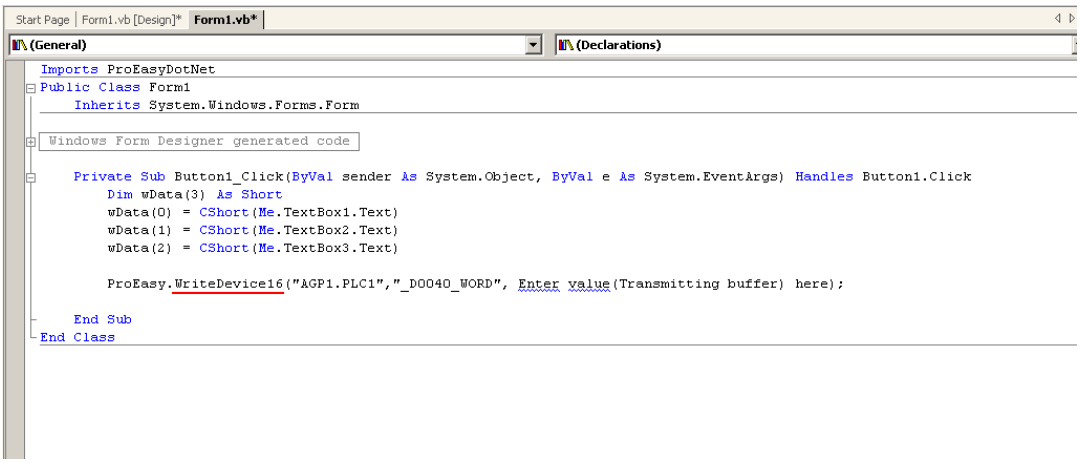
27 Set the data to be entered in [TextBox1] to [TextBox3] in the array.



28 Enter "ProEasy." before "WriteSymbol", and select [WriteDevice16] from the displayed list box.



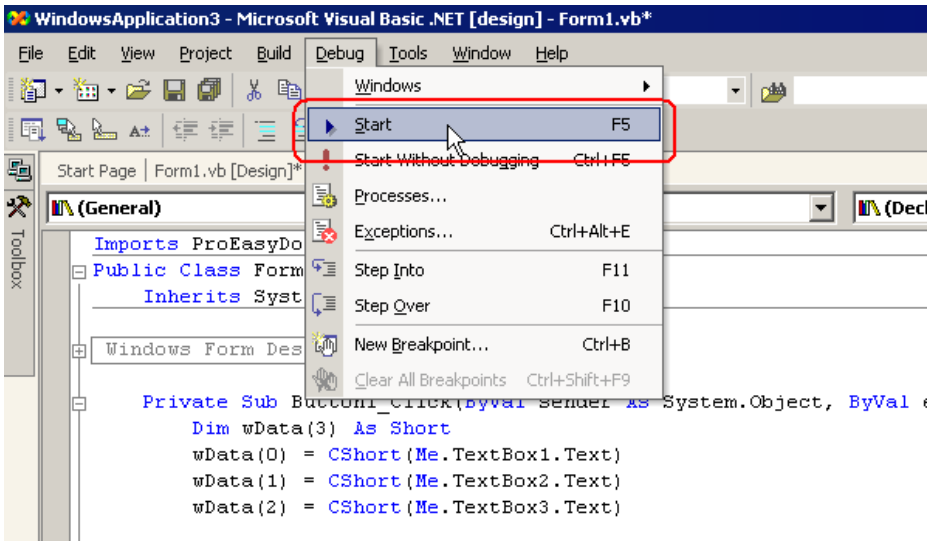
29 Delete "WriteSymbol" from the character string (write function) that has been pasted from the clipboard.



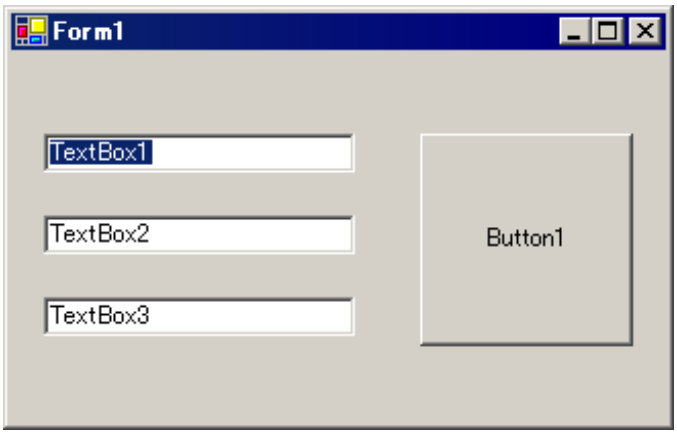
30 Specify a data storing area "wData" as the third argument. Enter "," (comma) at the end of the third argument, and then enter "3" to specify the length of the target symbol as the fourth argument. After that, delete ";" (semicolon) at the end of the line.

```
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", wData, 3)

    End Sub
End Class
```

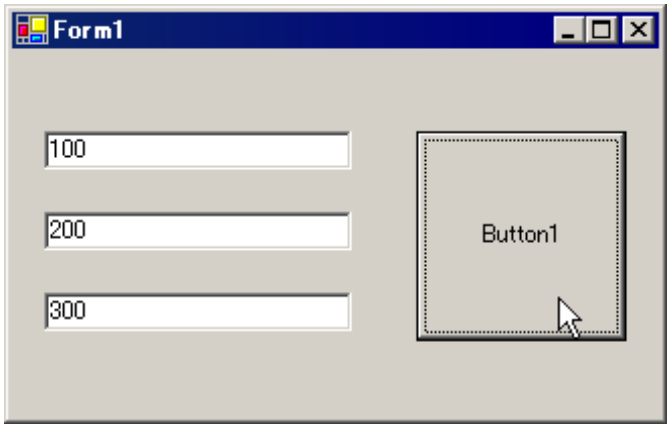
31 Select [Start] from the [Debug] menu.



32 Immediately after startup, a character string "TextBox*" is displayed in [TextBox].

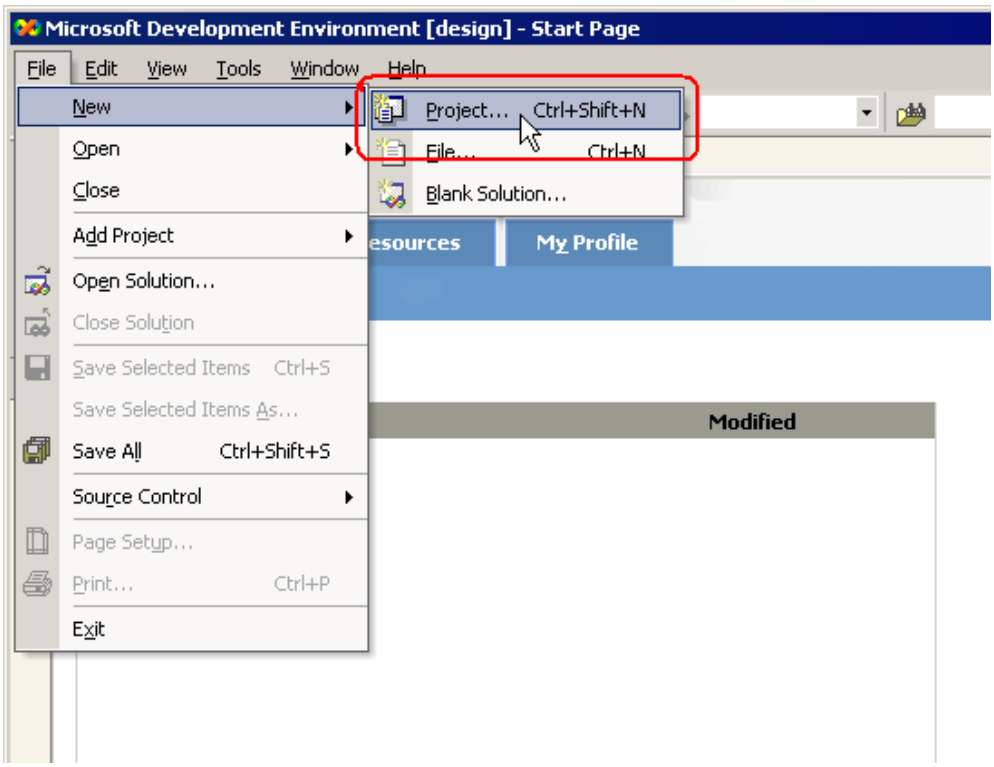


After entering the write data (three items) in [TextBox], click [Button1]. Then, the data will be written into the area specified with the symbol.

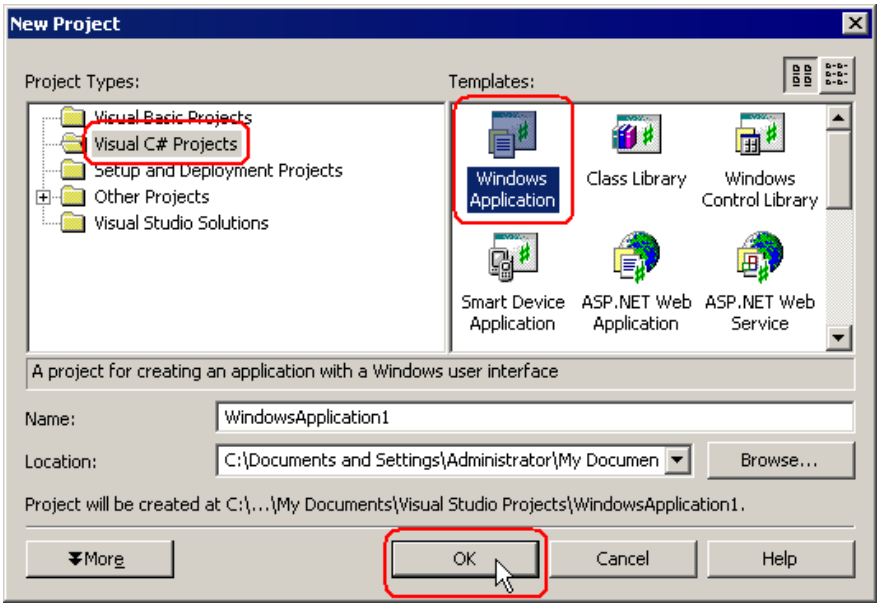


26.10.4 C# .NET Support Function

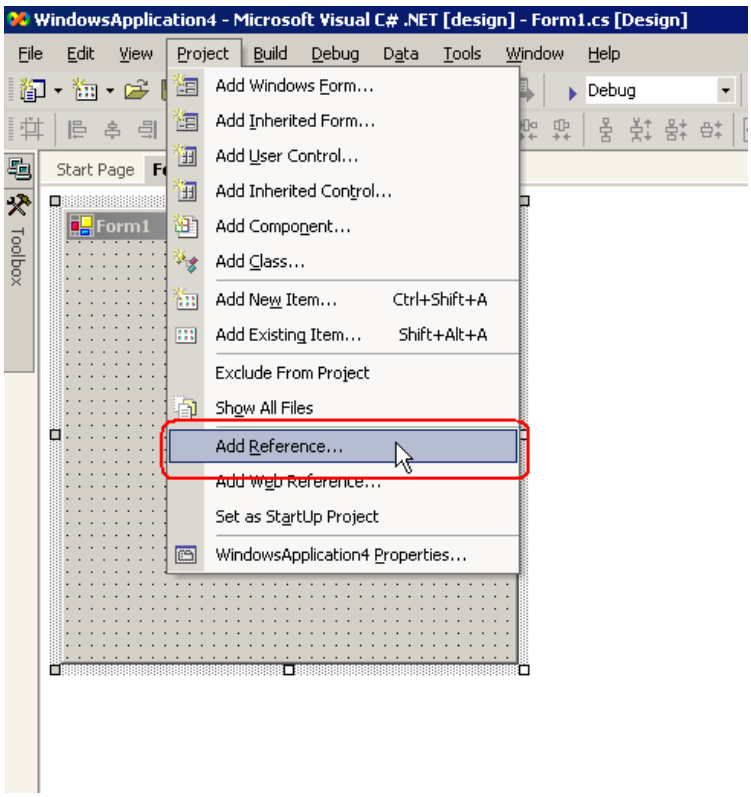
1 Start Microsoft Visual Studio .NET 2003 (or later version), and select [New] - [Project] from the [File] menu.



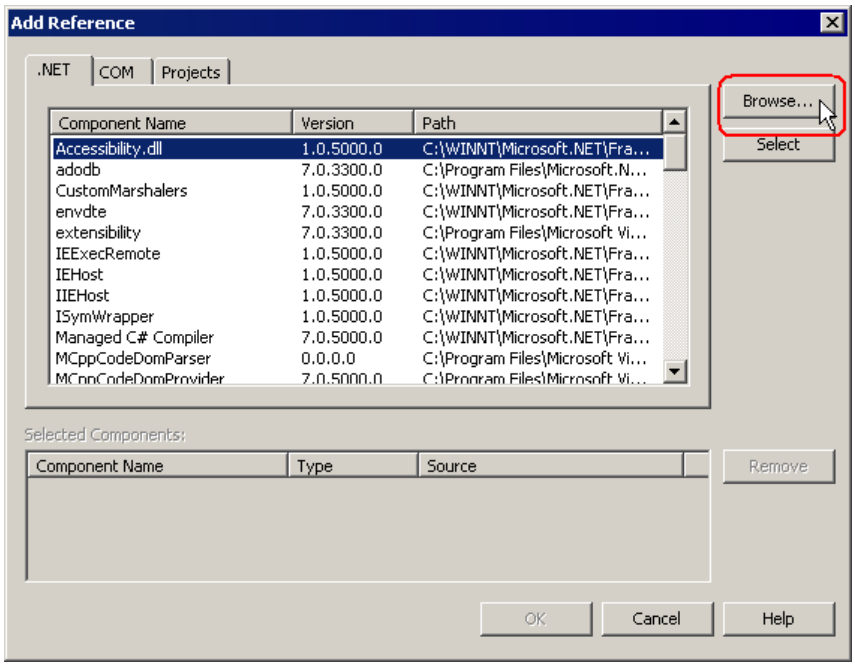
2 After selecting [Visual C# Projects] in [Project Types:], select [Windows Application] in [Templates:], and click the [OK] button.



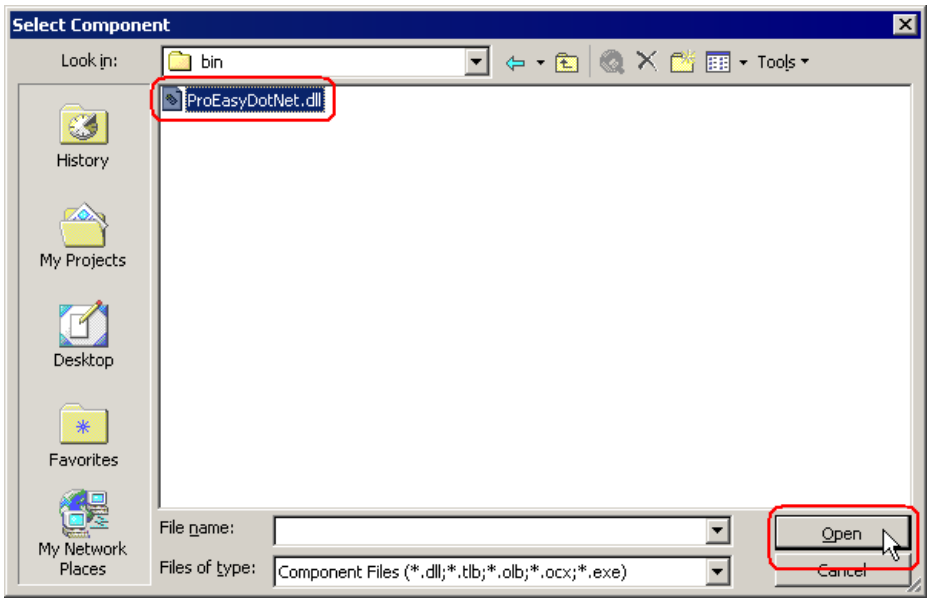
3 Select [Add Reference] from the [Project] menu.



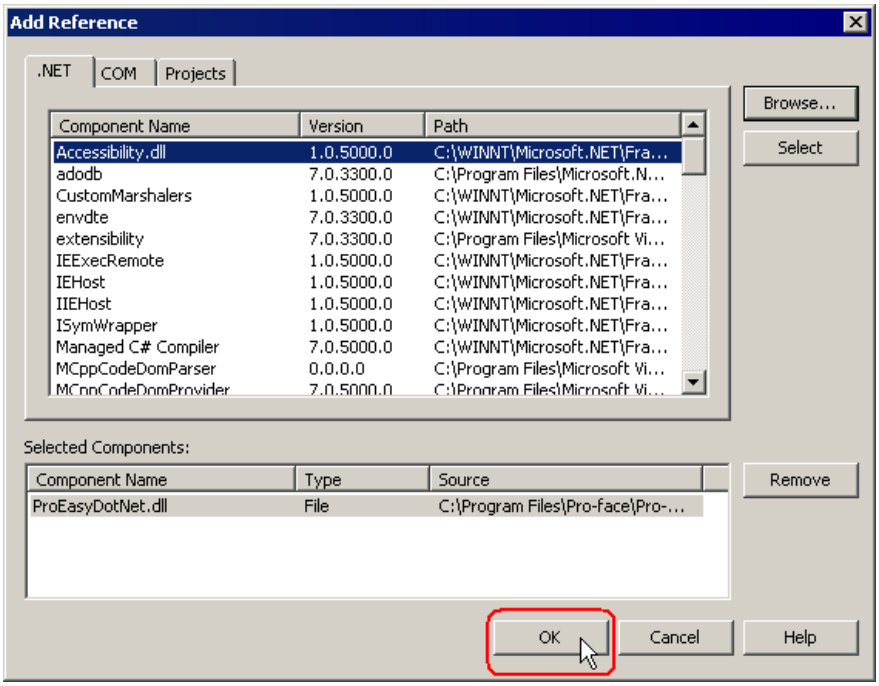
4 Click the [Browse] button.



5 Specify the directory for ProEazyDotNet.dll to be installed, and click the [Open] button. (When installed as standard, the directory is "C:\Program Files\Pro-face\Pro-Server EX\PRO-SDK\DotNet\bin\ProEazyDotNet.dll".)



6 Click the [OK] button.



"ProEasyDotNet.dll" will be registered.

This completes the C# .NET operating environment setup.

The above 1 to 6 steps apply to both reading and writing applications.

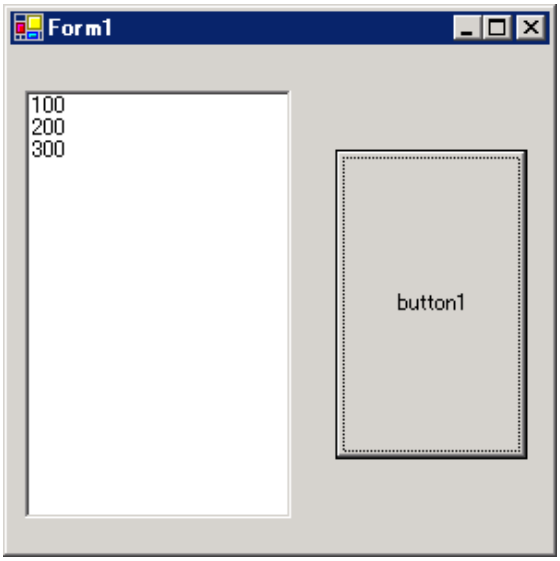
The following procedure varies depending on whether the application is intended for reading or writing, and so is explained individually.

To create a "Reading" application, refer to steps 7 to 19.

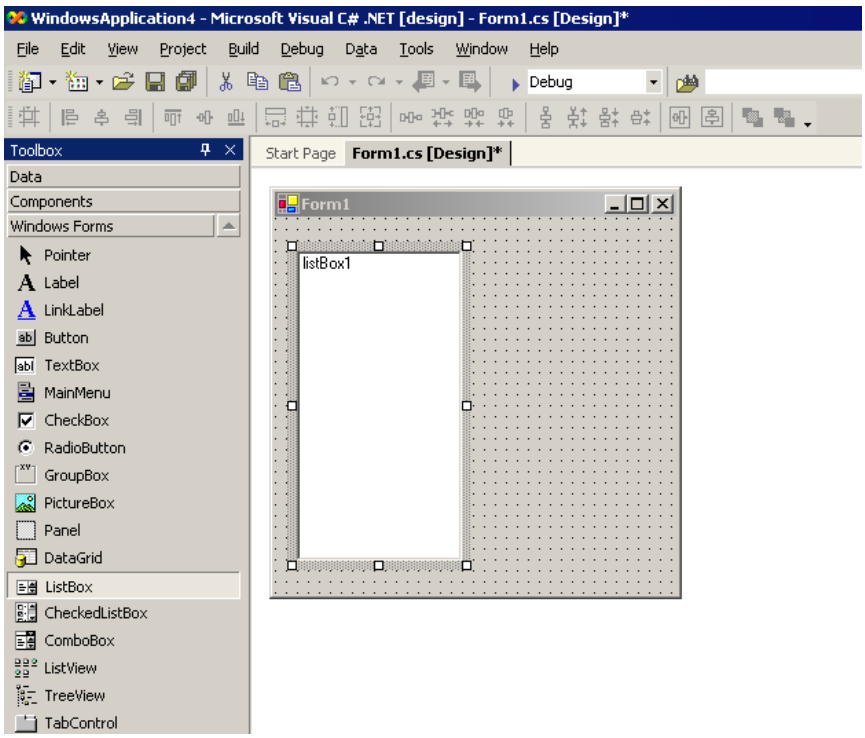
To create a "Writing" application, refer to steps 20 to 32.

Creating "Reading" application

This section describes the application that reads and displays data (signed 16 bits) on three items when you click [button1].

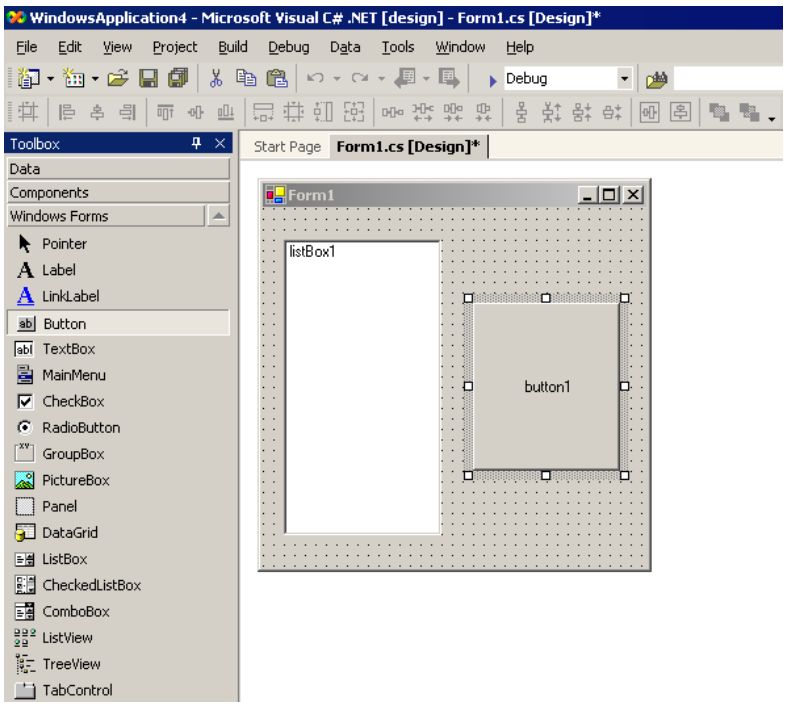


7 After selecting [ListBox] in [Toolbox], clip and paste it onto [Form1].

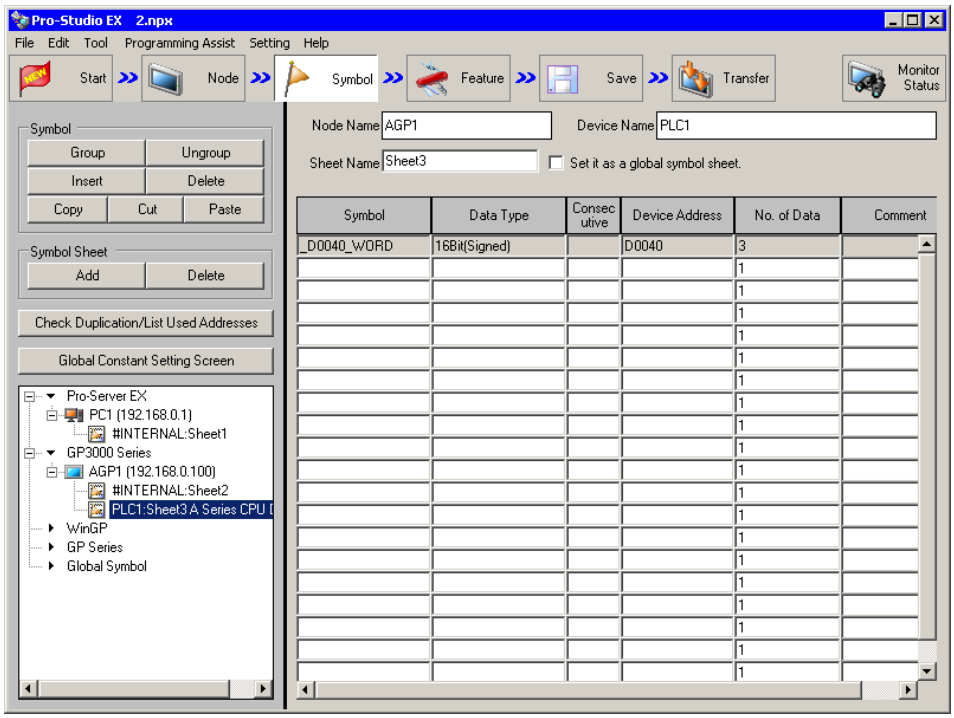


* If [Toolbox] is not displayed, select [Toolbox] from the [View] menu.

8 After selecting [Button] in [Toolbox], clip and paste it onto [Form1].

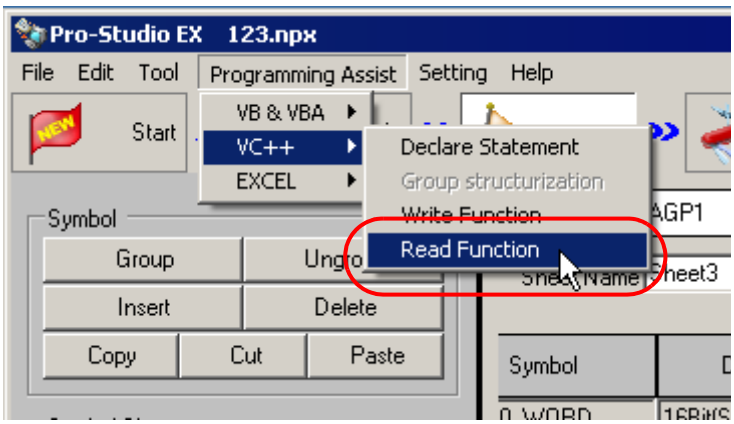


9 Select a desired read symbol name from the symbols that have been registered in 'Pro-Studio EX'.

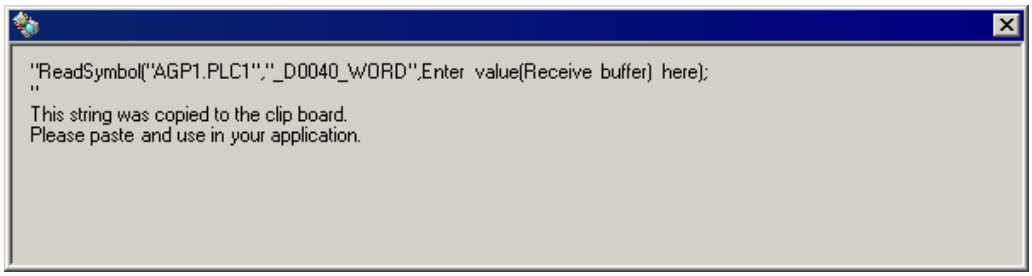


* The above example shows the symbol for the data type of [16Bit (Signed)] and the data quantity of "3".

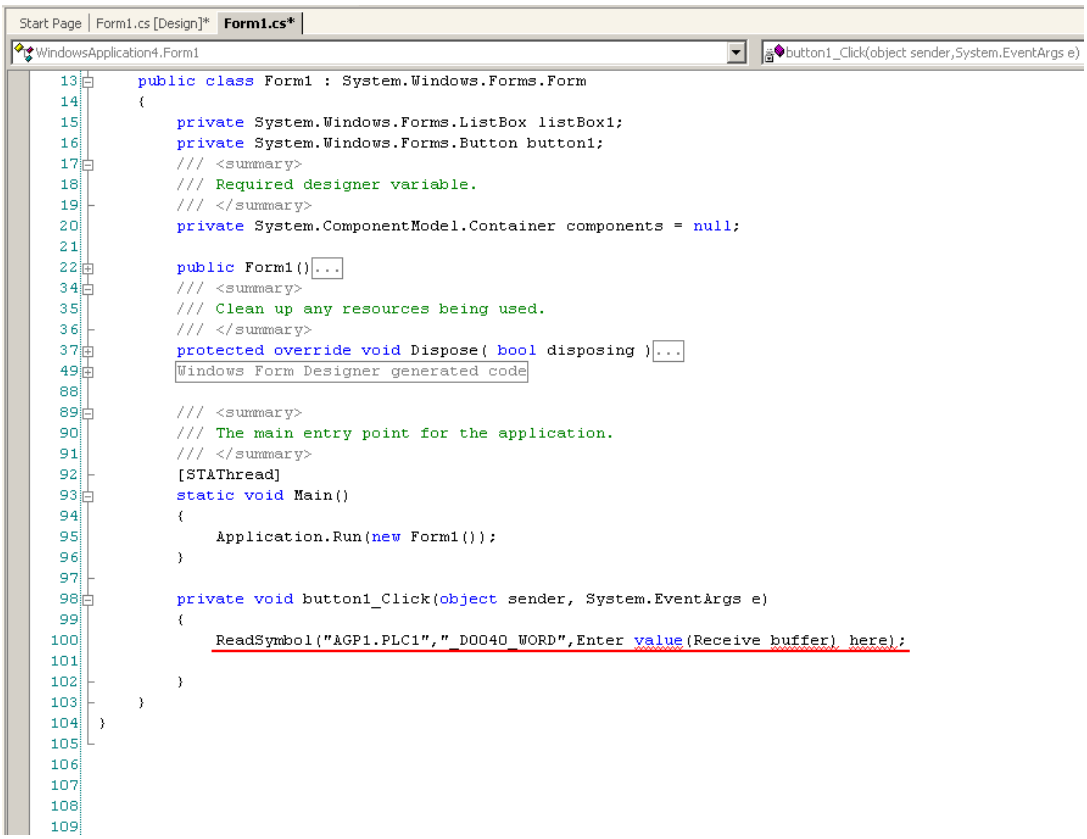
10 Select [VC++] - [Read Function] from the [Programming Assist] menu.



The read function is copied to the clipboard.

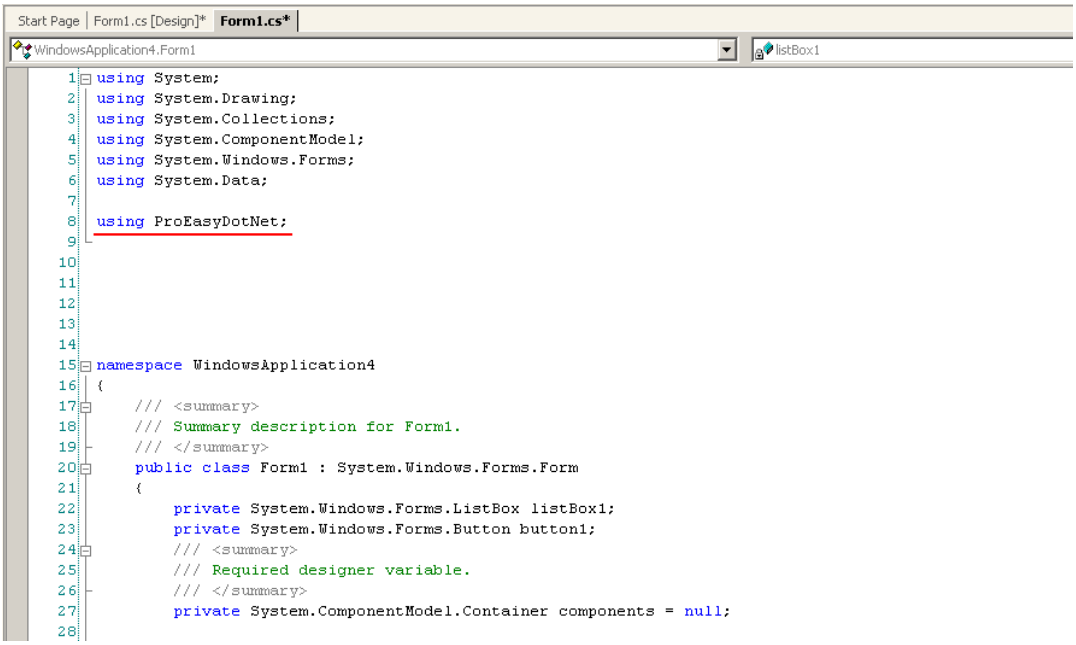


11 Double-click [button1] in [Form1], and paste the clipboard data (read function) below the [button1_Click] method ("private void button1_Click..." character string).



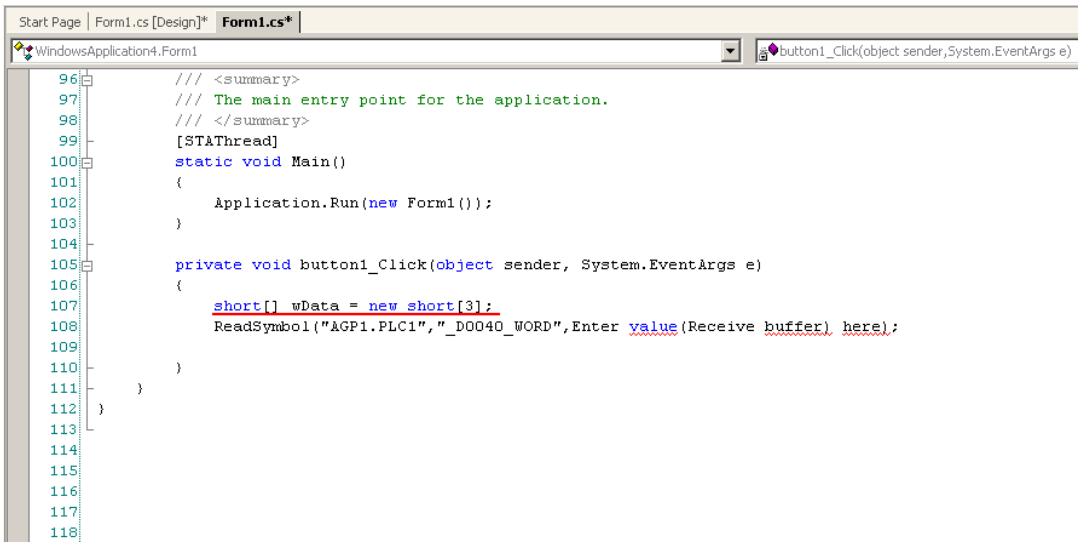
12 Describe the ProEasyDotNet directive.

Enter "using ProEasyDotNet;" at the bottom of the lines that state "using..." at the head of the source code.

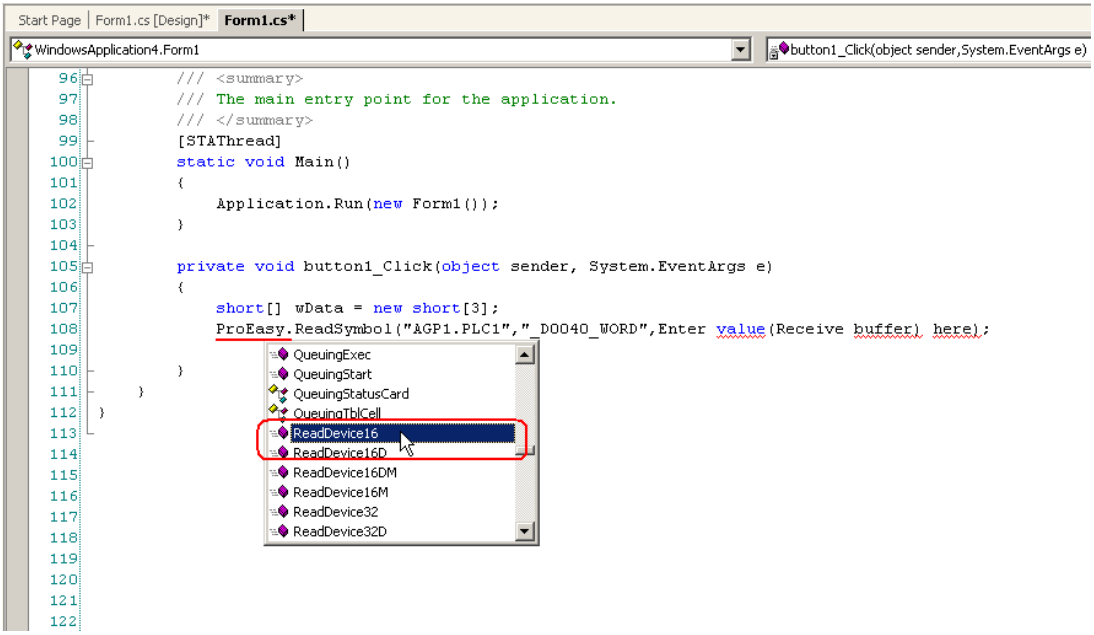


13 For the read data storing area, declare a variable "wData".

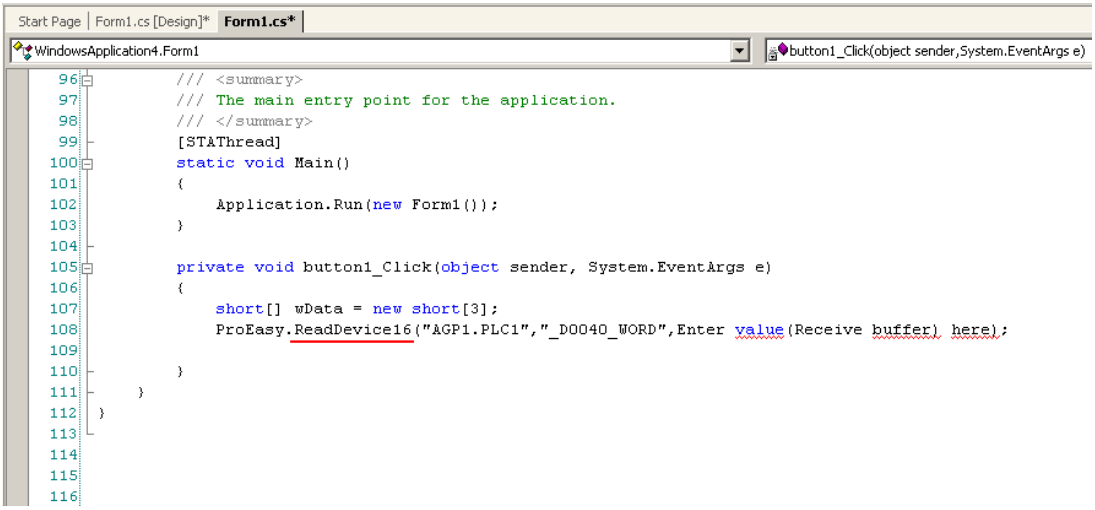
The array type ("Short" in this example) must conform to the data type of the target symbol. Specify the same data length as the target symbol ("3" in this example).



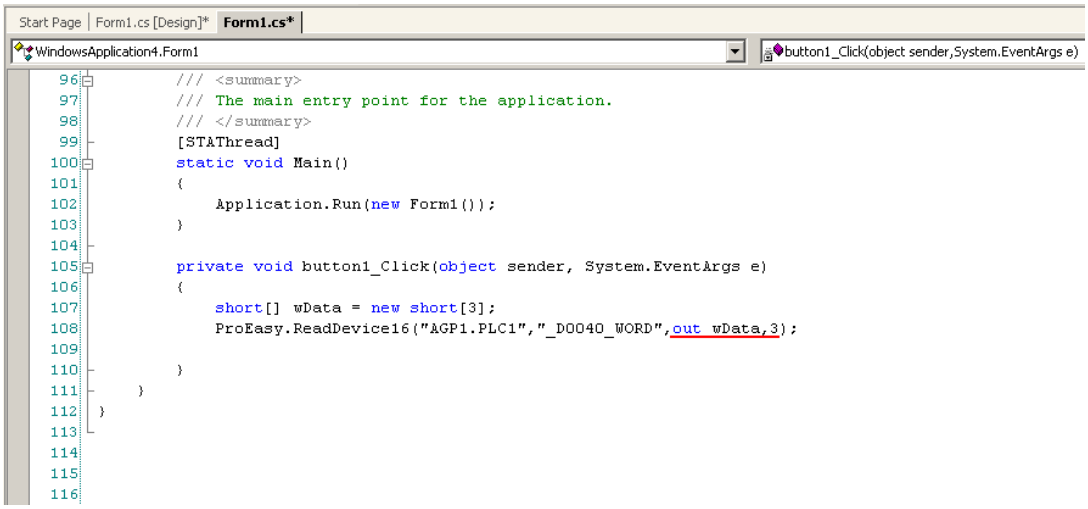
14 Enter "ProEasy." before "ReadSymbol", and select [ReadDevice16] from the displayed list box.



15 Delete "ReadSymbol" from the character string (read function) that has been pasted from the clipboard.

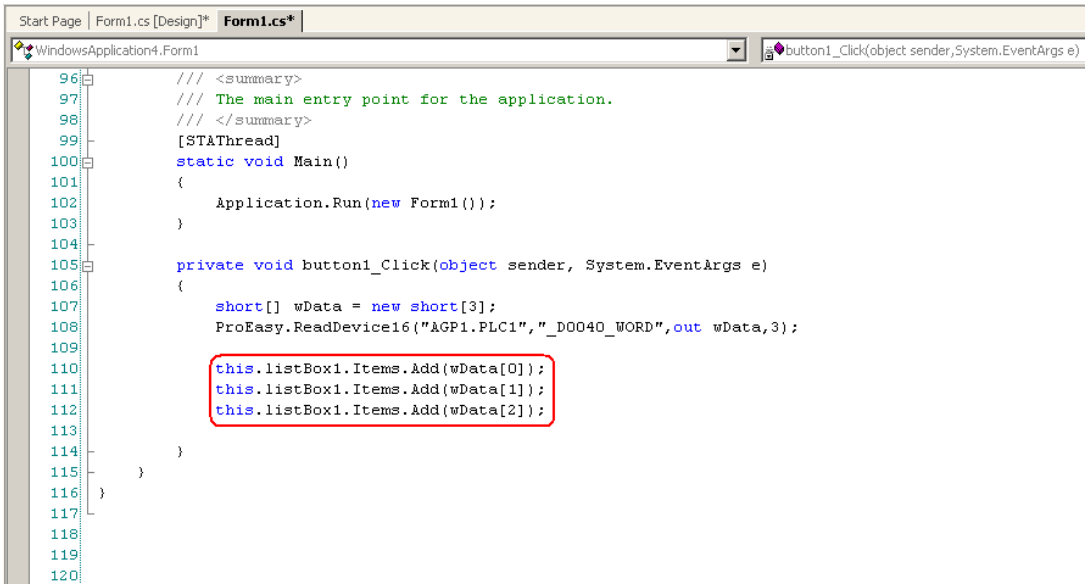


16 Specify a data storing area "wData" with the reference modifier (out), as the third argument. Enter "," (comma) at the end of the third argument, and then enter "3" to specify the length of the target symbol as the fourth argument.



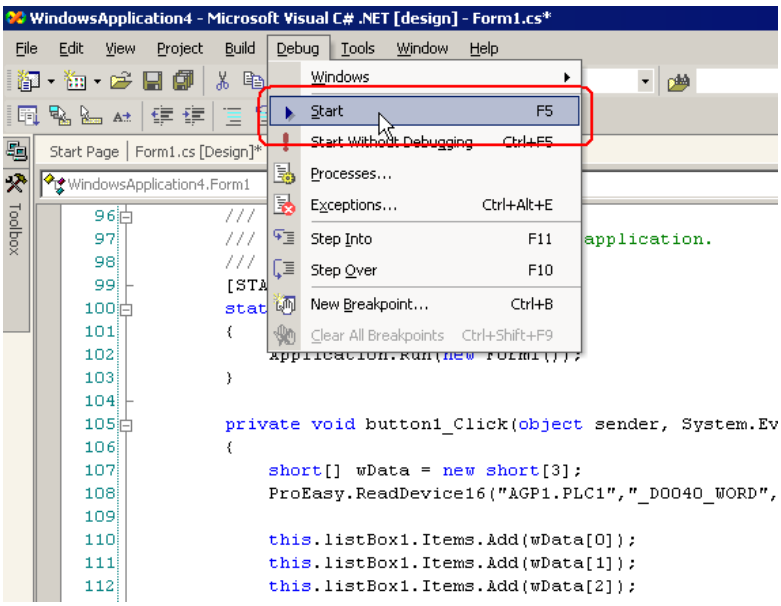

```
96          /// <summary>
97          /// The main entry point for the application.
98          /// </summary>
99          [STAThread]
100         static void Main()
101         {
102             Application.Run(new Form1());
103         }
104
105         private void button1_Click(object sender, 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 Add the read data on three items (wData[0], wData[1], wData[2]) into [listBox1] in this order.

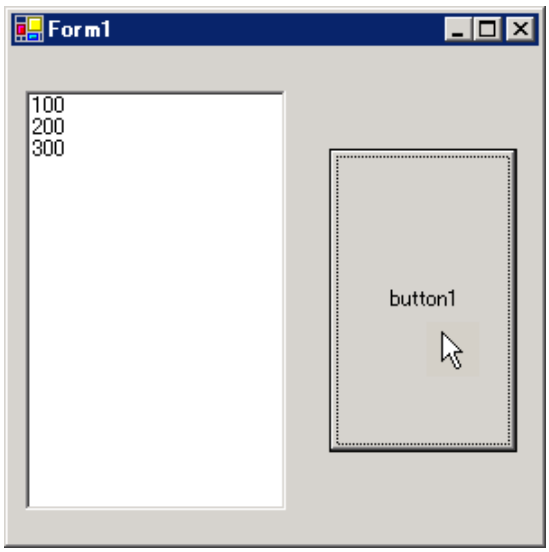



```
96          /// <summary>
97          /// The main entry point for the application.
98          /// </summary>
99          [STAThread]
100         static void Main()
101         {
102             Application.Run(new Form1());
103         }
104
105         private void button1_Click(object sender, 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 Select [Start] from the [Debug] menu.

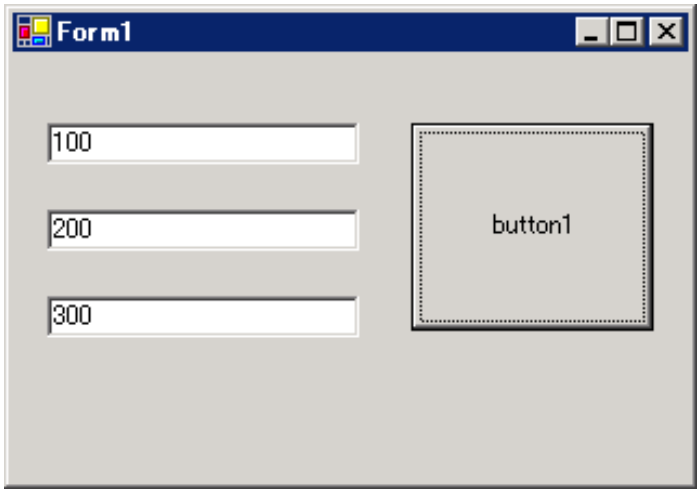


19 If you click [button1], the target symbol data (three items) are displayed in [ListBox].

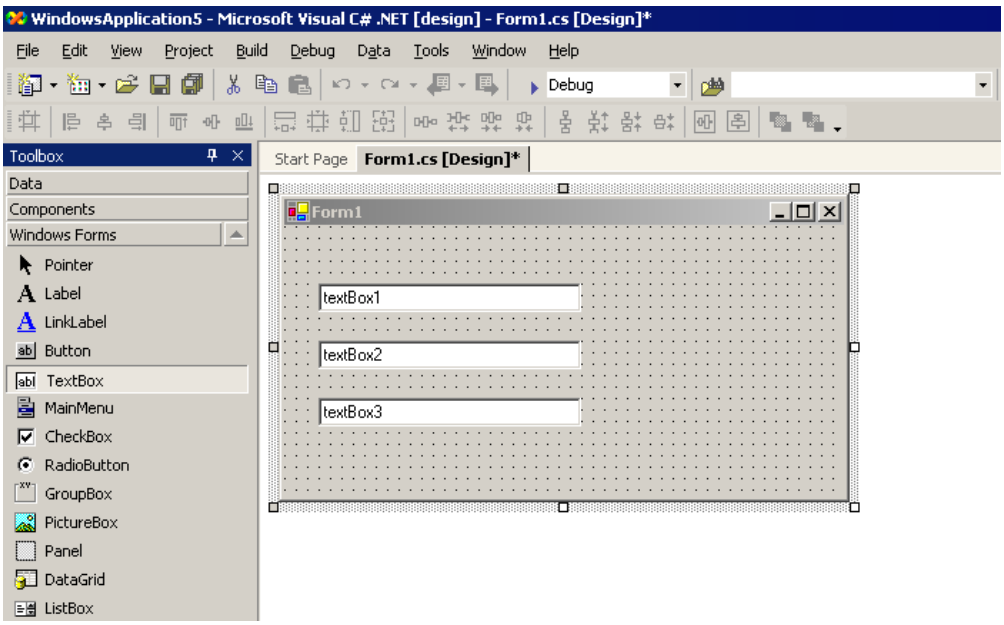


Creating "Writing" application

This section describes the application that writes data (signed 16 bits) on three items when you click [button1].

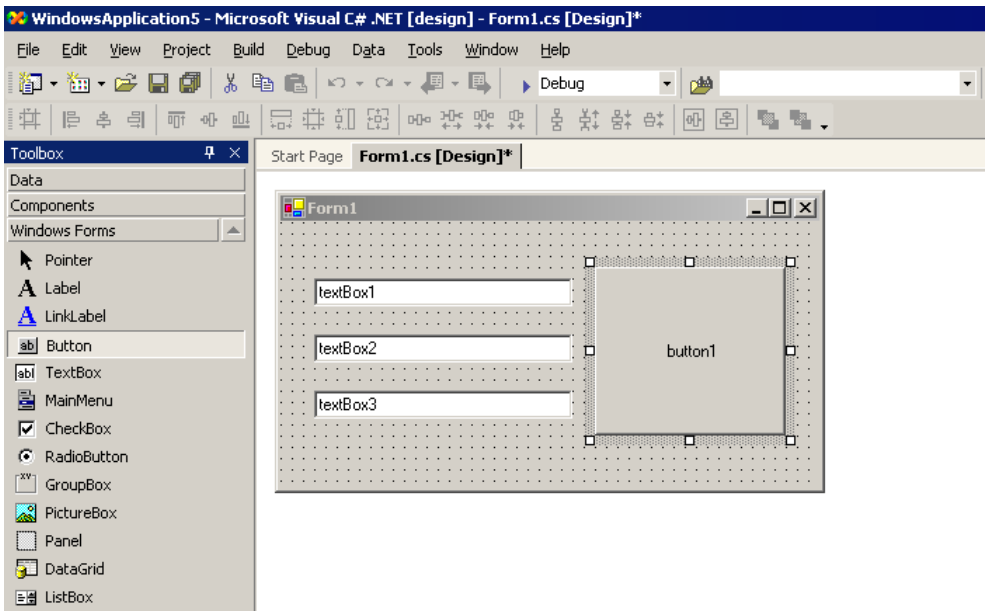


20 After selecting [TextBox] in [Toolbox], clip and paste three text boxes onto [Form1].

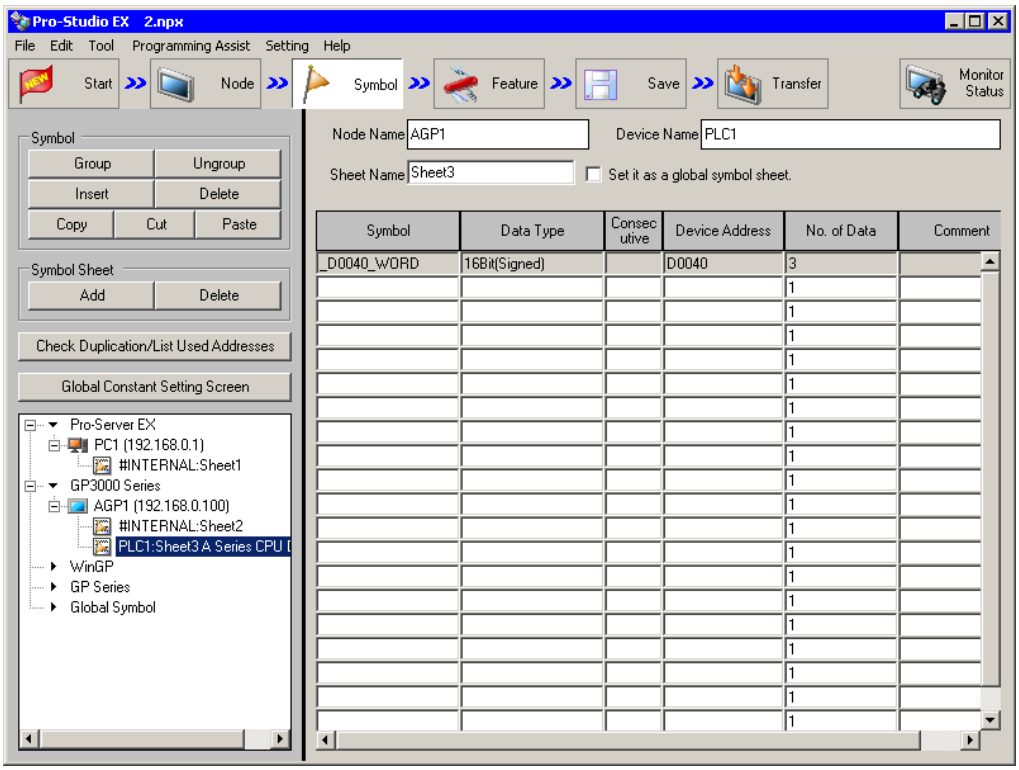


* If [Toolbox] is not displayed, select [Toolbox] from the [View] menu.

21 After selecting [Button] in [Toolbox], clip and paste it onto [Form1].

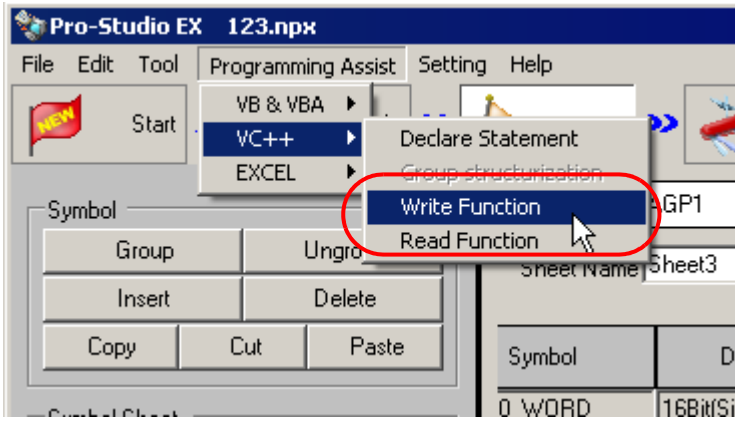


22 Select a desired write symbol name from the symbols that have been registered in 'Pro-Studio EX'. (Select the first writing area.)

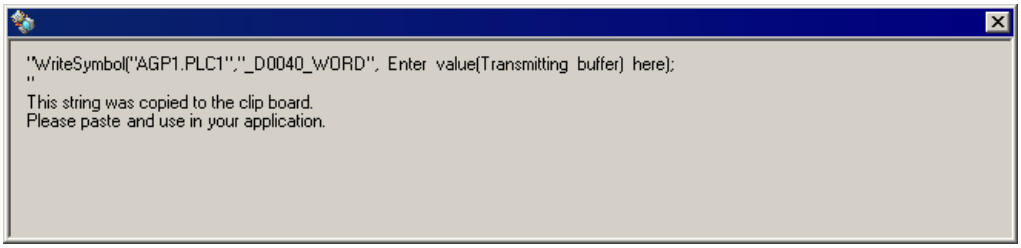


* The above example shows the symbol for the data type of [16Bit (Signed)] and the data quantity of "3".

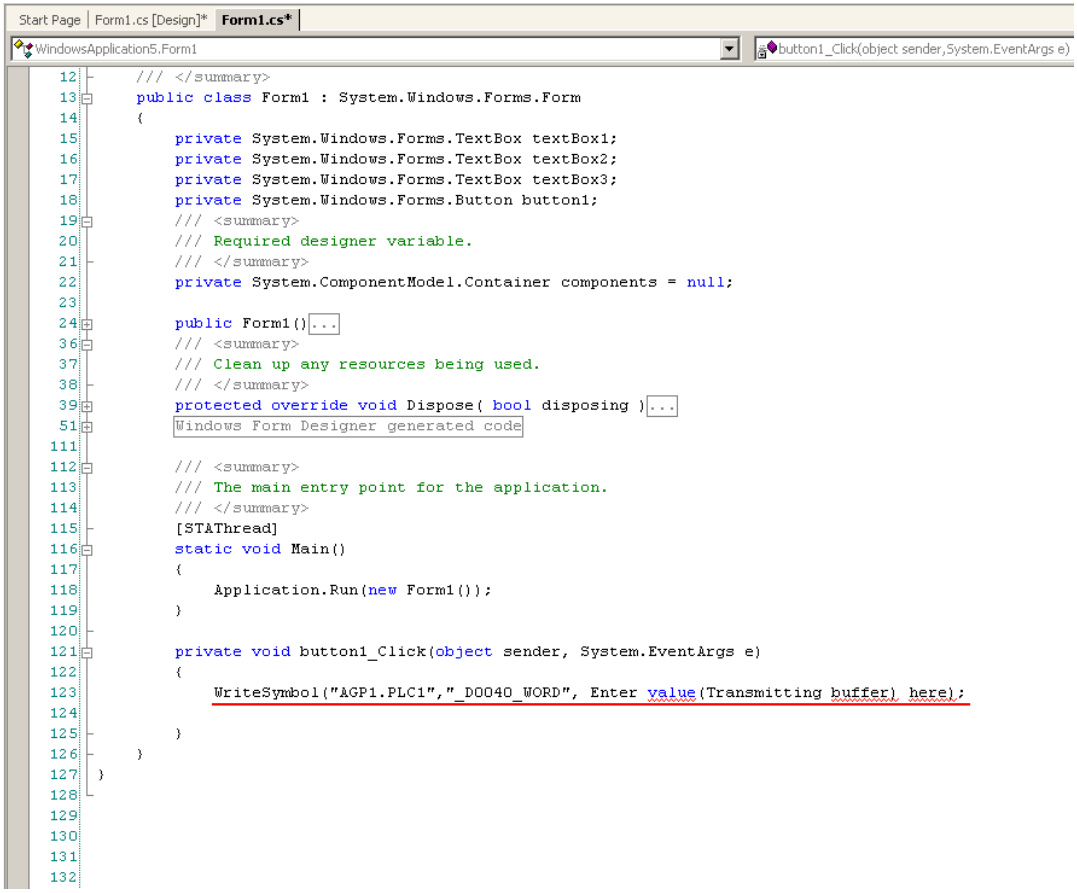
23 Select [VC++] - [Write Function] from the [Programming Assist] menu.



The write function is copied to the clipboard.

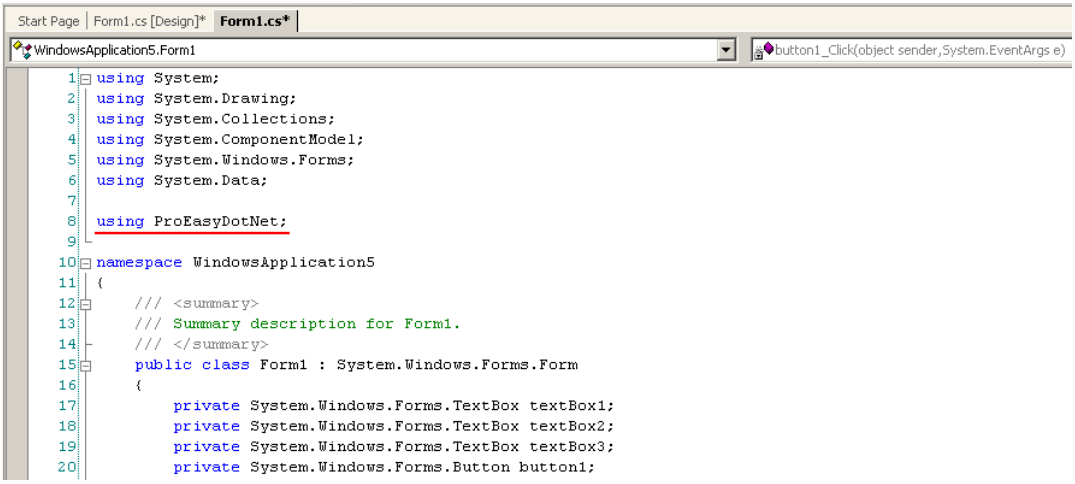


24 Double-click [button1] in [Form1], and paste the clipboard data (write function) below the [button1_Click] method ("private void button1_Click..." character string).



25 Describe the ProEasyDotNet directive.

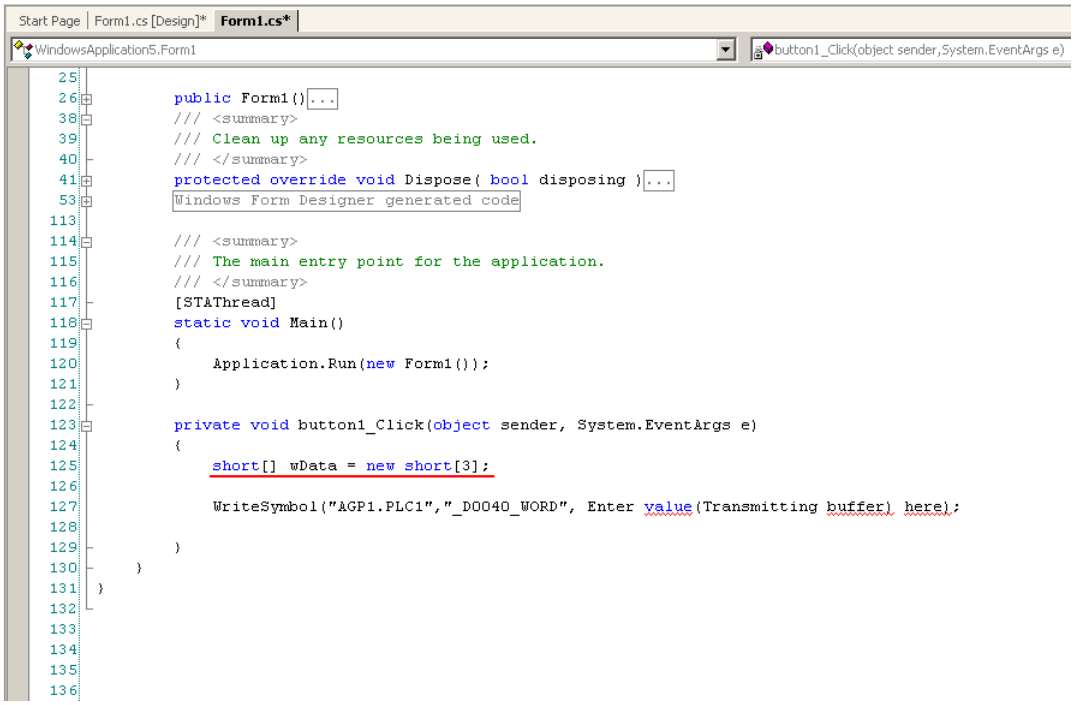
Enter "using ProEasyDotNet;" at the bottom of the lines that state "using..." at the head of the source code.




```
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;
```

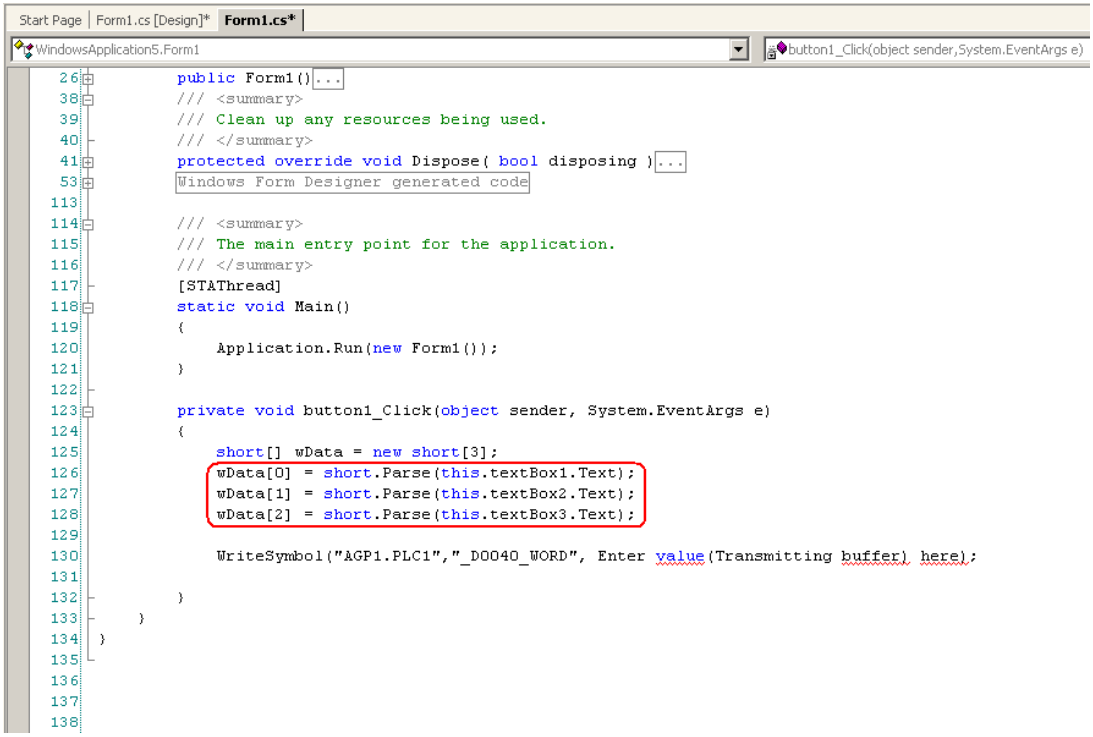
26 For the write data storing area, declare a variable "wData".

The array type ("Short" in this example) must conform to the data type of the target symbol. Specify the same data length as the target symbol ("3" in this example).



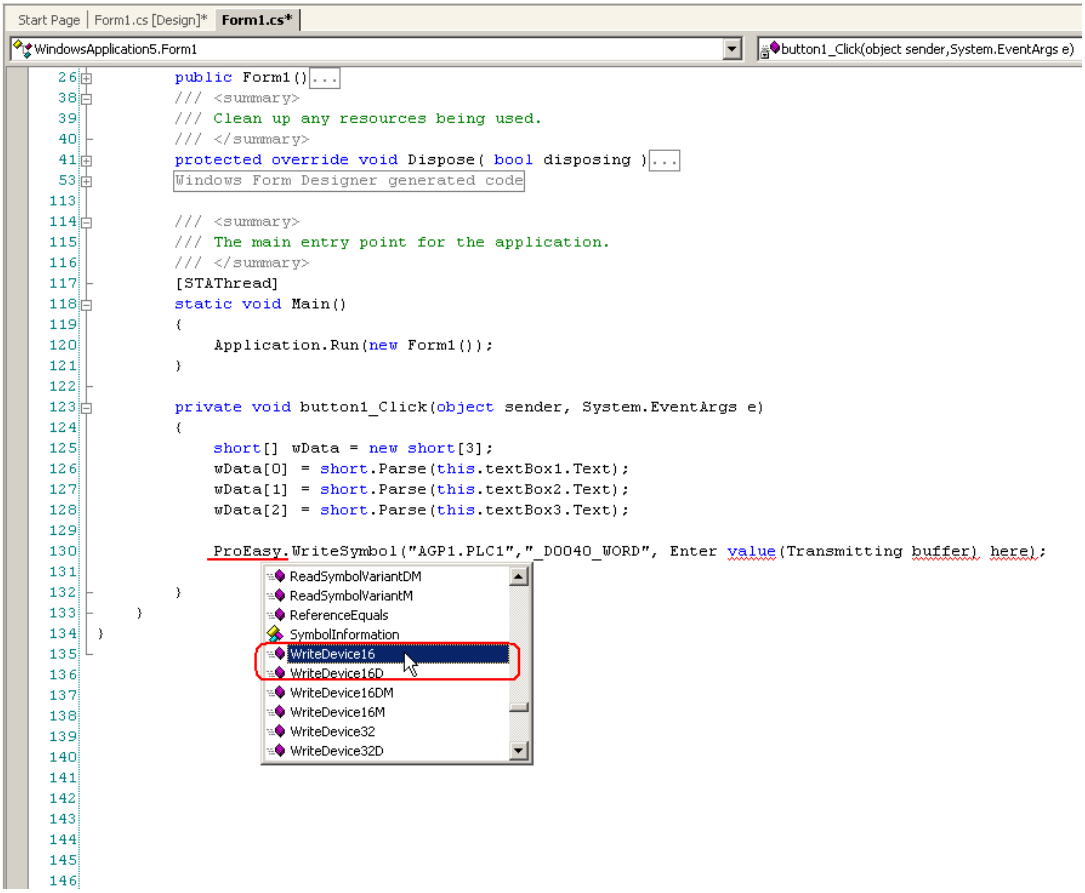

```
25
26          public Form1()...
38          /// <summary>
39          /// Clean up any resources being used.
40          /// </summary>
41          protected override void Dispose( bool disposing )...
53          Windows Form Designer generated code
113
114         /// <summary>
115         /// 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 Set the data to be entered in [textBox1] to [textBox3] in the array.

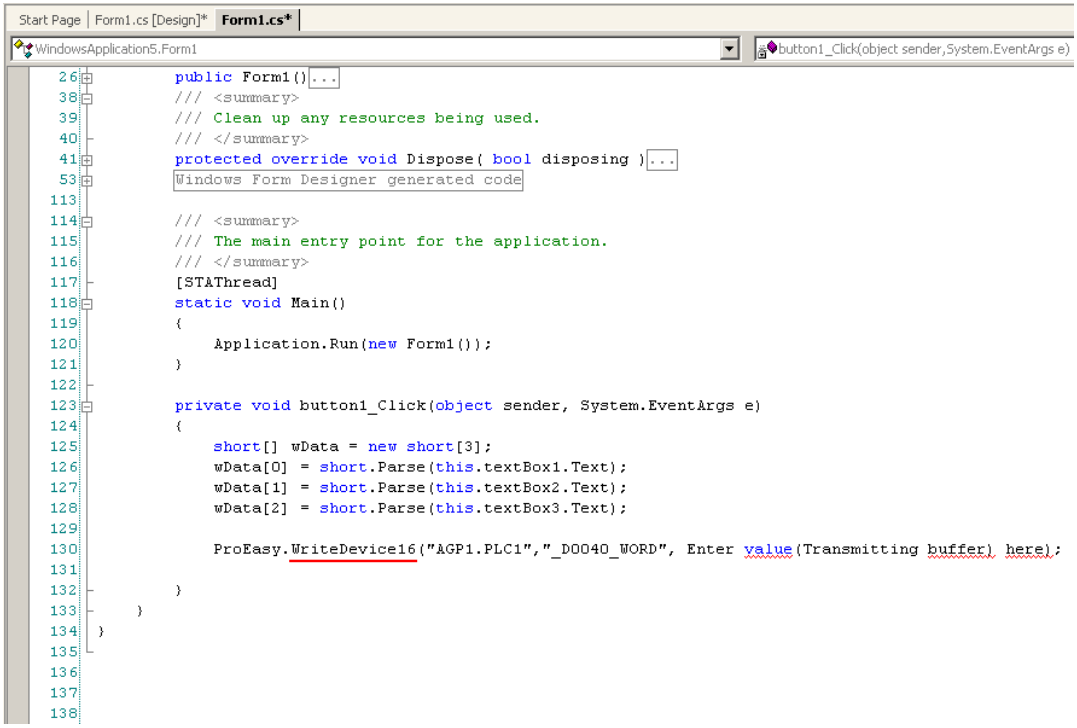


```
Start Page | Form1.cs [Design]* | Form1.cs*
WindowsApplication5.Form1 | button1_Click(object sender,System.EventArgs e)

 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
```

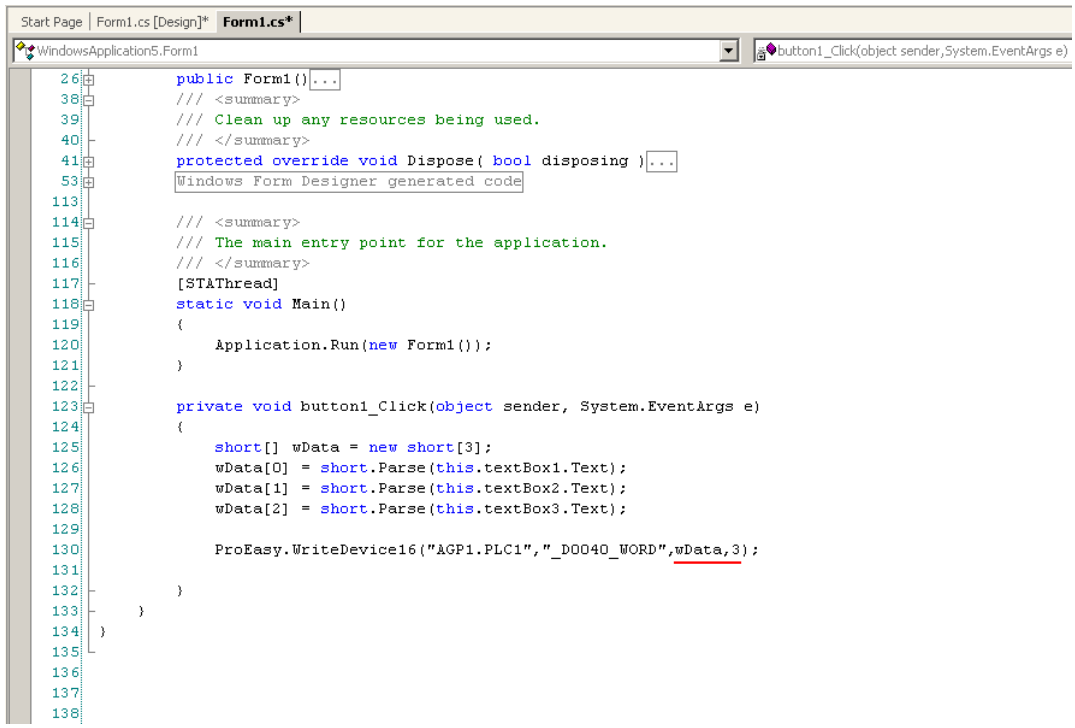
28 Enter "ProEasy." before "WriteSymbol", and select [WriteDevice16] from the displayed list box.



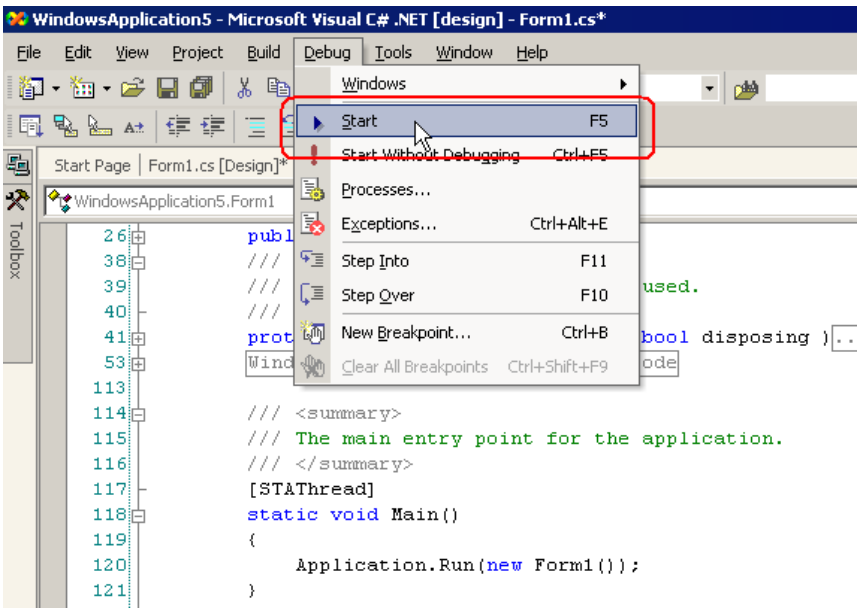
29 Delete "WriteSymbol" from the character string (write function) that has been pasted from the clipboard.



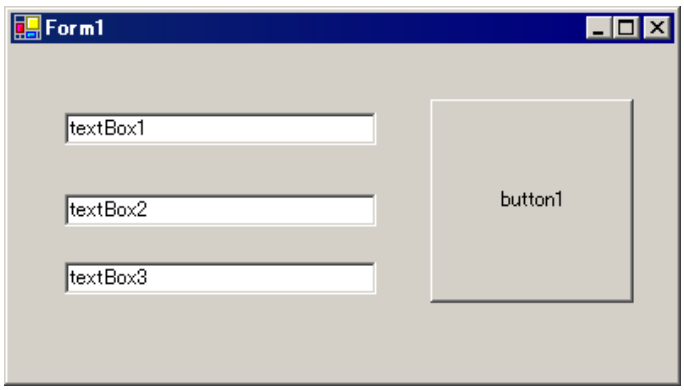
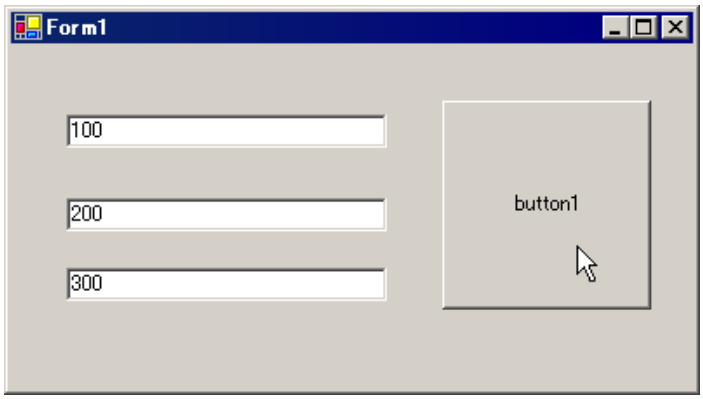
30 Specify a data storing area "wData" as the third argument. Enter "," (comma) at the end of the third argument, and then enter "3" to specify the length of the target symbol as the fourth argument.



31 Select [Start] from the [Debug] menu.



32 Immediately after startup, a character string "textBox*" is displayed in [TextBox].



After entering the write data (three items) in [TextBox], click [button1]. Then, the data will be written into the area specified with the symbol.